Professional Expertise Distilled

Applied SOA Patterns on the Oracle Platform

Fuse together your pragmatic Oracle experience with abstract SOA patterns with this practical guide

Sergey Popov

[PACKT] PUBLISHING enterprise professional expertise distilled

Applied SOA Patterns on the Oracle Platform

Fuse together your pragmatic Oracle experience with abstract SOA patterns with this practical guide

Sergey Popov

[PACKT] PUBLISHING enterprise
professional expertise distilled

BIRMINGHAM - MUMBAI

Applied SOA Patterns on the Oracle Platform

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its ealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: August 2014

Production reference: 1050814

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78217-056-3

www.packtpub.com

Cover image by Artie Ng (artherng@yahoo.com.au)

Credits

Author
Sergey Popov

Reviewers
Mehmet Demir
Gilberto Holms
Robert van Mölken
Fabio Persico
Phil Wilkins

Acquisition Editor
Joanne Fitzpatrick

Content Development Editor
Balaji Naidu

Technical Editors
Venu Manthena
Mrunmayee Patil
Shruti Rawool

Copy Editors
Alisha Aranha
Roshni Banerjee
Janbal Dharmaraj
Gladson Monteiro

Project Coordinator
Amey Sawant

Proofreaders
Simran Bhogal
Stephen Copestake
Maria Gould
Ameesha Green
Paul Hindle

Indexers
Monica Ajmera Mehta
Priya Subramani

Graphics
Sheetal Aute
Ronak Dhruv
Disha Haria

Production Coordinator
Nitesh Thakur

Cover Work
Nitesh Thakur

About the Author

Sergey Popov is an SOA Implementation Expert, Oracle Certified Professional, Oracle Fusion Middleware Architect, certified Oracle SOA Infrastructure Implementation Specialist, and certified SOA Trainer in Architecture and Security. With over 20 years of experience in establishing enterprise collaboration platforms based on SOA and integration principles, he started with earlier Oracle DB versions while still an undergraduate in the early 90s.

After graduating with Honors from St. Petersburg Telecommunication Institute, he became part of the shipping and transportation business, initially working in Norway for a large RORO and then later with container-shipping companies such as Wilh. Wilhelmsen ASA and Leif Høegh as an Integration / SOA Developer and Architect. During this technology-shifting period, when EDI was initially enhanced and later replaced by XML, a number of solutions were provided for message brokering, enterprise application integration, and public services implementation. By adopting the emerging SOA principles, lightweight service brokers were implemented, handling around 100 to 1,000 messages daily in all possible formats and protocols. With new Oracle products that were launched in early 2,000s, new technological solutions were tried and realized, based on Service Repositories and Enterprise Orchestrations.

Upon joining Accenture, Nordic, new opportunities emerged for him with regards to the implementation of the SOA methodology and Oracle-advanced products across Scandinavia and Northern Europe. Sergey was an Architect, responsible for enabling the service of a massive installation of Oracle E-Business Suite at Posten Norge, the largest Scandinavian logistics operator. Several OFM 10*g* products were employed in order to achieve the desirable high throughput. The project was considered successful by both the client and Oracle. Providing message-brokering solutions at TDC, Danish Telecom, and designing the entire SOA infrastructure blueprint for DNB NORD bank were other significant tasks that he accomplished at the time.

As a certified trainer in several SOA areas, Sergey in recent years has been engaged in providing extensive multipath training to highly skilled architects, participated as a speaker at SOA Symposium, and published several articles for Service Technology Magazine, which is dedicated to the optimal Service Repository taxonomy.

As an Enterprise SOA/SDP Architect at Liberty Global (LGI), Sergey participated in the implementation of the Pan-European Service Layer for the entire telecom enterprise, based on optimal combinations of various SOA patterns. The benefits of the SOA methodology allow you to combine Oracle Fusion products with the best-in-breed from Security and ESB platforms (Intel, Fuse, and ServiceMix). The success of this course would not have been possible without great efforts from the TMNS development and implementation team.

> Nine chapters that cover all the major SOA frameworks along with all the fundamental patterns cannot be written just in 10 months; they're the result of more than 10 years of practical experience, and all this time my wife Victoria has been supporting me, diligently and with ever-lasting patience.

About the Reviewers

Mehmet Demir is a TOGAF-certified Enterprise Architect with more than 15 years of experience in designing systems for large companies. He has hands-on experience in developing and implementing SOA-based solutions using Oracle Fusion Middleware, WebCenter Portal, WebCenter Content, BEA WebLogic/AquaLogic product technologies, and Oracle Identity Access Management Suite. As an Oracle-certified SOA Architect, IBM-certified SOA Designer, BEA-certified Architect, and Oracle WebCenter 11*g* Certified Implementation Specialist, Mehmet focuses on developing high-quality solutions using best practices. He is currently working for EPAM, Canada, as an Enterprise Architect, delivering high-value IT solutions to many of Canada's most prominent companies such as CIBC, Home Hardware, and Bell TV. Prior to EPAM, Mehmet worked for BEA Systems where he had been a principal member of the Canadian consulting team. In addition to his technical capabilities, Mehmet has an MBA from Schulich School of Business and is a certified Project Manager with PMI's PMP designation. Mehmet can be contacted at `http://ca.linkedin.com/in/demirmehmet`.

I would like to thank my beautiful wife Emily and my sweet daughters Lara, Selin, and Aylin for their support.

Gilberto Holms is currently working as an IT Architect at Multiplus SA, a Brazilian loyalty program company. He has around 8 years of experience in the software development industry, working on Java and Middleware technologies, and has been the Lead Architect for many JEE, SOA, and BPM solutions. In his current role at Multiplus, he works on the architecture, design, and implementation of strategic IT solutions that are mainly based on Oracle SOA and BPM technologies. Currently, he is particularly interested in API development, SOA enterprise governance, artificial intelligence algorithms, and open source projects. He regularly writes technical articles on SOA, BPM, Middleware, and Java on his blog, `http://gibaholms.wordpress.com/`.

Robert van Mölken is a Senior Oracle Integration Specialist with emphasis on building service-oriented business processes. He has over 6 years of experience in Oracle's SOA Suite and Service Bus where his speciality is with BPEL, SCA, SOAP, XPath, XQuery, XML, Java, JAX-WS, Advanced Queuing, and PL/SQL. Since 2007, he has had experience in dealing with Oracle SOA Suite 10*g* and later with SOA Suite 11*g*. Last year, he joined the Oracle SOA Suite 12*c* Beta and presented a new Fusion Middleware 12*c* product called Managed File Transfer along with the Product Manager at Oracle OpenWorld. He is also an active blogger on the technology blog of AMIS Services where he writes about SOA, testing, and the Internet of Things. Robert works at AMIS located in the Netherlands. AMIS helps partners to use the investments they put in to Oracle technology as effectively and economically as possible, and contributes to the success of their organization. AMIS is the Oracle knowledge partner in the Netherlands. This is evident from the world's leading weblog, `http://technology.amis.nl/`, the level of knowledge, projects, employees, and Oracle awards.

Fabio Persico was born in Sorrento in the south of Italy in 1981. After completing 2 years of an MSc in Computer Science in 2006, he got involved in the Oracle world through an internship of 9 months with Oracle, Italy. As an apprentice at Oracle, he had the chance to learn more about the J2EE platform and some Oracle products such as Oracle Database and the SOA Suite. After that, he continued to work with Oracle and got fully involved mainly in the SOA stack, working for many customers from different areas. He's been working with infoMENTUM Limited since 2012, where he is mainly playing the role of a developer/architect in a project based on the Oracle FM/SOA stack. Fabio is an Oracle Certified Specialist consultant.

I would like to thank Sergey Popov, the writer, for giving me the opportunity to work with him by reviewing this SOA patterns book. It has been a great experience, and I really enjoyed all the best practices that the author has shared with the reader.

Phil Wilkins has spent nearly 25 years in the software industry, working with both multinationals and software startups. He started out as a developer and has worked his way up through technical and development management roles. His last 12 years have been primarily in Java-based environments. He now works as an Enterprise Technical Architect within an IT group for a global optical healthcare manufacturer and retailer. Outside his work commitments, he has contributed his technical capabilities to support others in a wide range of activities: from the development of community websites to providing input and support to people authoring books and developing software ideas and businesses, including reviewing a number of Java- and Oracle-related books for Packt Publishing. When not immersed in work and technology, he spends his downtime pursuing his passion for music and spending time with his wife and two boys.

I'd like to take this opportunity to thank my wife Catherine and our two sons, Christopher and Aaron, for their tolerance during the innumerable hours I spent in front of a computer, contributing to activities for both my employer and many other IT-related activities that I've supported over the years.

www.PacktPub.com

Support files, eBooks, discount offers, and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

PACKTLiB™

http://PacktLib.PacktPub.com

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Instant updates on new Packt books

Get notified! Find out when new books are published by following @PacktEnterprise on Twitter, or the *Packt Enterprise* Facebook page.

Table of Contents

Preface

Arguably, distributed computing is the most complex concept in computer science. The practical realization of this concept in the form of service-oriented computing further adds to this complexity. Generally, there are two reasons: firstly, because of a compound architectural approach, SOA is based on already complex techniques, and secondly, to stay on the cutting edge of computing technology, SOA must appeal to non-IT businesses to be successfully adopted in modern enterprises. To achieve this goal of successful adoption, SOA architects must combine a vendor-neutral approach to systems design with a deep knowledge of platforms on which the solution will be realized. This combination will allow service-oriented solutions to be flexible and resilient at the same time.

Maintaining the right balance of these two success factors is quite a challenge in the multilayered, multiframework, and compound environments of SOA. Since there are several success factors with a magnitude of problems associated with their implementation, SOA adoption requires a structural and pattern-based approach. In this book, our task is a practical demonstration of pattern-oriented problem solving based on the concrete implementation of service collaboration and integration systems in different industries (telecom, shipping, and logistics). The book goes for the most complex and, at the same time, the most common use cases. Conceivably, the most challenging problems in SOA are related to dynamic service compositions, usually assembled on runtime and in a business-agnostic way. This is the ultimate realization of the SOA Composability principle. This principle is, in turn, the foundation of the main service-orientation promise: keeping businesses agile and adaptive to any type of environmental shifts by assembling new compositions (that is, business processes) out of existing atomic services.

The general approach to achieve this, also used in every chapter of this book, is as follows:

- Find the root cause of the problem and analyze it in strong relevance to the SOA design principles.
- Speculate the decomposition of the problem into smaller, more manageable parts that could be implemented as separate atomic components or services.
- Identify the ways of standardizing the decomposed components/services, focusing on the improvement of their reuse.
- Propose various vendor-neutral solutions (not exactly Oracle) based on the identified components/services and, again, diligently analyze them using the SOA design principles, focusing on the desired SOA characteristics.
- Present the most optimal solution based on an Oracle platform and compare it to other alternatives proposed during the analysis phase. Since we are vendor-neutral and focus primarily on the preferred solution's characteristics, we cannot guarantee that Oracle realization will always win, but it will be the closest bet for most of the discussed use cases.

In order to make the first step (the problem analysis) consistent, verifiable, and undisputable, in *Chapter 1*, *SOA Ecosystem – Interconnected Principles, Patterns, and Frameworks*, we introduce you to the SOA principles and the areas of their application. It is important to see these principles interconnected as their relations are not always straightforward and we should be very careful in balancing them in different frameworks and service layers. Some key SOA standards will also be discussed with the focus on those employed in the composition controllers design.

Logically, following the architectural two-folded task, after discussing the vendor-neutral SOA aspects, we look at the Oracle product's portfolio and see how it can help us in achieving the goals of service orientation. The introduction to the characteristics of Oracle Fusion Middleware will help us in the chapters to follow, when building practical solutions around Agnostic Composition controllers for different companies. Importantly, we will not jump into Oracle realization at the very beginning of every chapter (this part is dedicated to a certain SOA framework). Instead, we will look very closely at every alternative, check its feasibility, and see how common solutions (in the form of patterns) can help us in mitigating common problems for these frameworks.

What this book covers

Chapter 1, *SOA Ecosystem – Interconnected Principles, Patterns, and Frameworks*, sets the tone for the entire book, presenting the main SOA frameworks in relation to individual SOA characteristics and goals. To achieve these goals, we will discuss the SOA design principles, their dependencies, and roles in maintaining a robust SOA ecosystem. For a better understanding of the importance of these principles, we will start by presenting a practical and quite realistic use case, depicting the disaster that may follow when design principles are sacrificed to achieve short-lived tactical goals. These problems will be further analyzed during the course of this book and individual SOA patterns will be offered as proven solutions within every individual SOA framework. The practical outcome of this chapter will present you with a complete set of SOA frameworks and SOA Service Inventory patterns, which help shape the Service Inventory according to the presented frameworks.

We suggest that everyone, even seasoned veterans familiar with the concept of service orientation, begin with this chapter. Here, we establish the glossary and architectural vocabulary, essential not only to understand further material but also for your day-to-day technical communications. This chapter also sufficiently presents fundamental materials to prepare for the Certified SOA Professional examinations (`http://www.soaschool.com/certifications/professional`).

If you are an Oracle practitioner and familiar with the modern Fusion Middleware stack, you can skip the next chapter and proceed directly to service composition patterns, described in *Chapter 3*, *Building the Core – Enterprise Business Flows*, and *Chapter 4*, *From Traditional Integration to Composition – Enterprise Business Services*. If you already have hands-on experience with Agnostic Composition controllers and dynamic service invocation, we suggest that you first read *Chapter 5*, *Maintaining the Core – the Service Repository*, which explains the role of reusable service artifacts and Service Repository in runtime discoverability.

Chapter 2, *An Introduction to Oracle Fusion – a Solid Foundation for Service Inventory*, provides a list of Oracle products (OFM stack) and methodology(Oracle AIA+FP with Foundation Pack) that fit the pattern/frameworks matrix, presented in *Chapter 1*, *SOA Ecosystem – Interconnected Principles, Patterns, and Frameworks*. This chapter explains the roles of the tools and the Oracle roadmap in support of the SOA principles. Most importantly, it explains how Oracle products support SOA `WS-*` standards (WS-ReliableMessaging, WS-Coordination, WS-BPEL, WS-Addressing...) and how this fact aids in pattern implementation. Information from this chapter will help architects in setting realistic requirements and composing a proper RFI matrix for Oracle products in relation to the SOA frameworks.

Chapter 3, Building the Core – Enterprise Business Flows, first presents the SOA platform's refactoring initiative, undertaken in a large-scale telecom enterprise, aiming for the optimization of a complex multinational Service Inventory. Traditionally, the first target is complex long-running processes, most commonly, those based on BPEL. Oracle SOA Suite is perhaps the most mature tool for this job, but it is still widely misinterpreted by many developers and architects. This chapter will explain how to maintain the right balance using the four SCA components, minimize pressure on the BPEL dehydration store, achieve optimal performance, and improve agility of the composition logic using the Agnostic Composition controller. The chapter's practical outcome will be the Service Broker, suitable to handle dynamically synchronous and asynchronous service compositions.

Chapter 4, From Traditional Integration to Composition – Enterprise Business Services, continues discussion of the Telecom primer started in the previous chapter by addressing the separation of the concerns principle and untying the Agnostic Composition controller from the Orchestration platform and Enterprise Service Bus. This chapter will demonstrate how to build business-agnostic composition controllers on OSB to dynamically route messages and coordinate transactions in a reliable manner for synchronous and fast-running services. The roles of all ESB-related SOA patterns are explained in great detail.

Chapter 5, Maintaining the Core – the Service Repository, demonstrates how to design, collect, maintain, and access service metadata from the very beginning of the SOA project until the service is decommissioned at the end of the lifecycle. You will be presented with a lightweight service taxonomy, essential to maintain the service composition logic in the composition controllers designed in previous chapters. From a broader perspective, this chapter sets the basis for effective SOA Governance, presenting all SOA Foundational Inventory patterns and their implementation using Oracle Service Repository and Registry. The DB realization of a flexible service taxonomy will be the practical outcome of this chapter.

Chapter 6, Finding the Compromise – the Adapter Framework, discusses ways to balance and optimize the adapter framework in Enterprise Service Inventory. Oracle has the most advanced adapter framework for applications, protocols, and resources. This chapter will demonstrate what frameworks and tools (OSB or SCA) are the best candidates for patterns implementation and how to avoid the most common mistake, creating hybrid services. We also discuss in considerable detail ways to avoid adapters as a non-SOA approach through interface standardization.

Chapter 7, Gotcha! Implementing Security Layers, explains how services can be designed in a secure way from the very beginning. The core aspects of service security design are highlighted, starting from vulnerabilities and risk analysis to common attack types and risk mitigation methods. These aspects are presented from the attacker's and security architect's sides; the SOA Security pattern's role is demonstrated from components up to the Security Gateway levels.

Chapter 8, Taking Care – Error Handling, completes the Agnostic Composition controller design, started in *Chapter 3, Building the Core – Enterprise Business Flows*. Here we will demonstrate how complex recovery scenarios can be implemented using the standard Oracle Fault Management framework and custom composition controllers, acting as automated recovery tools. With the focus on proactive service monitoring and error prevention, we will discuss the SOA patterns that can contribute to one of the most complex SOA problems—recovery of the composite business service composed agnostically.

After completing the preceding chapters and gaining some practical experience in SOA implementations, you will be equipped to attain the Certified SOA Architect level (`http://www.soaschool.com/certifications/architect`).

Chapter 9, Additional SOA Patterns – Supporting Composition Controllers, concludes the book by presenting complex SOA patterns, realized on very interesting Oracle products: Coherence and Oracle Event Processing. Combined in line with the SOA patterns and enhanced by the business monitoring tool (BAM), these products present a new Oracle approach in the event-driven architecture—fast data. Using a logistics example, we will discuss how an event-driven network approach and Oracle CQL can improve data processing and business decision services in complex distributed environments.

What you need for this book

To implement solutions based on the examples in this book, install Oracle SOA Suite 11*g* Patch Set 6 (11.1.1.7). Also, for *Chapter 4, From Traditional Integration to Composition – Enterprise Business Services*, and *Chapter 6, Finding the Compromise – the Adapter Framework*, Oracle Service Bus (11.1.1.7) is needed. Oracle DB 11*g* (or 12*c*) is a prerequisite for any installation, but it will be used as a standalone tool for examples discussed in *Chapter 5, Maintaining the Core – the Service Repository*, and *Chapter 6, Finding the Compromise – the Adapter Framework*. A better understanding of the concept of Enterprise Service Repository, Oracle SR 11*g*, and Registry would be useful in *Chapter 5, Maintaining the Core – the Service Repository*. Oracle API Gateway (formerly, Oracle Enterprise Gateway, Release 11.1.2.2.0) is discussed in *Chapter 7, Gotcha! Implementing Security Layers*, and you could have it installed (optionally) to better understand the security patterns discussed in this chapter.

Who this book is for

Some say experience is something you don't get until you stop needing it. This book is what an established professional of today would have wanted to read at least ten years ago. Here you will find my combined experience of at least 15 large-scale service-oriented projects in three industries. Successful implementations were recognized by not only clients, but also Oracle. I really admire the skills and ingenuity of professionals who have worked together on the implementation of the described concepts. I believe that the presented materials will be useful for experts working at different levels:

- SOA architects working on Oracle products—from the solution to enterprise levels—will get a comprehensive guidance on how to apply an SOA practice on the Oracle platform.
- SOA architects practicing the vendor-neutral approach (although Java is not purely neutral anymore) will find enough materials on patterns, methods, and realizations of efficient and low-cost solutions for small- and mid-sized enterprises.
- SOA DevOps team leads will learn how to manage Oracle Fusion projects using both the Agile or Waterfall methodologies. Code snippets presented in the book are more than enough for developers to get going with their own implementation.

If you are looking for study materials on the SOA architecture to pass the vendor-neutral exams (such as SOACP SOASchool; `http://www.soaschool.com/`), this book should be sufficient to attain the Certified SOA Architect status. In fact, having the Certified Trainer status, we were asked several times to prepare for combined SOA school lectures, which condensed SOA architecture, analytics, and security courses into a one-week intensive training for experienced architects. In many aspects, *Applied SOA Patterns on the Oracle Platform* is the lecture material we use for these purposes. We should also mention that we use these materials actively in our day-to-day activities.

Please bear in mind that despite the numerous technical examples, this is not a Cookbook or Programmer's Guide (such as `http://www.packtpub.com/oracle-service-bus-11g-development-cookbook/book`). You can find plenty of them for every Oracle product we use in this book on the official Packt Publishing website. For a better understanding of the presented materials and examples, you must be familiar with the SCA concept (in particular, BPEL and Mediator), Rule Engines, and Oracle Service Bus (the implementation of proxies is a must). The common prerequisites include some Java skills (EJB and Servlets), XML, and PL/SQL. Nevertheless, we strive to present all the concepts in the most comprehensive manner and you will find plenty of references to the Oracle documentation and best practices.

Staying focused on the Agnostic Composition controllers, we had to rationalize the set of tools, excluding some really interesting ones such as Oracle BPM Suite. Unfortunately, it's virtually impossible to put all Oracle products from the Fusion Middleware stack into a single book; please see related books on the publisher's site.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "The `xsd:any` element is at the upper level in this hierarchy; it has an equivalent object in OOP."

A block of code is set as follows:

```
Public SearchObject getSearchResultObject() throws Exception {
   try{
       InputStream source    = getResultStream(search_url);
       Reader reader         = new InputStreamReader(source);
       Gson   gson           = new Gson();
       SearchObject response = gson.fromJson(reader,
       SearchObject.class);
       reader.close();
       return response;
    }
    catch (Exception e){
      log.error(getClass().getSimpleName(), "Error for URL "
      + search_url, e);
    }
    return null;
 }
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
//invoking parser
 execute immediate
         BEGIN '||v_parser||'(:1, :2, :3, :4 ); END;'
         USING IN ip_lob, IN v_msgid,
         OUT v_status,
         OUT v_status_text;
```

Any command-line input or output is written as follows:

```
loadjava -grant public –user <xdbuser>/<xdbuserpwd>@<XDBSID>
CustomServlet.class
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "The CTUMessage payload is our generic message container."

> Warnings or important notes appear in a box like this.

> Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Downloading the color images of this book

We also provide you a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from: `https://www.packtpub.com/sites/default/files/downloads/0563EN_ColoredImages.pdf`.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1

SOA Ecosystem – Interconnected Principles, Patterns, and Frameworks

In this chapter, we will discuss how **Service-oriented Architecture** (**SOA**) as a design approach allows us to achieve certain goals and the characteristics that have to be maintained to make these benefits feasible. The practical ways of attaining these characteristics are based on a concrete balance of very well-defined principles, and we will closely look at each one of them. This balance is maintained in specific areas of relevance and is formed in a structure of frameworks. Here, we will discuss issues that are frequently encountered within and across these frameworks, and the common patterns employed as a publicly approved way of solving these recurring problems. One of the main purposes of this chapter is to give developers and architects a matrix of the design rules (patterns) in relation to the corresponding frameworks, all based on SOA principles.

The characteristics, goals, and benefits of SOA

As an evolutionary approach, comprising the best of the architectural and technical solutions designed in the last forty years (arguably even more), SOA nowadays in many ways is quite well standardized with a well laid out vocabulary of meanings of technical terms. Along the course of the entire book, we will stick to the definition of SOA, summarized by Thomas Erl in *Service-Oriented Architecture: Concepts, Technology, and Design, Prentice Hall / PearsonPTR Publishing*.

This is also available at `http://serviceorientation.com/whatissoa/service_oriented_architecture`. The complete SOA Manifesto that was developed as a result of systematic collaboration of many experts' groups can be found at `http://serviceorientation.com/soamanifesto/annotated`.

Still, it's quite fascinating to see that debates are still being sparked and raged worldwide regarding proper terms and their meanings. We are not going to judge or participate in any form in these discussions. That's not the purpose of this book. Obviously, there is one good way to avoid that, which is to stay focused on the practical targets that SOA helps us to achieve. No wonder these goals and benefits are quite well defined and are the sole purposes and reasons why the SOA approach was proposed in the first place. Any practicing architect who has been through several projects (even if not defined as being SOA-based) could easily recollect the common requirements stated by both sides: Business and IT. Let's just quickly recollect them. So, any concrete solution should have the following properties:

- They should be kept as simple as possible while still meeting the business needs [R 1]
- They should be kept flexible and consistent to support the changing enterprise-wide business needs and enable the evolution of the company [R 2]
- They should be based on open industry standards [R 3]
- Systems and components within the proposed IT domain (architecture) will be viewed as a set of independent and reusable assets that can be composed to provide a solution for the company [R 4]
- They should be based on clearly defined, well-partitioned, and loosely-coupled components, processes, and roles [R 5]
- They should be designed for ease of testing [R 6]
- They should be based on a proven, reliable technology that is used as originally intended [R 7]
- They should be designed and developed, focusing on nonfunctional requirements right from the start [R 8]
- They should be secure; able to protect confidentiality and the privacy of all underlying resources and communications [R 9]
- They should be resilient to faults, that is, capable of staying operational even in the event of catastrophic failure of the internal components [R 10]

> These are actual consolidated requirements taken from more than ten projects and RFIs.

We could really continue on, but in general, these are the top-ten points of any requirements list, and it will be hard to go further without repeating them. Therefore, any list that is similar to this cannot be consistent with more than 15 unique statements within it. We suggest keeping these points up your sleeve until the end of this chapter. This is because at the end of the chapter, we will do some practical exercises of matching listed declarations to the capabilities of SOA. Quite often, these requirements are based on pure common sense, and some people declare them as design principles. It is hard to argue that real design principles should at least be based on common sense, but compliance to this simple fact is not enough to talk about elements from the previous list as principles. At the moment, they are just declarations of good intentions, and we, after several implementations of complex projects, know quite well what road is paved with them. We will talk about the definition of principles a bit later in a considerable amount of detail, but now it's important to analyze these wishes and find what's common there and how it is relevant to the service-oriented approach. It is quite simple to see that the whole list (with one small exception, which is just to confirm the general rule) can be divided into two categories. These categories are related to effort (first, third, fourth, fifth, sixth, seventh, and eighth items) and time (second and seventh items), with the seventh item equally relevant to both effort and time.

Standing a bit aside, the ninth item, generally described as compliance by security policies, is nothing more than pure money, as almost no one these days seeks fun in simple informational vandalism. All security breaches aim to steal your information, that is, money, and therefore, put you out of business. As a consequence, it's needless to say that time and effort can essentially be compared to money as well. So, unsurprisingly, everything boils down to the same logical end, that is, money, which is the key; we have learned this many times, when talking to the bosses (CIO, CEO, project manager, and so on). As stated previously, in IT, money comes in two general ways: either we consistently shorten the delivery cycle or reduce operational costs.

Firstly, we would like to place strong affectation on the word consistently, otherwise, all delivered solutions will tend to be quick and dirty with rocketing operational costs. The two ways (mentioned previously) don't need to be exactly inversely proportional, and proper balance can be found, as we will see later.

A shortened delivery cycle simply means that we will strive to employ the existing reusable components if feasible. Also, every new component or element of the infrastructure that we will add to our inventory will be designed, keeping reusability features in mind. The good rate of return from previous investments (that is, ROI) is the main direction of implementation for new products. At the same time, a higher level of reuse denotes a lower number of heterogeneous components and elements in the infrastructure.

A less diverse technical infrastructure with more standardized components tends to be more predictable and consequently more manageable, and the lowering **Total Cost of Ownership** (**TCO**), which is the key part of operational costs, becomes more attainable. With higher ROI and lower TCO, an organization becomes more adaptable to market changes. This is because with them, we will maintain a more transparent and understandable application portfolio with a high level of reuse, and at the same time, reserve more money for creating new and best-of-breed products in the areas of our business expansion.

How can these strategic business benefits be achieved from a components' development standpoint? We have already mentioned that to make our components more interoperable, we should reduce the level of disparity, but at what level? By building components on the same platforms using the same languages, versions, protocols, and so on? This will be unrealistic even within a single department, not to mention a decent-sized enterprise. Our development and implementation processes must be focused on reducing the integration efforts between components, aiming to standardize interfaces. Taking this standardization further onto higher levels, we could achieve a certain abstraction level that will be comprehensible to business analysts yet present enough technical details to be sufficient for IT personnel. The long time benefits promised (which may not be entirely directed) by **object-oriented programming** (**OOP**) are quite rarely achieved due to the complexity of inheritance and encapsulation concepts.

The **Agile** developing approach is the solid basis on which business analysts and technical leads can find mutual areas for fostering reusable components with minimal interaction cycles. Still, the Agile methodology in place is not the main prerequisite for achieving this, and if correctly maintained, the level of interface abstraction allows people from both business and technology fields to speak the same language. The main outcome of this exercise should be to provide a description of a component's interface with business-related capabilities that is desirable for the expected level of reuse. What is inside the component, that is, its technical implementation, is completely out of discussion, and it's up to the technical lead to decide which way to go.

Thus, various technical platforms can leverage their best sides where it's needed (or where it's inevitable due to specific skill sets in place of physical implementation) by staying interconnected without affecting each other's premises and project deadlines. Finally, the federated approach gives the opportunity to choose the best products from various vendors and assemble them in the business flows, abstracted and architected in the previous steps. Of course, these products must stay in compliance with the interface specifications and the operational requirements that we put in place. The opposite is also true, that is, setting standards from our business standpoint will help vendors to adjust their products and offerings in such a way that integration efforts will be minimal.

So, it's all about money, as the logical sequence mentioned earlier demonstrates. Have you noticed that in that logical exercise, we didn't use the abbreviation SOA at all? So far, we are just trying to convert the previously presented list of intentions derived from various project-design documents, such as **request for informations** (**RFIs**) and **request for proposals** (**RFPs**), into a concise list of benefits. Our next step will be to assess how attainable they are. Although that will be the purpose of the entire book, the key criterion will be defined here shortly. Before proceeding with this, we would like to stress again that the basic terminology around business benefits and design characteristics is based on the widely accepted structure presented by Thomas Erl as mentioned earlier. Also, we do not want to reinvent the wheel for the thousandth time and then participate in terminology wars, which will lead us nowhere. Thomas Erl has described the obvious benefits that we would like to achieve in a logical sequence, and you can see the proposed sequence for implementing the listed-out goals in the following table:

Goals and benefits	**Common solutions' requirements**
Increased ROI	1, 8
Reduced IT burden (low TCO)	1, 3
Increased organizational agility (shorter time to market)	4, 8, 10
Increased intrinsic interoperability (reduce integration efforts)	2, 4, 6
Increased vendor diversification options	3
Increased federation	5, 6
Increased business and technology alignment	1, 2

It is also obvious from the previous requirements list that some of the requirements are very contradictory, and in most practical implementations, there could be quite a few natural enemies present, such as:

- Security and performance (always blood enemies).
- Reliability factors and highly reusable components (for example, having a single point of failure).
- Resilience achieved by Redundant Implementation and IT costs, independent reusable assets and governance costs (for example, preventing the component logic from getting scattered over several implementations).
- Flexibility and reuse-by-design and development costs (for example, in the initial phase of development). Higher flexibility denotes that more execution paths are required, which requires more testing.

This list can go on as the previously mentioned points are just the obvious ones. Thus, the benefits summarized in the table are comparably more consistent as the most contradicting parts are abstracted. However, we must keep in mind that they are still there, and we will focus on them in more detail while discussing the implementation of design principles. What is important now is to distill the most common characteristics that any architectural approach will ensure in every application to attain these benefits.

It is clear that one of the primary requirements for reducing time to market is to improve communication between the technical leads and business analysts. If the ways of expressing the business and technical requirements are kept abstracted from the analysts, and at the same time, the essential technical specifications are kept in place for the developers, then this architectural model could be truly business-driven. The other way around is also valid. If IT provides a managed collection of reusable business-related services, then it's quite possible that new business opportunities can be spotted and proposed by business analysts; this is because new workflows are composed out of the existing services. The response time to the new challenges will be lowered as the change in the implementation's task force will be business-driven and IT will be resource-oriented at the same time.

Components, especially developed with a business recomposition option in mind, will gradually form some kind of components library. With strong sponsorship from architects, this library will become attractive for more and more extensive reuse in various business domains, depending on the business context of the components of course. This library has a name. Traditionally, it's called **repository**, and we will spend a lot of time discussing its purpose and architecture a bit further. However, from the characteristics standpoint, let's depict it as a technical platform that is capable of hosting these components and providing runtime and design-time visibility, which will be discussed further. Simplistically, this will be any application server with a management console, available for all enterprise developers and architects; it will present all reusable components as the sole **enterprise-centric assets**.

This second characteristic would be possible only when the presented components are designed with the highest level of composability in mind. This means that when integration efforts, including regression test requirements, platform performance enforcements, and activity monitoring are tamed enough to a level where the reusability option becomes so attractive for all the technical and business teams, the idea of reinventing the wheel would never come as a plausible option. Surely, these characteristics could have more governance efforts in the background than purely technical ones. Still, with proper planning based on honest and realistic maturity assessment and with evasion of the big bang's "all-or-nothing" approach, when SOA becomes more religion than the practical "one step at the time" approach, it’s quite achievable.

Components developed as reusable assets should follow commonly accepted standards; otherwise, reusability will be severely limited to one technology domain. Another alternative would be to reinvent the already existing standards, which is always a waste of time. It doesn't mean that any published standard must be followed blindly; the adoption of standards must be carefully planned. An enterprise's maturity analysis combined with marketing research on top products in a particular area will guide an architect towards common models, describing the component's behavior and implementation technique with minimal integration efforts. Thus, by achieving the first three characteristics, we will open the highly desirable option of maintaining the hot-pluggable infrastructure where best-of-breed products from various vendors could be combined into well-turned fabric based on common standards. It is an architect's responsibility to stay watchful, analyze standards' specifications, and deduct the crucial parts and to be focused on increasing the desirable characteristics.

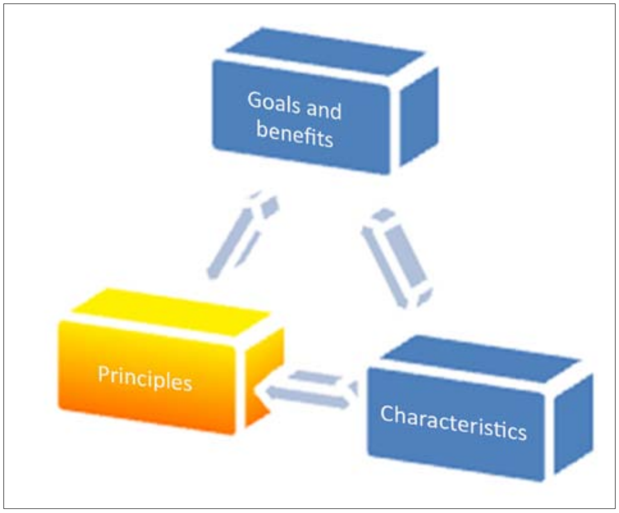
This design characteristic of making it possible for all components in a repository to stay vendor-neutral has an extra significance for us in the context of this book; it is dedicated to the realization of certain design patterns on the Oracle platform. Actually, there is no contradiction here. This is because we will strive to present concrete solutions in a vendor-neutral way first, if possible, and then demonstrate how Oracle tools could address the same issue. We will do this here and try to demonstrate the maturity of the Oracle platform, which is capable of delivering hot-pluggable solutions that could potentially pose fewer burdens for the enterprise IT domain.

So, now it's a good time to sum up the short descriptions given previously in the table of architectural characteristics, in the way they have been defined by Thomas Erl. Here, we will again repeat our exercise, trying to map the supporting core characteristics to most of our requirements (if not all).

Characteristics	Requirements [R 1] to [R 8]
Business-driven	2
Composition-centric	1, 5, 8
Enterprise-centric	4
Vendor-neutral	7

We have selected the most obvious characteristics, which directly support the most common requirements, summarized at the beginning of this chapter. There is no need to elaborate on this further, as we can easily see that other requirements are supported directly or indirectly. However, we would strongly recommend that you repeat this exercise every time you analyze the requirements for new products, systems, or components in the RFI scope or at any other stage of the project.

The following figure summarizes all that we have learned so far:



Until now, we have intentionally avoided mentioning the term SOA during this short exercise of outlining the keystones of the requirements analysis. The purpose is quite straightforward: if you could clearly define your goals in a very precise way and declare concise characteristics in support of these objectives, it really doesn't matter what the name of your design approach is. Some can call it common sense, and that's perfectly fine. Nothing could be better than a design approach based on common sense, which is easily comprehendible by business and IT. However, apparently something else is needed, and that would be the design principles as a strong foundation for the first two keystones.

In the following paragraphs, we will outline this foundation again using the classification provided by a best-selling SOA author and founder of the SOA School, Thomas Erl, which is accepted by Oracle. This time, we will strongly focus on SOA and **Service-Oriented Computing** (**SOC**) as the best technical implementation of common sense depicted earlier. There is no reason to evade it further—if it looks like a duck, swims like a duck, and quacks like a duck, then it is probably a duck. Goals and characteristics mentioned previously are exactly how the SOA declares them. How they will be achieved and supported is a matter of the principles' implementation. They are all interconnected, so the balance is also part of common sense, and as is usually put, it must be applied in a meaningful context. The extent of this is the level of realization of tactical and strategic goals and technical capabilities of the principles' implementation.

An example of architecting for tactical goals

> Please be forewarned as the following example is a recipe for a perfect disaster. We have to put this disclaimer as some could take it as direct architectural advice. It is also sadly realistic, since all that we have described next was taken from real implementations. We will use this example in the later chapters.

So, what are the tactical goals? The essence here is time, usually limited by a timeframe of a project or several milestones of non-correlated projects. It is always good to stay on budget and deliver on time what was promised. This is a common scenario for a component or a single application development process. Isolation, focus on performance, and reliability as primary targets have their obvious benefits. As a solution architect, do not bother your team much with interoperability, as you probably have another **enterprise application integration** (**EAI**) team that is especially dedicated to this purpose; they are somewhere nearby and are capable of performing the tricks. Skillful EAI means that some integration platforms are in place already, providing hub-and-spoke capabilities with all the necessary transformations, translations, protocol bridges, and so on. Honestly, nothing's wrong with that. At least, not yet.

All that you need is a capable integration team and be lucky enough not to be at the end in the row of endless regression tests. Also, it would be prudent to maintain a very thorough events/error log for your product, just in case you need to identify where all your inbound/outbound messages have gone. You must be able to prove that you (your application) have sent all the required outgoing messages, and they are definitely now on the integration platform's side (just search better); alternatively, if you haven't received what you need, the flaw is definitely on the part of the EAI's design.

As time is of the essence, moving further, you can take the liberty to define all your APIs and XSDs as close to your technical implementations as possible, based on the DB structure and the logic of the classes. Modern development platforms and SDK/XDK are truly advanced, so this task can really be done in no time by a right-mouse click. Following this path, you can provide newer versions of your application almost instantly after receiving new requirements, and it's purely EAI's responsibility to maintain concurrent APIs published on an integration platform. Again, just be the first in the list of regression tests.

As performance is declared to be one of the primary objectives, always demand for direct access to the resources you need. A direct DB access or **Remote Method Invocation** (**RMI**) is much faster than the hub-and-spoke integration approach. At the same time, do not let anyone access your internal resources, as it potentially could affect the third characteristic declared prime, that is, reliability. On second thoughts, it would be good to hide all your implementation details and keep them hidden from everyone. It will prevent any unauthorized access into your backend resources.

Strengthening security is a positive side effect of this isolation. The obvious fact that all technical details and data structures are already exposed via your autogenerated **Web Services Description Languages** (**WSDLs**) are just **Web Services** (**WS**) collaterals and must be handled again by EAI. This is because we follow a common principle of **separation of concerns** by delegating security operations to the middleware. By helping middleware handle error situations, you could provide a full-stack trace from your entire web-based API, leaving the standard **SOAP Fault** message by default. The EAI team will have to find all the necessary details and handle them according to their understanding of business logic, as we will keep our internal logic secluded for the reasons explained previously (security and resource protection).

As you already have direct connections to the resources in other systems, you could potentially implement some extra logic on your side in order to speed up the external processes and get the necessary results without waiting. Why not? We are endorsing distributed computing! You can go even further; you can include your public API's capabilities from other systems, as you already have access to them. So, it's a mashup, isn't it? For the sake of clarity, you just inform the EAI team that these new capabilities are foreign and not covered by your original SLA, as you cannot guarantee that the design of other systems would be good; however, you welcome everyone to use them. Speaking of SLA, quite soon you will spot that the autogenerated XSDs are a bit elaborate, causing some latency on the API side and extra processing overhead on the EAI platform. As an architect, you would propose quite a simple workaround (remember that performance is the essence): switch off the XSD validation at the EAI platform's end. It certainly helped a bit, but not enough. Later, you will discover the original cause, that is, the standard JAXB library responsible for message marshaling/demarshaling is way too slow.

The implementation of custom marshaling would certainly be helpful not only for your application, but also for others' as well. Why not help other teams by supplying them with a more robust and reliable XDK? In parallel, you can make some improvements to the XML structure, presenting custom elements within the message body for parsing acceleration. For instance, if you have several addresses in the message (billing, postal, and corporate), you could implement special predicates to indicate which one is to be used in a particular business case and when others should be suppressed. You can really dedicate some time to these tasks so that developing and adapting your components is not so burdensome.

Our tactical goals have been well achieved. All that's left is to explain to your CIO why the consolidated IT costs after three years of tactical architecting are almost equal to corporate revenues.

If you think the presented scenario is a bit artificial, please suspect not. On the contrary, some unnecessary technical details had been omitted to make it less chilling. However, we would like to make one thing clear: we are not against tactical goals at all; they are chunks of iterative development and essential parts of SCRUM sprints. We just believe that tactical goals and benefits must be a native part of some bigger strategy; otherwise, you could win a battle or two but lose the war very badly. The temptation to achieve your target instantly by buying another magic pill is always high but usually leads to a spaghetti-like infrastructure. In best case scenarios, you will get a lasagna-style infrastructure if your integration efforts are consistent (another term for expensive). So now, we are going to discuss the principles that could make our strategy capable of supporting declared goals and characteristics.

SOA principles

Don't worry, we will not be reinventing the terms here again. After more than ten years of implementation, the principles are quite well declared and explained. The consolidation done by Thomas Erl has been accepted de facto by most of the top market players, and what is most important for this exercise is that it has been accepted even by Oracle. You can refer to it at `http://serviceorientation.com/serviceorientation/index`.

Here, we will mostly focus on the relation of the principles and characteristics and the consequences that will follow if the principles are neglected. Jumping ahead, it would be right to say that the patterns are really needed when principles are not implemented as they are intended. The reasons for this could be different, which are mentioned as follows:

- The already existing burden of legacy systems prevents us from implementing more reusable solutions immediately. We really do not want any revolutions.
- The obvious political reasons of all kinds, usually caused by strong focus on tactical goals, temptation to pick the low-hanging fruit, and show quick results even if they are based on another silo in the app's stack.
- Most interestingly, patterns would be required to resolve the conflicts that arise during the implementation of different principles from the same technological area. Yes, principles can contradict and must be applied in a meaningful context.

It is also important to recognize that all these architectural principles are generic, common, and universal for the selected technological area (SOA in this case). Principles are also tangible, well recognized, and limited in number.

Some may say that our top-ten requirements can be perceived as principles as well. Even all illities could be principles because of their universality and simplicity. Unfortunately, simplicity here cannot help. We, as architects, should give strict, precise, and most importantly, tangible guidance to developers, and be able to follow our own recommendations.

The measurable outcome is the result of proper guidance, and the principles here are closest to the physical implementation and must be understood and followed. Some principles could be less tangible than others and could just present the results for collective implementation principles with lesser abstraction, but still the results of the implementations can be measured. Let's take the most common illities such as **reliability** or **flexibility** and try to explain to your developers (or yourself) in just few words how to code your components in order to achieve them.

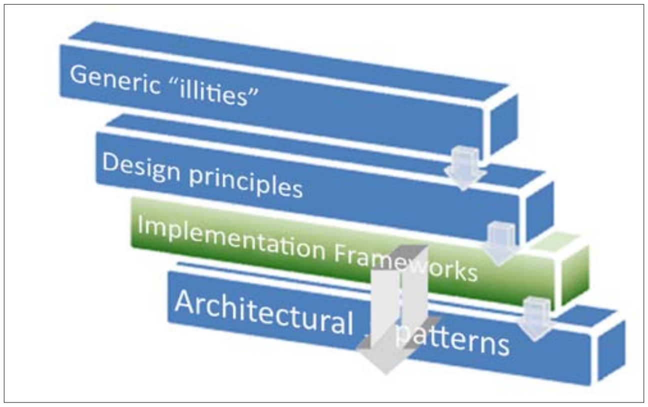
> Illities are also known as **non-functional requirements** (**NFRs**).

Depending on the technology platform, the explanation could take up to a couple of pages or several chapters. Still, they should seem obvious and even quite measurable. (Reliability is usually the **Mean Time Between Failure** (**MTBF**) and flexibility is also a time-based characteristic that displays how fast a system can be reconfigured for other business requirements.) So, NFRs are also precise technical requirements and not a guide in technical terms. Looking forward, let's propose a logical hierarchy of the terms, one way or another related to principles and their areas of application. By the end of this chapter, we will cover all of them. Your benefit from this exercise will be a clear outline that will guide you on how to analyze requirements and apply design rules for most of your SOA-related projects. The following table illustrates the principles and patterns discussed:

Principles and patterns	Quantity
Even being highly generic, the characteristics of generic illities have certain practical implications and materialize in at least six architectural frameworks.	7

Principles and patterns	Quantity
Technology stack's architectural principles (for SOA design principles) states that every application consists of several technology areas, the sharing or reuse of components, and composites. For every application, an individual and balanced combination of the universal principles is the key for successful implementation.	8
Architectural patterns form a pattern catalog, commonly approved as open standard (`.org`). It is the number of concrete patterns that are recognized.	>85

The following figure explains the preceding principles and patterns:



We will discuss frameworks separately after a quick walkthrough of the principles.

So let's start with the obvious ones that were already mentioned earlier.

Standardized service contract

In a standardized service contract, we really believe that the word service here is a bit of an overkill. Services today are strongly associated with the web service's technical implementations, so naturally, the first thing that comes to our mind would be WSDL with schemas, optionally, with policies. Nothing's wrong with that; it's truly the most common service implementation (or REST maybe), but the fact that any WSDL and XSD can be easily autogenerated compromises this idea. Autogeneration doesn't turn it into a standard. An autogenerated service contract is nothing but trouble, and if you haven't got it after the first exercise dedicated to tactical goals, we will have plenty of opportunities to convince you.

By doing so, you are just forcing everyone to adapt to your specifications, which include existing applications. Surely, if you are an architect in a big bank, this approach might work; we're hesitant about the stub's implementation time at the remote end.

The second point here is that we can build a very robust service-oriented system without a single web service. The Oracle PL/SQL will do it beautifully, and we will demonstrate it in the next chapters. So, the contract could be anything that declares public operations, available protocols, and data structures such as the PL/SQL package header `.pks`, C++ class header `.h`, and the Java interfaces (also called contracts). Component-based development is a completely valid approach in SOA if it's approved by all the members of the implementation domain. Interactions between different domains will require some integration efforts even if the technology is the same, but that would be true for web services as well, so that's not a major drawback of using components as SOA's building blocks.

The problem here is that all components' contracts are nondetachable compared to WSDLs. The true beauty of WS interfaces is that we can sit in a quiet corner along with developers and business analysts and by using just a pen and a napkin describe the prototype of service compositions, go back to our stations, and start coding right away. Talking seriously, by describing the detachable contract as WSDL, we can really provide a parallel development process and work on an iterative development in a reasonably painless manner. Simply speaking, you can compile the service logic (Java) without WSDL and try to do the same with a PL/SQL package body without the package specification. Finally, the most important thing is that this contract-first approach allows us to generate the code based on an initially defined and mutually accepted contract, and Oracle is really good at it. Practically, you can generate skeletons on any platform that you want to be your logic carrier, such as Java, BPEL, and Mediators.

A standard contract is the primary means of presenting your service as a corporate asset to maintain at least two main SOA characteristics: Composition-centric and Enterprise-centric. With the WS-based approach, you will achieve vendor neutrality as well.

Loose Coupling

This principle is probably the most well-comprehended principle. Everybody knows that tight coupling is bad. Is it really? To discuss this, let's first describe what kind of couplings we could get. You'll be able to understand this from the realization of service anatomy. Basically, we have the following:

- Service resources, presented as DB, file structures, and so on
- Technology platform (Java, .NET)

- Service logic implementation
- Parent service logic

Anything that links your contract or, even worse, your consumer to one of those service resources can obstruct the core SOA characteristics we are trying to maintain. So, all links going from contract to service resources or bypassing the contract are bad. The opposite direction is not much better because providing details of the technology platform or excessive resource demonstrations is not good, as it can provoke the service consumers to build their consumption logic based on these details. However, what about the contract-first principle? Yes, it's a positive thing, so coupling your service logic to the declared contract is a natural and decent way of implementing the service. However, neither the service logic nor the service contract has been set in stone—business is evolving, and so are our services. Quite soon, a new contract version will be published or the core service logic will be patched. It will eventually turn out that this positive coupling also has its deficiencies. No reason to despair though; it's life. All of us are evolving, and customers connected to our contract are always welcome. How to deal with this situation using various Oracle SOA patterns will be discussed further.

In addition to this, we would like to emphasize that coupling from customer to contract is the second positive coupling, although it is susceptible to the same problem like the one with contract evolution. All other couplings must be prohibited if possible. This statement is not as strong as you would expect. We have touched upon the reason for this earlier in the tactical goal's architecture example, that is, performance. Standard contract denotes the message processing overhead, some milliseconds (or more) in addition to the total processing time, CPU utilization, and memory consumption. Is it worthy enough to jump over the service contract and utilize service resources directly? Only you know what these milliseconds of overhead mean for your business, and the decision on what to sacrifice is yours. In general, the answer is no. Please look at your contract first. Is it truly standard? Assess your needs using the following logic:

- Do you clearly define your data structures with the required elements only?
- Do you avoid autogeneration, especially for operations with CLOB fields without `CDATA` or `<any>` elements? (Memory leaks during marshaling is a common outcome of this approach.)
- Can a concurrent contract with more lightweight technology (REST instead of SOAP) possibly solve the problem? (Concurrent contracts will be discussed further in the *Chapter 4*, *From Traditional Integration to Composition – Enterprise Business Services*.)
- How about a platform-specific SOAP/XML acceleration? Oracle's WLS T3 protocol could be useful as it has proven many times

- The tuning of the execution environment and proactive monitoring.
- If platform-neutral contracts (WSDL / REST-based) do not help, could we employ a component-based concurrent contract?

Always think what price you will pay to break this principle for gaining ten or fifty milliseconds of processing time. This principle directly supports the composition's centricity and vendor neutrality's SOA characteristics.

Service abstraction

The logical outcome from the implementation of the first two principles is standard, preferably (but not mandatory) a detachable service contract as a declaration of our capabilities, processing requirements, and input expectations. Still, the word standard is a bit vague. Let's put the discussion about existing standards aside for a moment and focus on the areas of standardization. The bottom line is that standardization is the way of generalizing information, a process of making it more abstract in order for it to be more multipurpose in predefined technical boundaries. Some of the elements of abstraction in service-orientation boundaries that we have already mentioned during the discussion of Loose Coupling are as follows:

- Do not reveal in your service contract the specifications of your technical platform (such as the coding language, SDK's details, and XDK properties)
- Do not expose details regarding your underlying resources (such as the DB structure, constraints, and especially the foreign keys)
- Be reasonably reserved regarding services-composition members that comprises your service

Why would you do that? It is because of the same reasons we mentioned while discussing the previous principle. Excessive information can provoke negative coupling to service resources, making the service less adaptive and reducing its reusability options.

For example, you have a lot of useful functions in your service logic. Obviously, you can fall into the trap of promising extra features in addition to the already agreed one. (Okay, not you, your new project manager.) It literally costs almost nothing at the beginning. Most probably, it will not even affect the level of standardization of your contract at first glance, which is shown as follows:

- Your data model that is based on your corporate-approved entity's **Canonical Data Model** (**CDM**)
- Your naming standards are very clean and comprehensible, based on industry standards

Who can give you a warrant that the business logic, encapsulated in your service, will not change tomorrow and that an auxiliary-declared operation becomes a burden or even an unwanted shortcut in the business process? How about a number of consumers who become dependent on your extra feature? Migration in SOA is not an easy task, even with certain SOA patterns applied.

On the other hand, even in a relatively static business ecosystem, this new feature could become so popular that all of the hardware power dedicated to your service scope will be consumed by only this one operation.

Level of abstraction – granularity and models

So, do not promise anything and keep everything for ourselves? Let's not blow this out of proportion. SOA is full of promises; it was designed in this way, and luckily, we have enough methods to keep these promises. If service capabilities (that is, operations) are correctly planned from the beginning and used unevenly, then maybe we have put too much on a single service's plate. What is the functional scope of this service? If this service handles one single business entity (such as invoice), then all our operations should be bound to its functional context, which is abstracted to the level of a functionally completed environment. You would hardly keep salt, sugar, and flour in one jar in your kitchen just because all of them are white. Still, it's rather amazing how this simple thing called granularity is neglected in the real life of service development.

Functional granularity is based on the understanding of service models. Entity services that are already mentioned are the first and closest abstracts to the atomic data representations in an enterprise, for example, invoice, order, cargo unit, and customer. All operations would be naturally based on the DB CRUD model but not limited by them. The number of truly unique entity services is rarely more than 20 in any enterprise. The functional granularity here is usually based on the OLAP/OLTP segregation:

- **Online transaction processing** (**OLTP**) as very short, real-time, CRUD-like operations with high demands for response time are naturally the primary capabilities of the entity services, and their operational time slot is frequently within the standard business hours (that is, 08:00-17:00)
- **Online analytical processing** (**OLAP**) operations are not that demanding when it comes to response times, but data volumes are usually higher and operational time slots are either evenly distributed around the clock or tend to be close to the regular nightly batch-operations time.

As you can see, mixing them together would not be a good idea if we have an overlapping operational time slot. The possible conflict between high volumes and high throughput will require your attention at the very early stages of service modeling. Should we abstract OLAP operations to DWH-specific services?

The second service model is the utility that usually presents the most reusable and supplemental logic, consumed by all other services. The level of functional abstraction here is really high and business-independent. Your transformation, translation, or measure-unit conversion services are typical representatives of this model. The level of functional granularity can be easily defined and operations can be tuned for high-usage demands. Migration issues are not that frequent here, so functional abstraction is fairly simple.

The last model or task service is what we usually know as workflow, which is the composition of other services combined together in order to fulfill one single task such as OrderProcurement and BookingRequest.

Distinctive properties of this service model comprise one task, one operation, and one business context. Functional abstraction should not be a big puzzle, but still we can see a lot of misinterpretation caused by the deceptive simplicity of modern development tools, providing neat visualization of service compositions, plus very mature resource adapter frameworks starting from order fulfillment (many thanks to Oracle for providing an extremely detailed **Fusion Order Demo** (**FOD**), available at `http://www.oracle.com/technetwork/developer-tools/jdev/learnmore/fod1111-407812.html`). An enthusiastic developer can soon include Invoice, Booking, and General Ledger flows into one monster. Entity services are commonly neglected, as we do not need them anymore; a DB adapter can provide us with the perfect result in five clicks. Additional interfaces that were constructed while composing this task service can be easily exposed to external consumers. The problem here is that this service is not a task anymore; it's a hybrid with the worst possible functional granularity, combining business-specific and business-agnostic capabilities. A thorough application of the abstraction principle from the very beginning could prevent this problem.

The deceptive simplicity of development can hoodwink developers, who are left alone without architectural guidance. This fact provokes developers to use it everywhere, whether it's appropriate or not. Some industry-specific forums and advisory boards quite often produce rather vague frameworks and business process specifications that are in fact not more than business heat maps. Following them too directly can easily result in such hybrid services with unclear abstraction levels.

Data granularity is the next level of granularity we should take into consideration when applying the abstraction principle. Processing one single order line or a complete bunch of orders received in one message in reality makes a difference; however, in your design of an Order XSD, all that it takes is to set `minOccurs` to greater than 1 for the `Order` node right under the `OrderHeader` element.

The data constraints granularity or constraints granularity in general is the next logical level of granularity. Again, talking about XSD, you could be really restrictive with your data type definitions while determining whether they are necessary by declaring a simple type using XSD patterns, explained as follows:

<table>
<tr><th>Fine-grained</th><th colspan="2">Coarse-grained</th></tr>
<tr><td><xsd:simpleType name="imageType">
   <xsd:restriction base="xsd:string">
      <xsd:pattern value="(.)+\.(gif|jpg|jpeg|bmp)"/>
   </xsd:restriction>
</xsd:simpleType></td><td>xsd:string</td><td>xsd:any</td></tr>
</table>

Here, you want to be sure that the image's filename provided in a message is safe (at least not executable). This could not be achieved by the most popular `xsd:string` data type alone in the service contract. The `xsd:any` element is at the upper level in this hierarchy, which in OOP has equivalent `Object`. All these levels of abstraction have full rights to exist, but you must clearly realize which part of your SOA infrastructure should employ these different levels of granularity. The other means of data granularity already mentioned are `minOccurs`, `maxOccurs`, and `nillable` that are applicable for the elements and `xsd:` attributes. Terms usually used for different levels of detailing are the fine- and coarse-grained granularity, and they are quite self-explanatory. The levels of declared granularity directly impact the location of the service-processing logic. This means that with a more fine-grained XSD, you will put more processing demands on the contract's message processing logic—XDK marshalers (serializers). With a more coarse-grained granularity, you inevitably put the big chunk of message parsing and validation logic into the service's component logic behind the contract. It could also make service difficult to test, as highly abstracted contract will not reflect any changes that are supposed to be presented to the Consumer.

Other abstractions include the abstraction of technical details hidden behind the service contract, and programming language aims to increase the federation of our heterogeneous service infrastructure. Abstracting contract-related parts of SLA, such as quality of service, availability information, and performance metrics also helps to standardize service profiles within a service inventory.

We have deliberately put aside the security considerations related to the abstraction principle until now. By declaring more precise data types, you could reveal technical information necessary for data-oriented attacks. When using the data type casts features, the attacker could trigger the error, revealing internal data structures associated with the element (the point of the attack) and exploit them. An element with the type `<any>` reveals nothing, but at the same time allows it to send any types of data, including the harmful code. With such a high level of abstraction, presenting the service contract with the operation `Process` and data model `Any`, you literally open the door for all kinds of parser-related attacks, memory leaks, and buffer overflows. Possible ways of balancing the granularity and abstraction levels for services that operate on different technology layers will be discussed further in *Chapter 7*, *Gotcha! Implementing Security Layers*.

The ultimate purpose of principles' implementation in SOA is to increase the service composability options as a direct method of increasing ROI. A less abstract service contract where more information is revealed tends to be more attractive for developers as they are more interpretable.

The implementation of this principle directly affects SOA characteristics such as composition centricity and vendor neutrality. This principle directly supports Loose Coupling. Abstraction from excessively expressed technical details will certainly increase the business value of the service (business-driven).

Service reusability

The first three fundamental principles combined together will lead us to the declaration of the first really tangible design principle, that is, **reusability**. One can say that this is the essence of all the SOA principles. Still, we will not crown it above all others, as it cannot be maintained alone without proper foundation of the first three. In the book dedicated to the Java EE enterprise architecture, *Sun Certified Enterprise Architect for Java EE Study Guide (2nd Edition)* by *Mark Cade, Humphrey Sheil, Prentice Hall Publishing*, this principle is not included in the requirements for the component's architecture. The first three include performance, scalability, and reliability, and that's absolutely true. No one needs reusable components that are unreliable and cannot perform as intended.

We just have to realize that these illities here are applied to the service logic, presented by Java components. If it's not reliable, and we would like to put that first, we must not present this logic as a public service. In traditional component architecture, the reusability support is delegated to the integration layer. In SOA, we strive to make services reusable by means of the following:

- Defining the standard contract, exposing canonical data and canonical operations.
- Making internal service logic more universal (another synonym for abstract) and suitable for reutilization by other services. As discussed earlier, only one service model is allowed to be highly specific, that is, task, as a composition of other services, fulfilling the specific operation.
- Preventing negative couplings by promoting the technical contract as only one way of accessing the service logic.

The level of reusability is really easy to assess: just count the number of compositions where this service participates. The implementation of this principle directly promotes composability and the enterprise's centricity. Let's now look at the two pure technical principles that support reusability.

Service autonomy

An service can maintain the required level of reliability (measured by MTBF as time, or percentage as an availability) necessary for consistent reuse only if it can possess and control its own underlying resources. Database, file objects, physical realization of the service logic, and so on should not be shared or delegated to other services. The service should be perceived as an atomic unit of concrete logic, functioning in a dedicated technical environment. In this case, the service behavior will be predictable, fulfilling scalability requirements, and making it possible to relocate the service into a similar technical environment with reasonably low efforts. This last illity is highly desirable for a cloud-based implementation.

Unfortunately, this principle is probably the hardest to implement. We all know that most commonly used databases are shared resources. License costs, bundles of legacy applications, common network infrastructure, and so on are the reasons why this principle is very hard to achieve without significant investment or considerable maintenance efforts.

This principle is quite often mistaken for Loose Coupling. Indeed, they are very similar with regards to the negative impact on service reusability, although we can draw a distinct line between them as mentioned as follows:

- Loose Coupling is the ratio between iterations, carrying through the service contract from consumer and service resources (relatively positive coupling) and iterations bypassing the contract (negative coupling). In fact, the service is always coupled if it's in use, positively and negatively. Service-oriented architecture based on components is more prone to negative coupling, as their APIs are more technology-specific.
- Service autonomy is the measure of service independence. A business usually has quite a limited number of **data warehouses** (**DWHs**) (usually one per business domain and ideally—only one). Therefore, all analytical services (InvoiceHistory, OrderHistory, and so on) using a single DWH DB will not be autonomous. Present them as one service (this is not an advice), move into a private cloud, and you will get a perfectly autonomous service. Now the question is the price.

The implementation of this principle directly promotes the composition and enterprise centricity and vendor neutrality.

There are no negative impacts on other principles, but as we have said, true service autonomy is the nirvana that is really hard to reach.

Service statefulness

This second technical and very tangible design principle is the support of the service reusability.

It's defined as an ability of a service to maintain low-resource consumption when needed, namely between service activities, while waiting for a response, and so on.

At first glance, it's more applicable to the long running asynchronous services, which could run for days or weeks. The deferring service state is vital here. We have to store execution scope variables and preinvocation data in a special database with all necessary information for waking it up when the response arrives. In this Hibernation DB (**dehydration store** in Oracle terms), we will have a chain of defer-awake records that are equal to the number of asynchronous invocations. Moving further, we can defer the information at any stage of the long running process for legal or compensative activities. This type of storage is compulsory for all task-orchestrated services and is usually provided centrally by an orchestration platform.

This fact makes all task orchestrated services far less autonomic than other service models. Surely, you can implement individual partial state deferral for every task-orchestrated service (task service hosted within an orchestration platform), making them ultimately autonomous. In that case, we truly admire the grandiosity of your project's budget, not to mention the infrastructure and support.

Asynchronous services are not the only ones that need to maintain their state. A poorly designed synchronous service with a lot of global variables, excessive looping or branching logic, and a lot of calls to underlying storage resources will consume a lot of memory and CPU. There is no remedy for this scenario that is provided by the SOA technology platform; you can only rebuild it from scratch. You could technically turn this service into an asynchronous process and set the queues as transport means; however, that's not what consumers expect, and the level of potential reuse drops considerably. Only services with predictable state management will have predictable behavior and scale well when necessary.

This principle addresses the same benefits as autonomy but has negative impact on the autonomy itself.

Service discoverability

Despite its obvious meaning, this is arguably the most-neglected principle. The main reason for such negligence is the misunderstanding of service governance boundaries. For some, governance starts after the service deployment process in production. Although it's been said many times before, we would like to repeat it again—governance starts long before the first line of code is written. You must plan for the following:

- Which service trace records will be left in your audit/trace log under the different logging settings
- How service activities will be perceived by different operational policies
- How a service could be dynamically invoked by different consumers and controllers

These items among many others form the so-called runtime discoverability. It is in your best interests to expose your service to all who can potentially use it. This is possible if you follow these points:

- Service operations and functional boundaries are well defined according to the service contract
- Service particulars presented in the form of service metadata help everyone understand possible service runtime roles, model, composability potential, and limitations

- Quality of service information describing service availability, reliability, and performance is guaranteed
- Supplementary information regarding all test results and test conditions are in support of declared performance
- Policy standards to which services adhere to mandatorily or optionally

All these items are elements of design-time discoverability. As part of governance efforts, special layers of service infrastructure will be established in order to support these two types of discoverability. We will dedicate a whole chapter to this challenge, as we reckon lack of discoverability is probably the main problem in the implementation of SOA. However, even if a technical platform is capable of supporting dynamic discovery and invocation, and the service metadata storage is full of service details down to the particular engines in use and rule types employed, we could still jeopardize potential reuse by keeping interpretability of discovered information below the comprehension level. Information that is too abstract (see the *Implementation of Service Abstraction principle*) will prevent the demonstration of full service potentials. A methodology in support of service taxonomy and metadata ontology (discussed in *Chapter 5*, *Maintaining the Core – the Service Repository* in great detail) will be not only established, but also conveyed to all individuals responsible for SOA governance from the very beginning to the end.

How do we measure service discoverability? What questions should you ask your developers and architects (including yourself) in order to understand the level of principle adoption? Refer to the following questions:

- Do we have an inventory for all enterprise assets acting as service consumers and service providers? In other words, what are the enterprise/domain boundaries?
- Do we have individual service profiles?
- What are the key elements of service metadata available from the service profile that we will use for a service lookup?
- From which service infrastructure layers will we perform the lookup and for what purpose? In other words, who is allowed to discover this information and when (security)?
- Can we perform reverse search metadata by service?
- How will these metadata elements be presented in a service message, in which parts, and at what level of detail?
- Are these metadata elements in the service messages covered by the existing SOA standards? Can we keep them vendor-neutral and minimalistic?

Believe it or not, a simple Excel spreadsheet will do the trick, and the explanation will be provided shortly. How many times have we witnessed the situation when without a well-structured and understandable taxonomy, even the most powerful harvesters (SOA artifacts introspection tools) with marvelous graphical representation of discovered relations just turn the situation from bad to worse?

This principle undoubtedly supports all four desirable SOA characteristics. This principle conflicts with the service abstraction and can negatively impact security when poorly implemented.

Service composability

Finally, we come to the last principle that completes the foundation of SOA. This principle is in fact the paramount realization of SOA, as it's the closest thing to money, the universal entity that is understandable by any members of an enterprise.

Next, we compose and recompose the new business applications and processes out of the existing building blocks; the less we waste, the more we gain through reuse. However, is there any overlap between composability and the reusability principles discussed before? Yes, but only at first glance, as the key here is in the measure of "waste" we would like to prevent. Almost everything in IT could be reused; the question is how much effort (time) it could cost for doing this.

The composability principle defines the measure of how easily any service from a particular service inventory could be involved in a new composition, regardless of the composition's size or complexity. Of course, a service should be involved in the operations it was designed for and in the roles it can support. Thus, the common quantifier for this principle is time. When designing a service from the very beginning, you should speculate on how hard it would be to implement composed capabilities using your service along with others, and how many compositions a single instance of your service can support from the performance standpoint. Yes, the statement *regardless of the composition's size or complexity* has its own limits, and these limits will be thoroughly tested during the unit stress test and properly documented in the individual service profile.

The quantification of this principle is roughly similar to the measure of flexibility of hub-and-spoke integration platform. For hub-and-spoke, with all enterprise applications connected to it, you have the canonical data models for all entities presented by the application. When a new application with its own data representation arrives, all that you need is to perform the following:

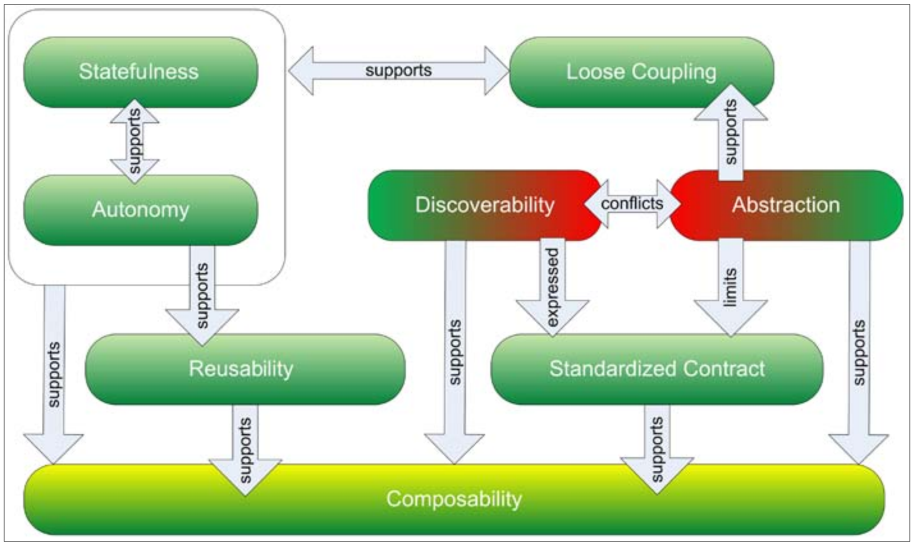
- Transform the newly received application data model to a canonical model
- Establish routing rules for message flows in a hub for this application

For simplicity, we assume that a unified canonical protocol is in place. With this simplistic model, we can estimate that a new XSLT (actually, two of them) with a reasonable level of complexity could be built in four hours, and the routing rules in a static table can be maintained in another two. Allocate a day for testing and we are ready for production in less than 16 business hours! Theoretically. Let's now recall how reality typically bends our plans.

No.	**How do we see it**	**What it means**
1	Yes, it's not a problem to build XSLT or XQuery in four hours. Obtaining the mapping instruction and understanding the meaning of field names and data types can take week(s).	Discoverability and Abstraction
2	Not all data elements needed for CDM and other applications in the farm are available via a public API.	Standardized contract
3	To extract necessary data, we have to bypass the API and reach out for internal resources.	Loose Coupling
4	The extraction of additional data inevitably implements call-back interchange patterns, which are not always positive. This could disrupt performance of the main app functions and put some logic outside the app boundaries.	Autonomy
5	Quite often, internal logic of a new application is more complex than what a simple request-response interchange pattern can provide, thus requiring the hub **atomic transaction coordinator's** (**ATCs**) capabilities. We have to put more logic on the hub.	Loose Coupling

This shortlist of five items is only the tip of the iceberg; usually, it's more than a page long. The bottom line is that the total time required for making a new application composable via the hub-and-spoke approach is usually from two to six months.

With the service-oriented approach, the result will be quite similar if any of the previously discussed principles are neglected or put off-balance. The following figure explains the general relations between principles and their importance for composability. **Loose Coupling** and **Abstraction** together with **Composability** are the regulatory principles, shaping and governing the implementation of others. Despite their regulatory status, only **Abstraction** is quite difficult to quantify, although it's still quite possible by assessing the amount of message-processing logic in marshaler / contract parser and core service logic.



The **Statefulness** and **Autonomy** services are pure technical principles and directly affect **Reusability**. It would be quite right to say that **Reusability** and **Discoverability** together have a major impact on **Composability**, but other principles also must be accounted for.

We would like to advise you to keep this relation matrix in front of you every time you are given the task to analyze an existing design or propose a new one based on expressed illities, or analyze what's behind the illities, which is promoted as design guidance. It will prevent you from establishing rules that are too vague for understanding and following. For the final exercise dedicated to principles alone, let's get back to our list of top ten generic wishes and analyze the sixth item; refer to the following table:

Designed for testing	Meaning	Requirement
Valid (we are testing what we are supposed to test)	Service / Component APIs can be easily exposed to any existing testing tool (JMeter, SoapUI, LoadUI, and so on) and all the test operations generated with low efforts.	Standardized Contract
Verifiable (we must be able to recreate the results)	Test results, achieved in one environment (JIT, for instance) can be easily recreated in any other environment in testing hierarchy. This means no surprises in production!	Autonomy, Statefulness

Designed for testing	Meaning	Requirement
Reliable (we will trust test results)	Service with acceptable characteristics achieved initially must not be obscured by later amendments/implementations breaking services, technical consistency.	Autonomy, Statefulness, Loose Coupling
Comprehendible (we must be able to understand the results)	Test results, sometime together with test suites must be stored in a place where it can be reached, accessed, comprehended by any concerned party and re-implemented if necessary.	Discoverability (Beware of Abstraction)

This simple exercise demonstrates that there's neither any need to invent new principles out of wishes or abstract illities, nor to multiply unnecessary entities:

Entia non sunt multiplicanda praeter necessitatem

– Ockham

Principles here act as precise technical guidance. Elaborative lists of more than 50 items, produced as a collective effort of several departments in some enterprises, usually leave only one question: how are you supposed to enforce all of them? A principle is the direct order, and it is only good if you can control its fulfillment.

For good measure, may we suggest that you repeat this exercise for the remaining nine requirements in that list?

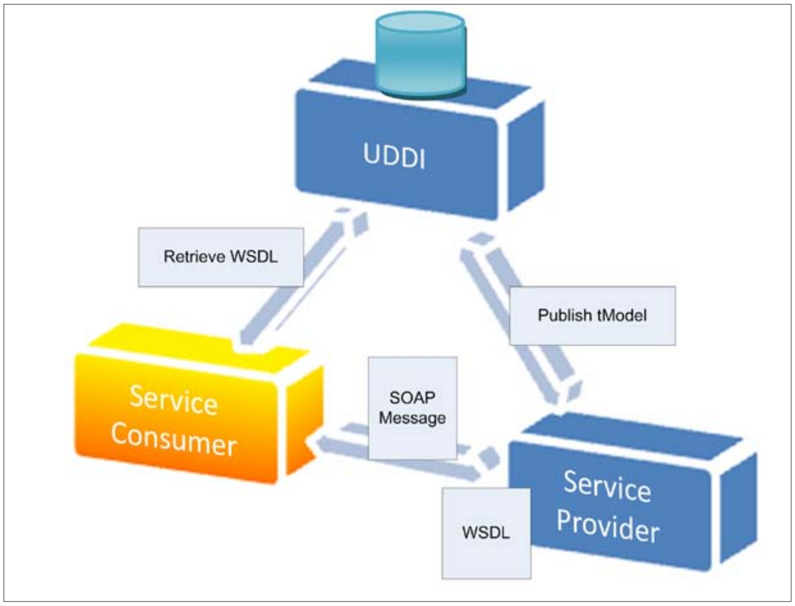
SOA technology concept

The deceptive simplicity of SOA as an architectural approach made it attractive twelve years ago, and this deception (actually, the misunderstanding) caused its downfall after two to three years of initial implementations. Initially, the idea was pure and simply brilliant; it is mentioned as follows:

- Quickly maturing XML, as the most universal standard and foundation for practically everything: messaging, transformations, protocols, data representation, the way you name it

- Emergence of web services, as the next logical step in object-orientation and procedural programming with the highest level of encapsulation and standard detachable API expressed via the WSDL contract
- SOAP presented in 1998 promised some transport-independent (to a certain extent) messaging protocol that was simple enough to gain popularity in the blink of an eye

In addition to the UDDI standard, employing lots of XML features has also been presented in order to support service discoverability. Thus, we got our first so-called Contemporary SOA representation in the shape of a triangle: Service Consumer (Sender), Service Provider (Receiver), and Service Registry (UDDI) shown as follows:



Message is always SOAP-based synchronous request/response; contract is WSDL with number of operations abstracted to be reasonable minimum. **Service Consumer** is displayed in a different color in order to stress the fact that it doesn't have to be a web service. This means that a service can be called from any program, interface, and device if WSDL can be understood and SOAP request-response can be supported.

This model will work (and more importantly it works) perfectly within designated boundaries for simple compositions or compositions based on multiple sequential invocations. We will discuss the obvious limitation of the contemporary model shortly; now, let's focus on core technologies, which are the core of all SOA models.

As stated before, we do not intend to cover all SOA standards, as we simply have no room for this in a single chapter. We will briefly touch only those which we will be using in the following chapters. The book *Web Service Contract Design and Versioning for SOA, Prentice Hall Publishing*, by Thomas Erl, could be a good reference in addition to the web recourses from standard authorities for SAML, OAuth, and so on. This section is just a technical recap of the absolute bare minimum requirements to link previous paragraphs dedicated to SOA principles with the soon-to-arrive SOA design rules based on patterns applied in particular SOA frameworks.

As an experienced architect, you are certainly quite familiar with all the technologies we will briefly touch on now.

XML

This is the foundation of most of the standards and applications, not only SOA. Please note that you do *not* have to use XML to make your platform service-oriented in order to achieve goals of service-orientation, but you will find it rather difficult not to do so. You will have to replace quite a sizeable amount of movable and static elements of your infrastructure (configuration files, interchange messages, transformation mappings, contracts, and so on) with some similar but older formats such as CSV, EDIFACT, and X12 with lots of unexpected consequences. Modern standards such as JSON are also not entirely XML-free. So, we would like to suggest something for your own architectural benefits. Please look at the simple W3C School XML quiz (`http://www.w3schools.com/xml/xml_quiz.asp`); it will only take five minutes. If your score is less than 100 percent, we suggest you refresh yourself by reading a good XML book.

Web Services (WS)

If service is an atomic building block of the whole SOA, then web services are the most popular variant of these building blocks. The reason for this is in an object/XML serializer, which is the native part of any WS and the link between a detached WSDL-based service contract and core service logic. For the Oracle platform (but not only), quite naturally employed Java marshaling/unmarshaling, Java-WS (or JWS) technology would be based on one of the following serialization APIs:

- Java architecture for XML binding: JAXB (exists on multiple implementations but is not always fully compatible).
- A more advanced JiBX. This can inject the conversion code directly into Java classes during the post-compilation process, and by doing so, improve the performance considerably when compared to JAXB. Also, it has its own runtime-binding component.
- Simplified mapping-free version of marshaler: XStream.

JAXB is still the most popular one because of the number of characteristics it offers:

- Runtime message validation
- XPath-oriented
- No post compilation required for code injection
- Can support very complex message structures

The JAXB API is part of Java SE and EE package bundles, but still it is better to check for the latest release if performance is an issue. If serialization performance is the major concern, look at the JiBX more closely as it could be up to five times (some claim more) faster.

Here again, the reliability and predictability of parser should be balanced with reasonable performance; otherwise, you will have to rebuild your services from scratch every time the XSD specifications change.

So, in the JWS specification, JAXB is responsible for mapping a Java class to the message's XSD using customizable annotations. **Java API for XML Web Services** (**JAX-WS**) is the technology that is responsible for mapping Java parameters to the WSDL declaration. These two specifications conclude **service endpoint interface** (**SEI**) as the representation of a standardized contract. Of course, just using them alone doesn't guarantee that contract will be truly standardized, but they are the two essential technical WS specs. The last and the most important part of the `WS` spec is the web container; it is responsible for performing basic HTTP operations: `POST` and `GET`. It's related to the handling of transport protocols, and we will discuss it right away.

WS transports

XML-based SOAP-messaging protocol handled by the web container is typically implemented as a servlet if you need to utilize the HTTP transport protocol, which is most common for JWS. SOAP is a type of XML structure, serving as a container for service message interactions. There are two mandatory parts: `soap:Envelope` as a root and `soap:Body` that acts as a business payload container. Two optional elements can also be present: `soap:Header` and `soap:Fault`. Although the header element is optional, its role for providing transport and processing-related metadata is enormous. Most of the `WS-*` extensions followed after the first publication of initial WS specs are related to the SOAP headers. There are some which could be equally distributed between the header and body. For instance, `WS-Security` naturally relates to the body and header via encrypted and signed elements and `WS-MetadataExchange` provides and distributes WSDL-related data necessary for establishing service interactions, which can also be done via the SOAP body.

Most commonly, the Java web container will probably use a servlet as the front controller, and it is responsible for parsing the header elements and invoking the JAXB mapping to Java objects to process the SOAP body.

We will discuss UDDI-related protocols as part of contemporary SOA further when we approach the Service Registry architecture.

Need for the WS-* extensions

There are no drawbacks in the contemporary SOA model. Every single technical element is mature and proven to be reliable after dozen of years of evolution and improvements. Actually, it was pretty acceptable from the very beginning, but its broad usage was severely limited by the initial constraints set by the simplified service interaction model: synchronous request-response between limited numbers of composition members (usually two). At the time of the first implementation, it was apparent that a substantial number of complex real-life requirements needed be addressed by the SOA technology platform to make it capable of fulfilling its promises; they are as follows:

- There is more than one simple **message exchange pattern** (**MEP**). We can count one-way MEPs, two-ways, callback MEP types along with all possible types of acknowledgements (responses), such as mandatory, only on errors, and so on.
- Asynchronous MEPs are equally popular and must be covered by the technology platform in a common way. The ability to maintain sync-async communication bridges for complex service interactions was the prerequisite for further SOA proliferation.
- Even synchronous service compositions could be far more complex than the basic request-response method with all elements of distributed transactions and two-phase commits requiring a transparent level of transaction coordination.
- Long-running transactions also need common and reliable methods of controlling process execution with a lot of callbacks and numerous activation-deactivation phases. First of all, this involves transparent coordination based on the correlation ID and correlations sets, and the ability to compensate unsuccessful transactions.
- Services must be able to reliably communicate in cases where we have infrastructure breakdowns or slow responses from other parties.
- Service messages must be equipped with information that is sufficient for supporting complex routings and distributions.

- Services and service registries are extremely vulnerable to security breaches due to high exposure to potential consumers (implementation of the Discoverability principle). This issue will be addressed with minimal impact on services intrinsic's interoperability.

These are only the most obvious requirements, which had to be fulfilled in order to make service orientation capable to serve its purpose and achieve the goals we discussed in the beginning of this chapter. It is apparent that all these issues must be addressed in a standard way; otherwise, proprietary implementations will be put across service environments' federation and vendor neutrality.

Most of the standards in the form of recommendations and profiles are provided by three main standardization committees, as shown in the following table:

About	OASIS	W3C	WS-I
URL	`https://www.oasis-open.org/`	`http://www.w3.org/`	`http://www.ws-i.org/`
Established	1993 as SGML Open	1994 by Tim Berners-Lee	2002
Approximate membership	600	About 390	200
Overall goal (as it relates to SOA)	OASIS promotes industry consensus and produces worldwide standards for security, Cloud computing, SOA, web services, Smart Grid, electronic publishing, emergency management, and other areas. OASIS's open standards offer the potential to lower the cost, stimulate innovation, grow global markets, and protect the right of free choice of technology. (From official site)	W3C's primary activity is to develop protocols and guidelines that ensure long-term growth for the Web. W3C's standards define key parts of what makes the World Wide Web work. (From official site)	The Web Services Interoperability Organization (WS-I) is an open industry organization chartered to establish best practices for the web services interoperability. It is for selected groups of web services standards across platforms, operating systems, and programming languages

About	OASIS	W3C	WS-I
Delivered Standards/ Specifications	UDDI, ebXML, SAML, XACML, WS-BPEL, WS-Security	XML, XML Schema, XQuery, XML Encryption, XML Signature XPath, XSLT, WSDL, SOAP, WS-CDL, WS-Addressing, Web Services Architecture WS-Eventing	Basic Profile, Basic Security Profile

SOA standards

In this section, we will group and discuss the service orientation standards and their roles in establishing framework-based infrastructure. Every standard deserves at least a single dedicated chapter, so we advise you to follow the links to the provided standardizations committee technical pages for more details on the latest versions.

Methodology and governance

Standards that define service repository taxonomy and semantics are explained in the following sections.

SOA Repository Artifact Model and Protocol (S-RAMP)

The details on S-RAMP are given in the following table:

Authority	Primarily addresses	Latest release
OASIS	Discoverability	`https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=s-ramp`

The S-RAMP technical specification defines a common data model for SOA repositories as well as an interaction protocol to facilitate the use of common tooling and sharing of data. S-RAMP is not intended to define general purpose ontology for SOA. Instead, the specification references the work of *The Open Group* (`http://www.opengroup.org/`), defining how it is integrated and used in the context of S-RAMP. S-RAMP is focused on publication and query of documents based on their content and metadata.

This specification will be used together with **Service-Aware Interoperability Framework** (**SAIF**) further for defining lightweight service repository taxonomy, which is suitable for service lookup and dynamic invocation by agnostic composition controllers.

Service definitions, routing, and reliability

The core standards that define services' contracts and reliable communications will be covered in the following sections.

WSDL

The details of WSDL are as shown in the following table:

Authority	Primarily addresses	Latest release
W3C	Reusability, Loose Coupling, and Discoverability, Composability	`http://www.w3.org/TR/wsdl20/`

Web Services Description Language Version 2.0 (**WSDL 2.0**) provides a model and an XML format for describing web services. Description has abstract and concrete parts. In the abstract part, we can define what messages can participate in which operations; consequently, in request and response, we can define what fault message can be generated. The concrete part binds operations with related messages to the physical service endpoint shown as follows:

```
<!-- WSDL definition structure -->
<definitions name="cargoBooking"
  targetNamespace="http://erikredshipping.com/booking/"
  xmlns="http://schemas.xmlsoap.org/wsdl/">
<!-- abstract definitions -->
   <types> ...
```

```
      <message> ...
      <portType> ...
   <!-- concrete definitions -->
      <binding> ...
      <service> ...
   </definition>
```

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

WSDL is equally important for implementing the `WS-*` specification as SOAP. Practically, all elements can be linked to the WS policy statements, describing behavior, data requirements, or QoS declarations. One of the most important pieces of information along with the declared data models / types (XSD) and operations (via declared canonical expressions such as `get`, `process`, and so on) is to bind the message exchange pattern.

The WSDL **Message Exchange Patterns** (**MEPs**) 2.0 are as follows:

- The **in-out pattern**, which is the standard request-response operation. This is the most common pattern.
- The **out-in pattern**, where a service provider initiates the interchange.
- The **in-only pattern**, which is the regular fire-and-forget MEP.
- The **out-only pattern**, which is the reverse of the in-only pattern.
- The **robust in-only pattern** is similar to the previous one, but it comes with the capability to provide the fault response message back if there is an error.
- The **robust out-only pattern** is similar to the out-only pattern, but it provides the optional fault message.
- The **in-optional-out pattern** is similar to the in-out pattern, but here, the response message is optional. This pattern also supports the generation of a fault message.
- The **out-optional-in pattern** is the reverse of the in-optional-out pattern, where the incoming message is optional. Here, generation of a fault message is supported.

We will touch upon some of these MEPs later, discussing the `WS-*` specification's roadmap.

WS-Addressing

The details of WS-Addressing are as shown in the following table:

Authority	Primarily addresses	Latest release
W3C	Loose Coupling and Composability	`http://www.w3.org/Submission/ws-addressing/`

This standard is the keystone for the whole `WS-*` stack, as all other standards are actively using it. It provides a neutral way of distributing messages. The SOAP header is the placeholder for all key elements, and they are presented in the following table:

wsa: Element	Description
`wsa:MessageID`	This property presents the ID of a message that can be used to uniquely identify a message.
`wsa:To`	This property provides the destination URI, and it indicates where the message will be sent to. If not specified, the destination defaults to `http://www.w3.org/2005/08/addressing/anonymous`.
`wsa:From`	This property provides the source endpoint reference, and it indicates where the message came from.
`wsa:ReplyTo`	This property provides the reply endpoint reference, and it indicates where the reply for the request message should be sent. If not specified, the reply endpoint defaults to `http://www.w3.org/2005/08/addressing/anonymous`.
`wsa:RelatesTo`	This property conveys the message ID of a related message along with the relationship type.
`wsa:FaultTo`	This property provides the fault endpoint reference. It indicates where the fault message should be sent to if there is a fault. If this is not present, usually the fault will be sent back to the endpoint where the request came from.
`wsa:Action`	This property displays the action related to a message. For example, the `wsa:Action` property can be used to identify the operation to be invoked upon receiving a request message. It must be provided in the message addressing properties of a message.
`wsa:ReferenceParameters`	This property references parameters that need to be communicated.

WS-Addressing allows the sending of messages to the specific instance of a service, which is not possible by WSDL concrete bindings alone. The SOAP request and response implementation is as follows:

Request	Response

The code for request implementation is as follows:
`<soapenv:Envelope`
`xmlns:soapenv ="http://www.`
`w3.org/2003/05/soap-envelope"`
`xmlns:wsa="http://www.`
`w3.org/2005/08/addressing/">`
`  <soapenv:Header>`
`    <wsa:MessageID>`
`      http://example.com/`
`someuniquestring`
`    </wsa:MessageID>`
`    <wsa:ReplyTo>`
`<wsa:Address>http://example.com/`
`Myclient</wsa:Address>`
`    </wsa:ReplyTo>`
`    <wsa:To>`
`      http://example.com/fabrikam/`
`Purchasing`
`    </wsa:To>`
`    <wsa:Action>`
`      http://example.com/fabrikam/`
`SubmitPO`
`    </wsa:Action>`
`  <soapenv:Header>`
`  <soapenv:Body>`
`    ...`
`   </soapenv:Body>`
`</soapenv:Envelope>`
	The code for request implementation is as follows:
	`< soapenv:Envelope`
	`  xmlns:soapenv ="http://`
	`www.w3.org/2003/05/soap-`
	`envelope"`
	`  xmlns:wsa="http://www.`
	`w3.org/2005/08/addressing">`
	`  <soapenv:Header>`
	`    <wsa:MessageID>http://`
	`example.com/`
	`someotheruniquestring</`
	`wsa:MessageID>`
	`    <wsa:RelatesTo>http://`
	`example.com/`
	`someuniquestring</`
	`wsa:RelatesTo>`
	`    <wsa:To>http://example.`
	`com/MyClient/wsa:To>`
	`    <wsa:Action>`
	`      http://example.com/`
	`fabrikam/SubmitPOAck`
	`    </wsa:Action>`
	`  </soapenv:Header>`
	`  <soapenv:Body>`
	`    ...`
	`  </soapenv:Body>`
	`</soapenv:Envelope>`

Requirements for addressing in WSDL are presented in the following fragment:

```
<binding name="cargoBookingPortBinding" type="tns:cargoBooking">
    <wsaw:UsingAddressing wsdl:required="true" />
......
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
 style="document"/>
    <operation name="bookCargoUnit">
```

```
        <soap:operation soapAction=""/>
        <input>
          <soap:body use="literal"/>
        </input>
        <output>
          <soap:body use="literal"/>
        </output>
        <fault name="missingCargoId">
          <soap:fault name=" missingCargoId" use="literal"/>
        </fault>
      </operation>
    </binding>
```

WS-ReliableMessaging

WS-ReliableMessaging provides an interoperable protocol that a **Reliable Messaging** (**RM**) source and RM destination are used to provide the application source and destination with a guarantee that a message that is sent will be delivered:

Authority	Primarily addresses	Latest release
OASIS	Composability, Loose Coupling,	• `http://docs.oasis-open.org/ws-rx/wsrm/200702` • `http://specs.xmlsoap.org/ws/2005/02/rm/ws-reliablemessaging.pdf`

The guarantee is specified as a delivery assurance. The protocol supports the endpoints by providing these delivery assurances. It is the responsibility of the RM source and the RM destination to fulfill the delivery assurances or raise an error. It would be right to see the analogy between WSRM and JMS in the Java world in terms of delivery assurance. The key differences are that JMS is highly platform-specific with a standard API, whereas WSRM is platform-independent by means of SOAP (and WSDL). Of course, WSRM agents (handlers) must be implemented behind services WSDLs and also on the client side to retransmit the message if necessary or provide the acknowledgement; however, these agents are invisible at the service/application's interaction levels. WSRM is an extension of SOAP, and all of its protocols are based on the concept of **sequence**. Sequence is the number of predefined steps, shown as follows:

- `CreateSequence`
- `CreateSequenceResponse`

- `CloseSequence`
- `CloseSequenceResponse`
- `TerminateSequence`
- `TerminateSequenceResponse`

You can see the latest specification on the official OASIS site.

Reliable messaging in a WSDL implementation is as shown in the following code snippet:

```
<?xml version="1.0" encoding="utf-8"?>

<wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
                  xmlns:xs="http://www.w3.org/2001/XMLSchema"
                  xmlns:wsa="http://www.w3.org/2005/08/addressing"
                  xmlns:wsam="http://www.w3.org/2007/
                  05/addressing/metadata"
                  xmlns:rm="http://docs.oasis-open.org/ws
                  -rx/wsrm/200702"
                  xmlns:tns="http://docs.oasis-open.org/
                  ws-rx/wsrm/200702/wsdl"
                        targetNamespace="http://docs.oasis
                        -open.org/ws-rx/wsrm/200702/wsdl">

  <wsdl:types>
    <xs:schema>
      <xs:import namespace="http://docs.oasis-open.org/ws-rx/
wsrm/200702"
      schemaLocation="http://docs.oasis-open.org/ws-rx/wsrm/200702/
wsrm-1.1-schema-200702.xsd"/>
    </xs:schema>
  </wsdl:types>

  <wsdl:message name="CreateSequence">
    <wsdl:part name="create" element="rm:CreateSequence"/>
  </wsdl:message>
  <wsdl:message name="CreateSequenceResponse">
    <wsdl:part name="createResponse" element="rm:CreateSequenceRespon
se"/>
  </wsdl:message>
   ....
  <wsdl:message name="TerminateSequence">
```

```
      <wsdl:part name="terminate" element="rm:TerminateSequence"/>
    </wsdl:message>
    <wsdl:message name="TerminateSequenceResponse">
      <wsdl:part name="terminateResponse" element="rm:TerminateSequence
Response"/>
    </wsdl:message>

    <wsdl:portType name="SequenceAbstractPortType">
      <wsdl:operation name="CreateSequence">
        <wsdl:input message="tns:CreateSequence" wsam:Action=
"http://docs.oasis-open.org/ws-rx/wsrm/200702/CreateSequence"/>
        <wsdl:output message="tns:CreateSequenceResponse"
wsam:Action="http://docs.oasis-open.org/ws-rx/wsrm/200702/
CreateSequenceResponse"/>
      </wsdl:operation>
     ....
      <wsdl:operation name="TerminateSequence">
        <wsdl:input message="tns:TerminateSequence" wsam:Action=
"http://docs.oasis-open.org/ws-rx/wsrm/200702/TerminateSequence"/>
        <wsdl:output message="tns:TerminateSequenceResponse"
wsam:Action="http://docs.oasis-open.org/ws-rx/wsrm/200702/
TerminateSequenceResponse"/>
      </wsdl:operation>
    </wsdl:portType>

</wsdl:definitions>
```

Policy and metadata

The service metadata describes what is needed for the service consumers, including composition controllers to establish successful interchange sessions with service provider(s). Some of the WS specifications are described in the following sections.

WS-MetadataExchange

Almost all elements of WSDL can be perceived as `metadata: XSD` structures, and also as message data types, policies, provider's capabilities, requirements for transaction control, or reliable messaging. The details of WS-MetadataExchange are as shown in the following table:

Authority	Primarily addresses	Latest release
W3C	Discoverability, Composability, Loose Coupling	`http://www.w3.org/TR/ws-metadata-exchange/`

These metadata elements can be pushed or pulled as a whole or partially from the service provider's contract. In general, these standards describe the way of encapsulating this data and the extraction protocols. Some elements responsible for pulling or pushing data and its encapsulation are presented in the following table:

Element	**Description**
`mex:GetMetadata`	A requester may send a `GetMetadata` request message to an endpoint to retrieve the metadata associated with that endpoint. This operation may be supported by the `WS-MetadataExchange` compliant service endpoints.
`mex:MetadataReference`	This is an endpoint reference to a metadata resource and is of the type `EndpointReferenceType` as defined by `WS-Addressing`. This metadata resource must support the `Get` operation, `WS-Transfer`, to allow the retrieval of the metadata unit for the `MetadataSection` class's dialect and identifier.
`mex:Location`	The `mex:Location` element may be used to specify a reference to an HTTP metadata resource. A requester may use an HTTP `GET` operation on the indicated URL to retrieve the metadata.
`mex:Metadata`	This contains one `MetadataSection` child class for each distinct unit of metadata. When there is a large amount of metadata, the children should contain `MetadataReferences` or `MetadataLocations` instead of the actual information.

Standard Business Document Header (SBDH)

The SBDH standard provides a document header that identifies a key data about a specific business document. The details of SBDH are shown in the following table:

Authority	**Primarily addresses**	**Latest release**
UN/CEFACT GS1	Discoverability	`http://www.gs1tw.org/twct/gs1w/download/SBDH_v1.3_Technical_Implementation_Guide.pdf`

Since SBDH standardizes the way data is presented, the data elements within SBDH can be easily located and leveraged by multiple applications. SBDH is a business document header and should not be confused with a transport header. It is created before the transport routing header is applied to the document and is retained after the transport header is removed.

Although SBDH is not the transport header, data in it can be used by transport applications to determine the routing header since it does contain the sender, receiver, and document details. It can also be used by business applications to determine the appropriate process that is to be performed on the business document. The specifications are explained in the following section.

WS-Policy

This very wide specification establishes conditions and restrictions for a service's invocation and consequently for all compositions it may participate in. The details of WS-Policy are as follows:

Authority	W3C
Primarily addresses	Composability, Loose Coupling, and Abstraction
Latest release	`http://www.w3.org/TR/ws-policy/`

Most importantly, when compared with human-readable SLAs, these conditions are expressed in a machine-readable form. In fact, this specification has a lot of common features with the metadata exchange standard, but it's far wider as it expresses service requirements and preferences regarding all other `WS-*` specifications in a detachable form, presented as an XML policy file. This separation allows you to centralize all policies and present them in a hierarchical way, simplifying attachment to the provider's WSDL.

Some delivery assurance elements along with their descriptions are as follows:

Element	Description
`AtMostOnce`	Messages are delivered at most once, without duplication. It is possible that some messages may not be delivered at all.
`AtLeastOnce`	Every message is delivered at least once. It is possible that some messages are delivered more than once.
`ExactlyOnce`	Every message is delivered exactly once, without duplication.
`InOrder`	Messages are delivered in the order that they were sent. This delivery assurance can be combined with one of the preceding three assurances.

The `WS-Policy` delivery assurance elements are typically used together with other `WS-*` specifications for enforcing certain operational requirements, such as reliable messaging timeout and acknowledgement interval, as shown in the next code snippet.

The following code will help us understand how the policy is defined as a child element of the `wsdl:definitions` element:

```
<wsp:Policy wsu:Id="RMAcknowledge_policy">
    <wsp:ExactlyOne>
        <wsp:All>
            <wsaw:UsingAddressing/>
            <wsrm:RMAssertion>
                <wsrm:AcknowledgementInterval Milliseconds="500"/>
                <wsrm:InactivityTimeout Milliseconds="100000"/>
            </wsrm:RMAssertion>
        </wsp:All>
    </wsp:ExactlyOne>
</wsp:Policy>
```

The last step will be the referencing of the policy with the child element of the `wsdl:binding` element shown as follows:

```
<wsdl:binding name="testWsRmBinding" type="tns:TestWSRM">
    <wsp:PolicyReference URI="# RMAcknowledge_policy "/>
…...
```

Transaction control and activity coordination

The specifications of transaction control and activity coordination are explained in detail in the following sections.

WS-Coordination

The details of WS-Coordination are as shown in the following table:

Authority	Primarily addresses	Latest release
OASIS	Reusability, Composability	`http://docs.oasis-open.org/ws-tx/wstx-wscoor-1.1-spec-errata-os/wstx-wscoor-1.1-spec-errata-os.html`

This specification is the foundation for the service federation, allowing services from different business and technology domains to coordinate their interoperability operations, both long- and short-term living. According to this specification, it's realized in two stages:

- In the first phase, all participants registered in one unified coordination context and communication protocols are agreed upon

- In the second phase, all registered participants exchange messages according to the protocols and rules of engagement established during the registration phase

This specification is responsible for the following:

- Creating and formatting `CoordinationContext` that contains registration information, which is mandatory for all participants
- Establishing a coordination protocol based on `CoordinationContext`, and the ways of distributing this context between participants
- Establishing a registration protocol

`CoordinationContext` in the header of a SOAP message can be seen in the following code snippet:

```
<wscoor:CoordinationContext>
    <wsu:Expires>2014-03-21T00:00:00.0000000-05:00</wsu:Expires>
    <wsu:Identifier>
        uuid:0d13748c-7a09-8520-a911-17c73f09ac82
    </wsu:Identifier>
    <wscoor:CoordinationType>
        http://schemas.xmlsoap.org/ws/2003/09/wsat
    </wscoor:CoordinationType>
    <wscoor:RegistrationService>
        <wsa:Address>
            http://erikredshipping.com/ShcheduleCoordinationService/
RegistrationCoordinator
        </wsa:Address>
        <wsa:ReferenceParameters >
            ....
            <refApp:App1> ... </refApp:App1>
            <refApp:App2> ... </refApp:App2>
            ....
        </wsa:ReferenceParameters >
    </wscoor:RegistrationService>
</wscoor:CoordinationContext>
```

The `WS-Coordination` specification is the foundation for other specifications, setting the standards for complex coordinated activities: `WS-AtomicTransaction` and `WS-BusinessActivity`. Both of these define a type of agreement coordination that addresses the needs of complementary classes of activities, ACID- and BASE-type requirements, as discussed later in this chapter.

Security

Security standards and specifications will be discussed in *Chapter 2*, *An Introduction to Oracle Fusion – a Solid Foundation for Service Inventory*, and *Chapter 7*, *Gotcha! Implementing Security Layers*, dedicated to security and Oracle's approach for its implementation.

Interconnected WS-* standards

To understand how `WS-*` standards could help solve some problems that are not addressed by the simple model of contemporary SOA, we will walk through the process-identification flow diagram shown in the next figure. This diagram is based on the interactions of web-based services as native areas of the `WS-*` standards implementation. We will start with the definition of the process as a number of services invocations, sequential or parallel, with different durational and transactional requirements.

First, let's identify whether it is a long or short running process. Longevity is the subject of technical and/or business timeouts. The first one is related to a time slot; here, your services in composition could hold on to the active state without draining too many resources. The second is set by business requirements and can be quite substantial (days, weeks, and so on). Moving by the left lane, we will first look at the synchronous services and the standards associated with them. In common cases, we could assemble service compositions from different domains, so a **Service Broker** will be compulsory; it will help us resolve the disparity of the data models, formats, and protocol bridging as well.

Also, brokering means that routing and mediation could be necessary for complex service activities when more than two services are involved. In these cases, Service Broker will act as a **Composition Controller**. Both of them are SOA patterns, arguably most-commonly used, and we will see their implementation in detail soon. If it's just a synchronous single interaction between a service consumer and a service provider, it's called a **primitive** activity. In general, the following two major standards must be taken into consideration:

- WSDL should have a binding to a proper MEP; otherwise, communication simply will not be possible.
- The `WS-ReliableMessaging` service must be implemented if a feeble connection or slow response from a service message-receiver could affect service activities. The application of this standard will ensure that the message is delivered or at least provide an acknowledgement about the state of the delivery.

Let's move on to the middle lane. If more than two participants in a synchronous composition can be expected, for example, one composition initiator and two or more composition members, the number of `WS-*` standards and specifications involved will be doubled. Composition Controller becomes mandatory in this scenario. In addition to WSDL MEPs and WS-RM from primitive activities, the transaction coordination based on `WS-AtomicTransaction` arguably will be one of the most important specifications for these types of composition. It will address typical ACID requirements related to multiphase commits:

- At design time, define all the **Atomic Transaction Coordinators** (**ATC**) phases in WSDL with specific messages such as `wsat:Prepare`, `wsat:Prepared`, `wsat:Aborted`, `wsat:Commit`, `wsat:Rollback`, `wsat:Commited`, and `wsat:Reply`. Bind ATC messages to the operations.
- At runtime, initiate the transaction, send messages, collect votes, commit if all participants have voted positively, send the rollback signal if any vote is not received in the designated time slot or is negative, repeat operations if necessary, and propagate the response to the initial sender.

With many services involved in this complex activity, some from different domains with different data models (which is common in the adapter technology layer), requirements for complex transformations could be anticipated. The transformation technique could be different, of course; sometimes, you may come across a proprietary technique. However, here we will discuss the standard XSLT/XQuery approaches. Most common complex transformations that we will be coming across in our designs are as follows:

- **The message aggregations design**: Here, we combine several messages in one that are further processed by a service that can accept only coarse-grained data models. If messages arrive sequentially, XSLT alone will not be enough; some transformation's intermediate store will be necessary if we do not want to keep the messages in.
- **The message debatching or splitting design**: Here, we separate one message that contains a batch of similar messages into a sequence of messages suitable for processing by a service with a fine-grained data model. This task is very common; inbound messages usually in a non-XML format and whole processing will require a `WS-RM` implementation, as ordered sequential processing is involved in providing possible acknowledgements for batch or individual messages.

Finally, for the middle lane, the most common standard `WS-Addressing` will be compulsory if message mediation is required for complex service activities. Usually, it's presented as a content-based routing, sometimes based on rules handled by the rule engine; static routing tables are also common.

This concludes the mapping of synchronous service activities to the technology standards and `WS-*` specifications.

Asynchronous services are more complex, as more infrastructural technical elements are required as follows:

- Asynchronous queuing will require server resources such as topics and/or queues
- State repository will be required for resource hibernation during the composition's inactive state

Their realization is not covered by any particular `WS-*` specification, but their presence is significant for the `WS-BusinessActivity` specification's implementation. The `WS-BusinessActivity` specification together with `WS-AtomicTransaction` and support from `WS-Coordination` has presented the mechanism controlling service activities over a long period of time. The **Business Activity** coordination protocol can be perceived as a chain of discrete over-the-time atomic coordinations. As it is clear from the context, rollback actions are virtually impossible between different stages of business activities, so the possibility to have an arbitrary compensative transaction is the essential part of this protocol. Using compensation, it is possible to return the data of an application's participant to a consistent state, but not exactly to the state before the transaction, such as the Atomic Coordination. The complexity of compensative transactions could be very high, allowing you to correct the changes happened many steps before the actual error.

Further implementation of standards and recommendations in the right lane is similar to implementing complex synchronous activities; this confirms the fact that an asynchronous business activity is a chain of complex synchronous activities. A special significance of asynchronous transactions is that it has a `WS-Coordination` specification for its role of establishing a coordinator service model we mentioned earlier.

The composition of a coordinator service model consists of the following services:

- The activation service, which creates new coordination contexts and associates them with the planning activity
- The registration service, which registers a service's participant and distributes coordination contexts among them
- Protocol-specific services, which represent the protocols supported by the coordinator's coordination type
- The coordinator controller service of this composition, also known as the coordination service

A key element of the coordination context is the correlation key, which is common for all activities in a particular composition. WS-Addressing's elements `wsa:ReferenceParameters` and `wsa:ReplyTo` could be employed as a container for correlation ID and address, where the response will be sent:

```
<soap:Header>
   <wsa:MessageID>uuid:35f19ca8-c9fe</wsa:MessageID>
   <wsa:Action>http://erikredcarrier.com/ship</wsa:Action>
   <wsa:To>http://www.portaquaba.com:7070/ShippingService</wsa:To>
   <wsa:ReplyTo>
     <wsa:Address>
         http://www.portbremen.com:7777/response
     </wsa:Address>
     <wsa:ReferenceParameters>
            <customHeader>correlationKey</customHeader>
      </wsa:ReferenceParameters>
   </wsa:ReplyTo>
...
```

Combined together, these elements form the SOA pattern called **Service Callback**, which is most commonly used in asynchronous communications.

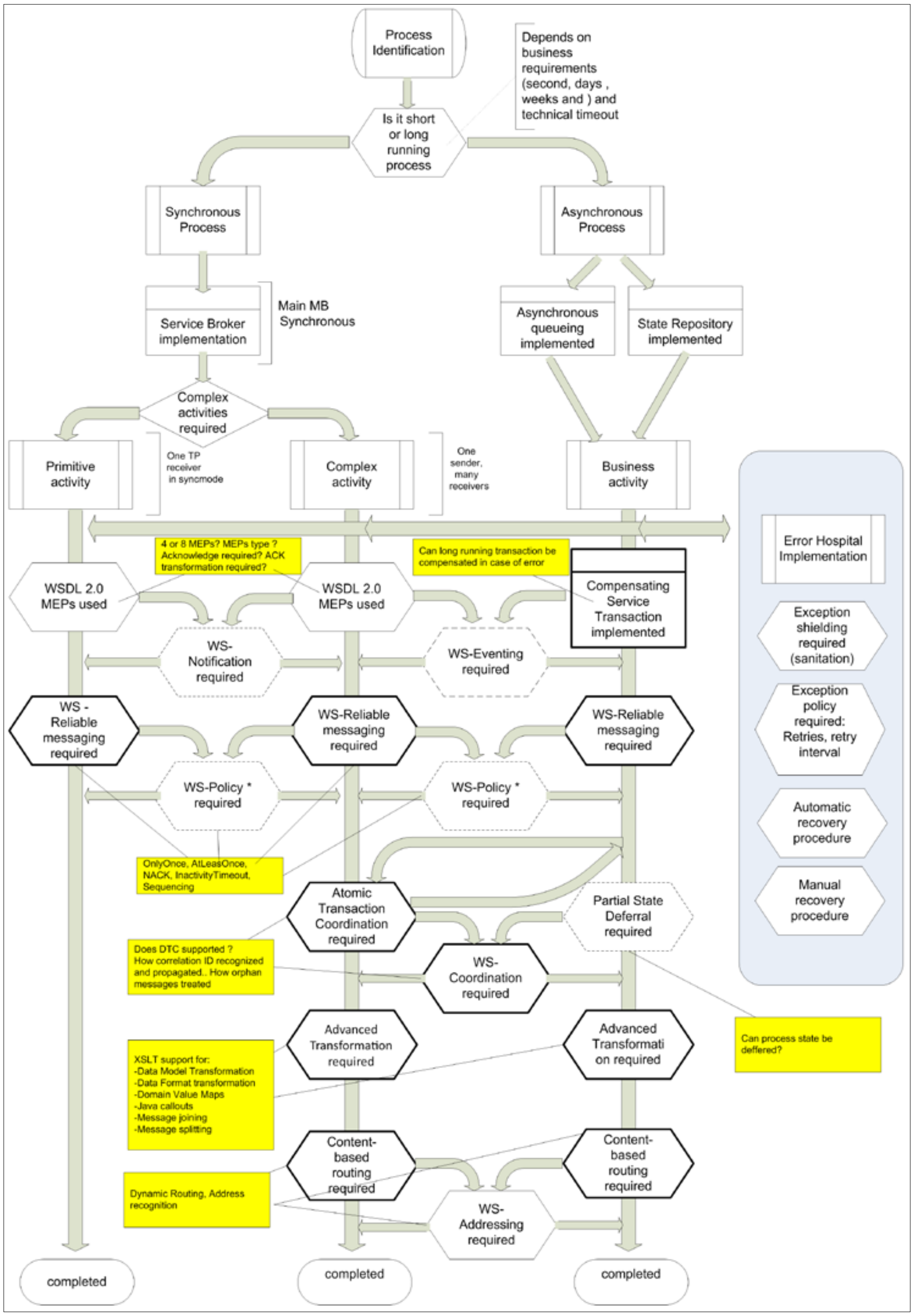
Another pattern that is common to all types of service interactions related to the service is the **Policy Centralization SOA** pattern. The `WS-Policy` specification is a machine-readable language for representing these web service capabilities and requirements as policies. A policy makes it possible for providers to represent such capabilities and requirements in a machine-readable form. A separate policy file such as `policy.xml` can contain several policy expressions that are grouped under different WS-Utility identifiers (`wsu:Id`) for simplified referencing, both external (`http://erikredcarrier.shipping.com/policy.xml#common`) and internal (just `#common`); please see the `PolicyReference` examples shown in the following code snippet:

```
<Policy wsu:Id="common">
   <wsap:UsingAddressing />
   ....
   <!--  other policies with usage attributes : wsp:Optional="true"
-->
</Policy>
```

The representation `Policy` to the service consumers is designed by binding policy expressions to WSDL elements, to service operation, for example as shown in the following code:

```
<wsdl:binding name="AddressingBinding" type="tns:RealTimeDataInterfa
ce" >
  <PolicyReference URI="#common" />
  <wsdl:operation name="getCargoStatus" >...</wsdl:operation>
  ...
</wsdl:binding>
```

The last standards, presented as optional for all lanes are related to complex event processing specifications: WS-Notifications (OASIS) and WS-Eventing (W3C). In a nutshell, both of them describe the publish-subscribe protocols with the propagation of events that follow right from the publisher to the subscriber, some sort of fire-and-forget messaging patterns. It is important to remember that both subscribers and consumers are not always the same actors; most interestingly, for Brokered Notification, some form of Service Broker called Event Broker will be employed for distributing events between multiple consumers.


Process Identification
Depends on business requirements (second, days , weeks and) and technical timeout
Is it short or long running process
Synchronous Process
Asynchronous Process
Service Broker implementation
Main MB Synchronous
Asynchronous queueing implemented
State Repository implemented
Complex activities required
Primitive activity
One TP receiver in syncmode
Complex activity
One sender, many receivers
Business activity
Error Hospital Implementation
4 or 8 MEPs? MEPs type ? Acknowledge required? ACK transformation required?
Can long running transaction be compensated in case of error
WSDL 2.0 MEPs used
WSDL 2.0 MEPs used
Compensating Service Transaction implemented
WS-Notification required
WS-Eventing required
Exception shielding required (sanitation)
Exception policy required: Retries, retry interval
WS - Reliable messaging required
WS-Reliable messaging required
WS-Reliable messaging required
WS-Policy * required
WS-Policy * required
Automatic recovery procedure
Manual recovery procedure
OnlyOnce, AtLeasOnce, NACK, InactivityTimeout, Sequencing
Atomic Transaction Coordination required
Partial State Deferral required
Does DTC supported ? How correlation ID recognized and propagated.. How orphan messages treated
WS-Coordination required
Can process state be deffered?
Advanced Transformation required
Advanced Transformati on required
XSLT support for:
-Data Model Transformation
-Data Format transformation
-Domain Value Maps
-Java callouts
-Message joining
-Message splitting
Content-based routing required
Content-based routing required
Dynamic Routing, Address recognition
WS-Addressing required
completed
completed
completed

SOA Standards and Patterns implementation roadmap

We didn't cover all the existing `WS-*` specifications, but the ones described earlier are quite capable of dealing with challenges, something that the contemporary SOA is unable to address. These specifications were under development during the last ten years, and some of them are still evolving; however, overall, the technology stack based on `WS-*` is very mature. Although some of the specifications are a bit complex, the fact that all of them are based on clearly defined principles has made them commonly acceptable and adoptable not only by main market players/sponsors of standardization committees, but also by open source communities, acting as some of the most valuable contributors toward the proliferation of these standards.

Abstract principles and vendor-neutral standards do not exist in a vacuum; they have practical service boundaries as we have seen during the discussion of the standards' implementation roadmap. We even touched upon some roles of patterns discussing principle-standards relations. These relations could have many forms and dependencies with regards to the infrastructural areas of service implementation. Every area has its own distinctive characteristics, a proprietary for every single step of service's lifecycle such as analysis, modeling, development, testing, and implementation.

Collectively, they are shaping service boundaries in the form of complex building blocks that connect internal information assets with external consumers of those assets.

Distinctive characteristics of these blocks are formalized in a collection of design rules, one for each particular area; however, collectively, they aim at the same goal of maintaining a desirable level of composability for services that exist in these ecosystems.

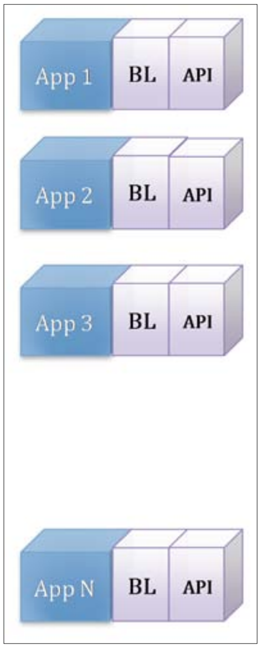
SOA frameworks

Framework is one of the most commonly used terms, not just only in IT (probably, together with pattern). It is also commonly said that SOA is a framework in itself. This means that the SOA framework is dedicated to the technological and operational areas for the implementation of SOA Principles in order to achieve the predefined goals. One problem here though is that it's too often mentioned that principles must be applied in a meaningful context.

Too much about framework, which by its dictionary definition should show some precision as mentioned in *The American Heritage® Dictionary of the English Language, Fourth Edition* copyright ©2000 by the *Houghton Mifflin Company*:

1. A structure for supporting or enclosing something else, especially a skeletal support used as the basis for something being constructed.
2. A fundamental structure, as for a written work.
3. A set of assumptions, concepts, values, and practices that constitutes a way of viewing reality.

So, maybe from reviewing SOA realities, we could identify the number of frameworks it consists of and their distinguishing characteristics, as we can see in the following figure:



The starting point would be the realization of the ultimate goal of any business application's infrastructure to have a hot-pluggable collection of business applications (**App1** to **App N**), each providing and effectively serving its own functional context. All functional contexts have strictly defined non-overlapping boundaries; physical realization of applications has a high level of technical autonomy, which in fact makes these applications hot-pluggable. At any given point in time, an application that becomes technically obsolete or functionally irrelevant can be replaced with its more modern and cost-efficient equivalent.

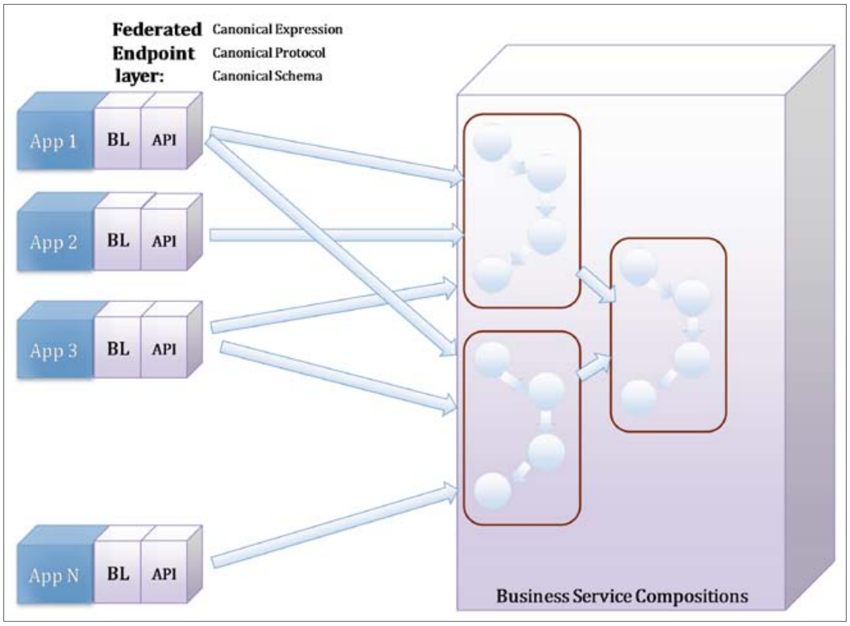
As long as all functional boundaries are precise and individual, **business logic** (**BL**) encapsulated within each application is transpired and comprehendible. It can be clearly separated from application resources and modeled/remodeled using platform-independent means as there are no parasitic couplings between separated elements of business logic of any kind. Every application presents public operations and related data models via standardized APIs using common notations, available for all involved parties in a comprehendible way.

A number of communication protocols and exchange patterns have been reduced to reasonable minimum with well defined timeouts and compensation activities if there are occasional miscommunications.

What we just described is a collection of services, as they should be designed from the beginning. A service can be perceived as an application if a functional context permits this without affecting Loose Coupling and Abstraction negatively.

The following two assumptions are distinctive:

- **Assumption 1**: Services have non-overlapping functional boundaries, encapsulating the logic that is specific only for this service.
- **Assumption 2**: Services allow you to access the service logic only via a publicly available service contract, which is expressed using the canonical data model and canonical expressions for capabilities. Canonical protocols complete the picture.



In this case, the implementation of a new business process would be achieved by simply recomposing existing service capabilities in the sequence and duration as per the new functional context.

We will need some hypothetical area where this recomposition will be possible technically. There shouldn't be any problem, as the protocol, data, and interfaces stay homogenous. Any programming language or platform will do if the result of the recomposition—the new service—follows the initially declared standards for the services.

The fact that a newly composed application is also the service is important, as this composition could potentially be part of an even bigger composition; therefore, all strict standards must be applied all the way. This idealistic picture is pretty close to the contemporary SOA model. We have only one solid framework here that is used for business service composition, and with this, we practically accomplish very little. All our compositions would be just invocations of existing APIs' capabilities with values' assignments in between. No transformation is required as all data models are CDM-compliant, and a unified protocol eliminates the necessity of the bridging protocol.

The first real wake-up call would come from the realization that not all our APIs really are parts of the canonical Federated Endpoint Layer (as explained in *Chapter 6*, *Finding the Compromise – the Adapter Framework*). The *Assumption 2* item mentioned in the previous bullet list is quite often too optimistic in real life; this means that:

- Data model transformation and data format transformation (translation) could be needed.
- There could be no APIs at all. Even worse, data required for a new composite service will be pulled from various sources, sometimes with multiple data cleansing and filtering routines extended over a period of time.
- Complex transformations will be performed to maintain the initially declared data granularity on service composition, such as aggregation or debatching.
- Messaging and/or transport protocols are not canonical, and this disparity must be harmonized.

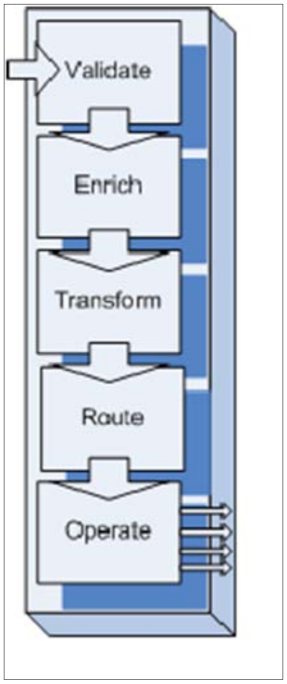
The Application Business Connector Services framework

Following the principle of separation of concerns, we must preserve our Business Service Composition's hypothetical area that is free from all disparities; this is suitable only for fast and clean compositions. Thus, the implementation of Application Business Connector Services is compulsory. This is the first concrete framework in the list of SOA frameworks we are going to discuss. The sole purpose of this framework is to compensate for the lack of the service(s) to present a standardized contract, suitable for repeatable reuse and utilization. Services residing in this framework are special forms of wrappers, designed to receive/extract, translate, transform, filter, validate, and propagate (route) further information required for business compositions, that is, implement the **VETRO pattern** (`http://www.oracle.com/technetwork/articles/soa/jellema-esb-pattern-1385306.html`). You can see the required functionalities, implementation techniques, and service models for this framework in the following list of requirements:

- Implementation technique:
    - Synchronous implementation for simple MEPs
    - Asynchronous transaction coordinator for adapters (data-collectors), handling long-running data aggregation transactions
- Service models:
    - Adapter services (Legacy Wrappers, File Gateways, FTP Hotels, and so on) are utility services in general
- Required functionalities:
    - Availability of Protocol adapters such as SOAP.
    - Availability of Transport Adapters available such as JMS/MQ, AQ, and HTTP.
    - Availability of Application adapters such as OEBS, Siebel, and so on.
    - Availability of Component adapters such as DBs for instance.
    - Presence of Atomic Transaction Coordinator at the adapter level, and the implementation of the voting mechanism.
    - WS-ReliableMessaging support.
    - Data model and data format transformations support (types and engines) and protocol bridges (`SOAP<->REST`).
    - Automated fault handlers (retry mechanisms) for southbound adapters mostly.

  - Implementation of the FTP(S) Gateway pattern.
  - Implementation of the File Gateway pattern.
  - Filesystem objects' (FSO) replication such as files and folders/subfolders.
  - Reverse proxy realization for security reasons.
  - API for JCA adapters that provides the possibility to create your own adapters.
  - Reliable store-and-forward queue mechanisms with full message traceability and no message losses.
  - Message filtering mechanisms, suppressing unwanted responses.
  - Multiconsumer queues' support.
  - Implementation of the high-availability reliable adapters.

The following figure explains the presented list:



This framework is kind of special. Ideally, it is not supposed to exist; yet, it is one of the most common and, at the same time, heavy frameworks in use. It's heavy with embedded functionalities, practically all `WS-*` standards employed here in order to support the VETRO pattern's implementation.

There is a strong reason for viewing these types of services as temporary services, exiting only during the time of the transition of incapable application to a consistent service state with all SOA principles properly applied. This transition could be very time consuming, so some characteristics should be observed constantly in order to not create hybrid services and make the situation worse.

The ABCS design rules usually are as follows:

- Northbound ABCS receives any type of messages and produces a canonical model (CDM) for the service composition layer. Inbound Debatching/Aggregation is performed here.
- Southbound ABCS gets CDM and reliably delivers application-related messages.
- ABCS is responsible for consistently maintaining all message header elements.
- Error occurring in the active (poller) northbound ABCS will be not propagated further. Acknowledge the message returned to the northbound application if necessary, and a record is usually created in the audit log. Numbers of extraction retries could be unlimited, as a business transaction is not started yet. Still, limiting the number of retries can be wise.
- Error occurring in southbound ABCS is reported back to the composition controller or actual service worker. The number of retries allowed is limited and policy-based. Depending on the type of transaction (sync-async), ATC rules are applied.
- ABCS is highly tailored to the endpoint application service. It is not recommended to have one terminal adapter for several endpoints. It would be quite right to say that the adapter belongs more to a service-enabled application than an SOA inventory.
- If the number of identical adapters is substantial, the Adapter Factory pattern could be applied for instantiating particular adapters of similar types (only the actual endpoint is different and looked up from the registry). Adapter Factory Service acts as a coordinator and *does not* belong to the ABCS technology layer.
- Transaction Coordinator is a valid SOA pattern for ABCS if data extraction procedures require transactional control. For instance, some of the data must be extracted from the first DB and the extraction flags subsequently updated, then another portion of data extracted from second DB, and values changed again and after that consolidated and validated following the final update. We must remember that data extraction routines must not be mixed with business logic; otherwise, we unconsciously introduce a hybrid service on an adapter layer, scattering the logic over and making this solution very hard to maintain.

Again, from pure logic, ABCS's have no right to exist. If your application is not capable of being a reliable composition member, it is better to something about it, such as redesign, rewrite, retest, and reimplement. Alas, it doesn't work this way in real life. From the version control standpoint, ABCS's are inevitable; however, how many times did we witness that adapter services with complexities are compared to the application logic it tried to isolate and abstract. From very beginning though, it meant presenting a lightweight wrapper only. In most cases, that's the result of the second dissonance with reality, that is, *Assumption 1* is wrong and our application, even with proper WS API, is too bulky, sluggish, and not reliable to act as a composition member. Yes, implementation of ABCS could potentially help, but some SOA principles have to be sacrificed (original applications such as Autonomy, Loose Coupling, and Abstraction), which makes the situation even worse. Thus, a service must be redesigned in this scenario, as an adapter alone cannot improve its composability.

Obviously, two new linked frameworks have to be introduced, although there is nothing new about them; they are as follows:

- The Object Modeling and Design framework must be adopted for proper implementation of the separation of concerns principle. As a practical outcome of its application, functional boundaries of the services must be rearranged; autonomy of the services improved due to resource isolation, if it's possible; and the negative couplings resolved by the application of logic-to-contract positive coupling.
- The XML Design framework as a support of canonical business model must be presented. XML message is a serialized transportable representation of the business object, encapsulated in service logic. Therefore, this framework in general is the extension of objects' modeling and design.

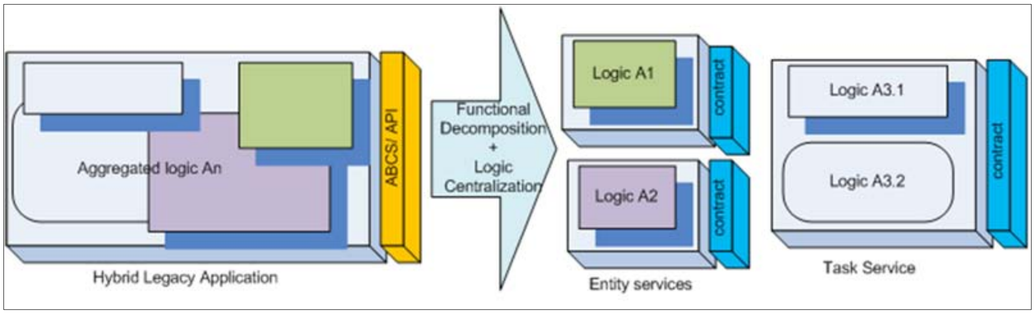
It is hard to say which one of these two frameworks is the primary one as both of them have a particular purpose, that is, to promote the Contract-First design rule as much as possible. Surely, this design rule is more applicable in the top-down approach when we have the possibility to design a service from scratch. Redesigning a bulky legacy application significantly limits our options, but functional decomposition together with the discussed ABCS's layer still makes it possible.

The Object Modeling and Design framework

The details of the GF02: Object Modeling and Design framework are as shown in the following table:

Implementation technique	Service models	Required functionality
Avoid the generation of a contract (WSDL) from the service logic	All service models that include Utility services, Entity Services, Task Services, and so on (with some limitations for orchestrated services)	Object-to-XML mappings
Identify all service metadata elements		Flexible XML marshalling
Register metadata elements in an individual service profile		Support of WS-Security
		Persistence support (mapping relational data to XML and object model)
		Complete support of object- and aspect-oriented programming (by default)

The following figure explains the table:



This framework is vendor-specific and is only related to the component/service implementation. Anything will do really, such as Spring Web Services / MVC, the .NET framework, and so on.

For functional decomposition, we will the need combined efforts of business analysts, service architects, and technical infrastructure experts for balancing SOA principles discussed previously with technical feasibility of redesigned services (in order to understand the physical level of decomposition required). To follow the Contract-First principle, any SDK is good.

The XML Modeling and Design framework

The third related framework, XML Design, is interrelated to Object Design and primarily concerned with establishing a canonical business model for the service inventory. This model is not a single entity; it's a collection of enterprise entities in the form of Enterprise Business Objects closely related to the existing DB models that describe primary enterprise assets such as Order, Invoice, and CargoUnit. One possible source of them is already mentioned in functional decomposition, which is the oldie but goodie reverse engineering fashion that can harvest these entities for us. With the top-down approach, these entities can be acquired from technology-specific forums that are related to business functional areas such as telecommunication, transportation, and healthcare. However, you must be quite skeptical about the amount of data presented on these resources; these specifications are some sort of all-weather cases that are oversized and over-bloated with elements you could never use in your business. By following them blindly, you can end up with a simple purchase order with two or three order lines of 3 Meg size. Another problem is that these models are quite illogically partitioned, mixing together all essential building blocks:

- **Qualified data types** (**QDT**)
- **Qualified data object** (**QDO**)
- Message header and process header elements
- Message tracking data

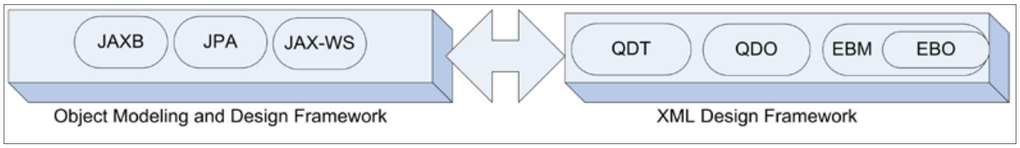
Amazingly, through some international projects, we also witnessed the implementation of regional data models using local languages. Yes, you won't believe it. An entire local messaging hierarchy (XML elements, attributes, documentation, and constraints) has been implemented in one Scandinavian language—so much for discoverability and composability, without mentioning the encoding issues.

The foundation for the XML framework has been well set by the *UN/CEFACT Document XML Naming and Design Rules* draft version published in August 2004. Please acquire the latest version and follow the design rules section (see *Appendix C, Naming and Designing the Rules List*). They are all important, from *[R 1]* to *[R 185]*; we just would like to quote the most important part from our point of view.

The following table lists the implementation techniques, service models, and some selected design rules using the contract-first approach, as follows:

Implementation technique	Service models	Selected Design rules (out of 185)
Avoid XSD generation from DB model	All: Utility Services, Entity Services, and Task Services	[R 4] ELEMENT, ATTRIBUTE AND TYPE NAMES MUST BE IN THE ENGLISH LANGUAGE, USING THE PRIMARY ENGLISH SPELLINGS PROVIDED IN THE OXFORD ENGLISH DICTIONARY.
Identify all EBO/EBM elements		[R 5] LOWER-CAMEL-CASE (LCC) MUST BE USED FOR NAMING ATTRIBUTES.
Follow Naming and Design Rules List		[R 6] UPPER-CAMEL-CASE (UCC) MUST BE USED FOR NAMING ELEMENTS AND TYPES.
Register message-related metadata elements in service repository		[R 13] A ROOT SCHEMA MUST BE CREATED FOR EACH UNIQUE BUSINESS INFORMATION EXCHANGE.
		[R 23] A QUALIFIED DATA TYPE SCHEMA MODULE MUST BE CREATED

The figure explaining the table is as follows:



Needless to say, these rules also must be applied wisely with proper understanding of all consequences. For example, following the rule *[R 60]*, that is, *THE XSD: ANY ELEMENT MUST NOT BE USED*, we would not be able to implement an agnostic composition controller using the message container concept. However, this rule is highly important for security reasons and must be strictly followed for ABCSs and in Perimeter Gateways implementations.

For now, we have identified three frameworks that are responsible for shaping and creating a service, establishing communication, and resolving interchange disparities. The Hypothetical Business Service Composition area remains untouched, but now it is the time to find out what is it, really. From the *standards roadmap* diagram, we realized that complex compositions could be synchronous with elements of atomic transaction coordination and asynchronous with running chains of complex communications, lasting for very extended period of time. Different technical standards and operational requirements delineate two new frameworks, each dedicated to its own type of service interactions, mostly based on time metrics. We will start with asynchronous service compositions, as they are more technically demanding.

The Enterprise Business Flows framework

Firstly, the presence of State Repository is a very distinctive feature, a proprietary for this framework. It will allow all business flows to be consolidated under this technical platform, deferring the transactional state while waiting for a response. It can be done by any form of DB such as Relational, XML, or NoSQL, depending on types of messages running in these flows. The Relational type is still one of the most common; it allows you to conduct a quick search of hibernated instances and associations with the designated Correlation ID. Possible glitches, resulting in leaving orphan-hibernated records, can also be more easily dealt with in a relational environment. The storage overhead, which is common for relational models, must be taken into consideration during the planning phase. Also, it's important to realize a unified Enterprise Business Flow framework's state repository can present a single point of failure and/or a performance bottleneck, so proper DBA administration, replication, and maintenance is one of the highest demands for this framework.

To make business flows more manageable and centralized, another element of technical infrastructure should be observed as a decision service is based on rules engine. We deliberately segregate the rules, associated with pure business logic and centralize them in one location, thus making it possible to employ dynamic message routing and process invocation, value mapping, and data sorting processes. Of course, taking some logic away from the compound process reduces its autonomy and also could present a single point of failure, but this calculated risk makes business flows more agile and quickly adaptable to the shifting business conditions.

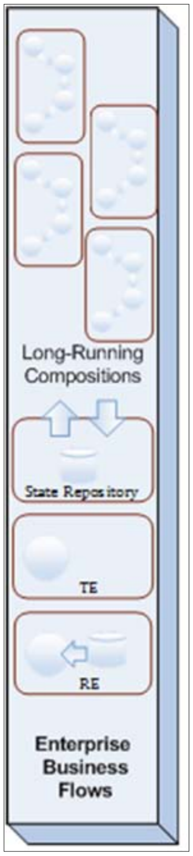
Task-orchestrated services are most commonly composed using languages based on `WS-BPEL` specifications (Business Process Execution Language). Common requirements (such as the implementation techniques, service models, and required functionalities) for the GF04: EBF framework are given in the following following list of requirements:

- Implementation technique:
    - WS-BPEL, Orchestration

- Service models:
    - Task Orchestrated Services
- Required functionalities:
    - Level of WS-BPEL support (versions, extensions) with full orchestration features, which include branching, parallel processing, conditions, looping, scoping, and so on.
    - Correlations and correlations sets, which include native support for long running processes. Native transition for different protocols, that is, `SOAP/HTTP <->JMS`.
    - Implicit Correlation support.
    - Support of standard BPEL Faults, the ability to assign recovery operations dynamically or statically, and fault handling in long-running compositions using compensations.
    - Custom activity implementation and custom variable assignment extensions. BPEL 2.0 extensibility mechanism implementation.
    - Embedded Java support (or any other high-level language).
    - Level of XPath support (XPath versions).
    - SOAP/Message Header support, that is, the ability to reassign the whole header to the new message.
    - Level of XQuery support (XQuery version).
    - Level of REST support in BPEL. Native support REST and SOAP resources (partner links).
    - The `forEach` looping and branching support for the XML nodes with various interactions technique.
    - Assigning data by default to the missing/empty nodes.
    - Supporting transport and message processing metadata (Message Tracking Data or Process context metadata).
    - SBDH support (optionally).
    - Dynamic partner links invocation.
    - Rule-based invocation/mediation. Limitations for MEPs and data transformation.
    - Orchestration engine's capability to clean-up orphan/obsolete data in hibernation store automatically or by schedule.
    - Transformation accelerators, partial validation.
    - Transport protocol accelerators.
    - Possibility to use external transformation engines for complex callouts in transformation.

- Runtime optimization of message size in order to avoid possible memory leak.
- Various compensation flows implementation technique.
- Possibility to dynamically invoke different compensation flows by rules/types of failures.
- Asynchronous Service Broker implementation.
- WSIF support (implementations, extensions).
- Human task workflow support.
- WS-Policy support.
- Availability of High Availability (HA) patterns tested for EBF.
- Level of CEP support (signals, probes, sensors, patterns analyzers, event language, and so on).
- Common J2EE patterns/artifacts support SpringBeans, EJB, and so on.

The figure explaining the previous list is as follows:



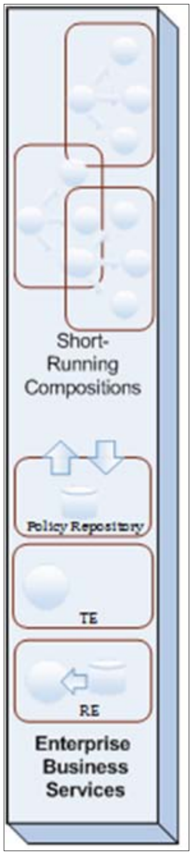
The Enterprise Business Services framework

Business services with high throughput demands, participating in quick running compositions are naturally interconnected in the **Enterprise Business Services** (**EBS**) framework. Let's not get confused by the naming. The GF05: ESB framework in itself provides tools and methods to address the required functionalities, which are listed as follows in terms of implementation techniques and required functionalities:

- Implementation technique:
    - Service Broker as composition controller
    - Mediators
    - Asynchronous message queues
- Required functionality:
    - Atomic Transaction Coordinator implementation.
    - WS-RM support (versions, implementations, and extensions).
    - WS-Addressing support (versions, implementations, and extensions).
    - Native WS-Coordination support (versions, implementations, and extensions).
    - Supported transport protocols.
    - Supported messaging protocols.
    - Supported protocol's sync-async bridging.
    - Supported MEPS (for WSDL1.1 - 4; for WSDL 2.0-8).
    - Concurrent Contract support (contact versioning, by proxy or by other means, such as agents/facades).
    - Supported adapters (including JCA).
    - Process pipes orchestration: looping, branching, termination, and service chaining.
    - Basic security support: encryption, digital signature, and authorization/authentication.
    - Types of data model transformation (XQuery, XSLT).
    - Types of message objects transformation (`XSD<->JSON`, and so on.).
    - Content based routing, dynamic rule-based routing.
    - Types of message validation (complete, partial, nodes, security screening by regular expression/pattern, and so on).
    - Federated heterogeneous ESB support (native links to other vendors using the `WS-*` standard), such as coexistence with Mule or GlassfishESB (ESB grids, snowflakes).

- WS-Policy, WS-SecurityPolicy, WS-PolicyAttachment support.
- Big message volume / throughput native support, such as message throttling, internal component balancing, if any.
- Possibility support sticky sessions (WS-Correlation), in a cluster as well, together or without LBs.
- Scalability and clusters (number of nodes supported, dynamic scaling, dynamic cloud burst).
- JCache (jsr-107) support or other caching support technique for distributed in-memory operations with message's zero-loss tolerance. Number of transactions achieved in reference architecture.
- Available Rile Engines, rule types, ruleflows, RETE support, and available APIs.
- HA patterns tested for ESB.

The figure explaining the previous list is as follows:



Service Bus as a pattern provides structural ways to solve the problems we most probably have while applying these programmatic and methodological tools in order to fulfill the requirements. First, similar to enterprise business flows, we will also deal with composition controllers, managing complex transaction, but in Atomic, Consistent, Isolated, and Durable way (so called ACID), compared to EBF's tolerant way, abbreviated as BASE for Basic Availability, Soft-state, and Eventual consistency. Therefore, the ATC implementation is the one of the top requirements, usually fulfilled by composition controllers and transaction registration services.

Synchronous service brokers together with message mediators are the natural elements of ESB. Similar to the EBF framework, EBS has to consolidate and centralize the business rules, as the magnitude of tasks is similar to asynchronous compositions:

- Rule-based routing and transformations
- Rule-based invocation and computation

It doesn't mean that rule stores must be separate for synchronous and asynchronous frameworks. Decentralization must be conducted on technical requirements: performance and the level of reliability. It is possible to have two (or more) rule engines and one centralized rule store implemented with high availability options.

The same is true for the Policy store and Policy centralization itself, as they are vital parts of EBS and service buses. The point here is that due to its more lightweight nature when compared to EBF, service buses act as service gateways and service perimeter guards, at least the core components of ESB such as Service Brokers do this. Security policies, message mediation and invocation policies, and QoS policies are attached to the service contracts, and most importantly, they are globally enforced through the implementation of policy declaration and policy enforcement points on ESB.

We already mentioned several shared stores required for process-related entities and message metadata. Looking at the broader picture, the physical service implementation requires some sort of logical structure, providing a segregation of the services by types, runtime roles, models, engines required for service executions, failover types, and so on. Actually, we already segregated the Enterprise Business Flow framework for long running services from the fast-spinning Enterprise Service Bus. It is simply inevitable, as technical requirements for platforms holding and running these services are quite opposite, and this fact is obvious enough to start this segregation from the modeling and analysis phase of a service's lifecycle. That's a purely governance matter and will be addressed by a separate framework.

The Enterprise Service Repository / Inventory framework

As service governance is a never-ending process that begins before a service is created and doesn't finish until it is decommissioned, shaping and defining the service inventory should be addressed at the beginning, before other frameworks. Nevertheless, we would prefer to come to this point at the end of frameworks' discussion, when high demands for it become clear and obvious. Yet, this is probably the most misunderstood and obscured framework. Even naming can be confusing, so we need to explain the difference between repositories and inventories first.

The service repository is the centralized store of service-related artifacts and metadata that (possibly) include service code, test results and metrics, and services message attributes. This store is organized as a container with clearly defined metadata taxonomy and service ontology, supporting fast search and lookup. This container is mostly of the design-time nature, but can be actively used on runtime as well; it is usually supplied with elaborative human-readable interface in support of design-time discoverability, allowing artifact harvesting out of exiting implementations and expanding the arbitrary service taxonomy.

The service inventory is the runtime-accessible list of service artifacts mostly related to a service contract that supports fast search and dynamic service invocations. These are supported by machine-readable APIs, which are capable of registering newly deployed services, and search by different elements of service contract (WSDL and tModels).

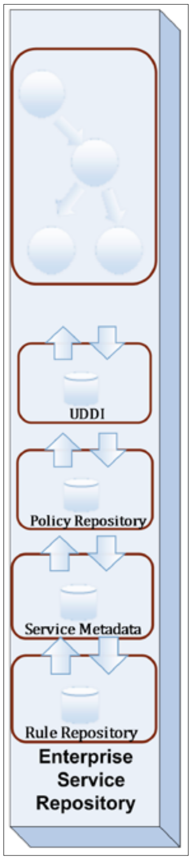
Both support discoverability, that is, runtime and design-time, and therefore should not be implemented separately. One just supports the other. The role of this framework is immense; the UDDI mechanism was declared as an essential part of contemporary SOA from the very beginning. We would risk assuming that initially slow acceptance of SOA was also caused by immature or complex taxonomies of service inventories. Simply put, it's rather hard to invoke and reuse something that is difficult to find or comprehend. At the present moment, most standards related to the service taxonomy are under development and still maturing.

Common frameworks' requirements for **GF06: ESR** are consolidated as follows:

- The Service Repository tool available with links to EBS and EBF
- An Object Harvesting mechanism for existing objects/installations (service/objects discovery in designtime)
- UDDI support for ESR, automatic registration on deployment

- Automatic service discovery in runtime (ESR unified)
- Service templates available for ESR (tModels, or something more human readable)
- BPMN2.0 support for all stages of service development—modeling, analysis, and conversion: `UML->XML->XDD->SRL`

The following figure explains the Common frameworks' requirements for **GF06: ESR**:

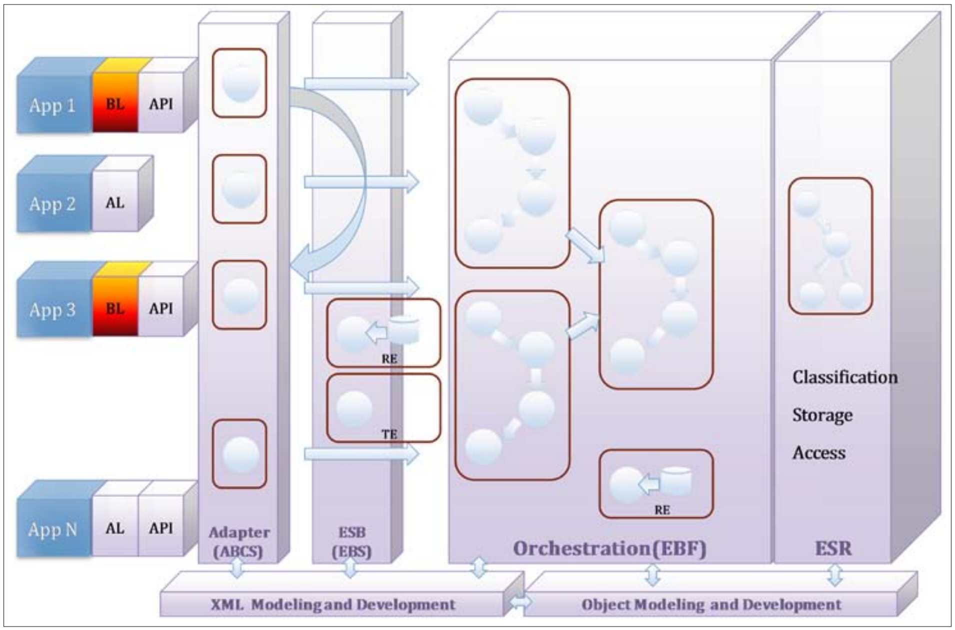


The role of this framework is to position services as an enterprise asset during design time, collect all necessary service metadata and store it in a well-partitioned repository, and provide runtime search capabilities for a dynamic invocation of the service and service-related artifacts. The results of invocation should be properly logged using elements of this framework for further business analysis and usage monitoring. One of the most challenging tasks here is the establishment of the mentioned well-partitioned repository, and we will dedicate a whole chapter for defining its possible taxonomy.

SOA Service Patterns that help to shape a Service inventory

Now is the time to put all the discussed frameworks together and identify the role of patterns in every framework. Six core frameworks are identified, but some more should be explored to complete the picture:

- **Security framework**: Reliable security is a result of diligent measures that are applied to all elements of all frameworks, and it's impossible to say where it is more or less important to have a secure design in place. Still, from the application of principle separation of concerns, a considerable amount of security features are usually dedicated to Secure Perimeter and most commonly implemented as Enterprise Service Bus. So, the security framework has a lot of similarities to the EBS framework.
- **Automated Testing framework**: This requires technical infrastructure elements for supporting continuous integration or other forms of automated build and testing.
- **Automated Deployment**: This framework is strongly related to the previous test automation framework and are usually used together.
- **Governance framework**: This acts as a combination of precepts and standards with relations to processes and roles. This framework is based on proper ESR implementation, but it is much wider than the service's metadata taxonomy.



One could extend this list of framework, adding more fragmentation; however, from any practical point of view, these ten frameworks (that is, 6 + 4) are pretty adequate for dividing the entire technical infrastructure into distinguished and consistent areas of implementation principles, standards, and patterns. However, what are patterns, the main subject of this book?

Through the examples we already explored, we can agree on some of the most frequent problems that are well known to any practicing architect. These problems are common, and so will the solutions be. Thus, commonly approved solutions for regular problems have a common name pattern. The history of patterns in distributed computing is quite similar to the history of the SOA itself. It was a stiff curve from love and passion to hate and disbelief, and there is a very simple explanation for it; that is, the pattern is not a panacea and not an ultimate purpose of your solution design. It would be prudent if from now on you exclude the following questions from your architectural vocabulary:

- What kind of pattern will we apply here?
- Is this design according to the approved pattern catalog?
- Is it pattern or anti-pattern (`http://www.oracle.com/technetwork/topics/entarch/oea-soa-antipatterns-133388.pdf`)?

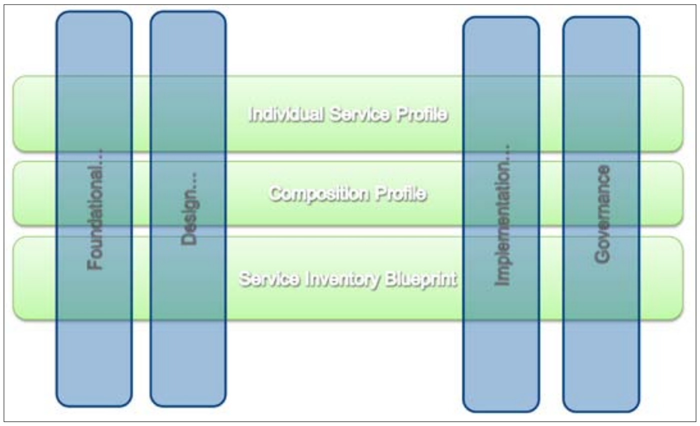
A pattern is a stick for the lame, a remedy for the disease, not the other way around. A clear realization of the problem must be identified first; otherwise, you'll be left looking out for a suitable lame for the stick. This is usually unsuccessful and is followed by the amputation of the healthy application's part, just for making use of the patented remedy from the catalogue with some catchy name. If fact, the patterns' catalogues should be seen as common problems' catalogues. The implementation of the pattern as a working solution could be in jeopardy if:

- You misunderstand the problem on hands, its location (as a framework), root cause, and its relation with the execution environment
- You misinterpret the existing standards, their areas of application, and, as a result, attempt to reinvent it
- You misinterpret the design principles, the balance between them, and the areas of their application

We already gave you a map of the frameworks and standards in relation to the principles, so it's time now to list all the frameworks we mentioned in this chapter and structure them for further discussion, as shown in the following table:

Framework	Standards
Foundational	Functional Decomposition, Enterprise Inventory, and Logic Centralization
Design	Contract Centralization, State Messaging, Messaging Metadata, and Service Broker
Implementation	Agnostic Controller and subcontroller, Compensation Service Transaction, Composition Autonomy, Inventory Endpoint, Partial State Deferral, Legacy Wrapper, and File Gateway
Governance	Service Decomposition, Service Refactoring, and Metadata Centralization

The following figure explains the preceding table:



All these patterns will be discussed during the implementation of three already mentioned main SOA compound patterns: Enterprise Service Bus, Orchestration, and Federated Endpoint Layer.

Summary

Services are the functionally consistent atomic units of work with technical boundaries, providing the required level of autonomy where principles of the service orientation are applied in order to maintain characteristics that are essential for achieving the goals of service orientation.

Service orientation is the architectural approach that is based on the recognition of a service as a unified business block.

One of the key roles of the independent standards (`WS-*` specifications) is to ensure that service-oriented solutions based on these standards stay truly vendor-neutral.

The SOA framework is a structured and technically independent area where design principles and standards can be repeatedly applied together in a measured balance during various stages of analysis, modeling, development, implementation, testing, and governing in order to achieve the desired technical characteristics. As some `WS-*` specifications can be seen as a framework as well, most of the SOA frameworks have a compound nature.

SOA patterns are commonly accepted and approved solutions to repeatable and recognizable problems usually occurring in different frameworks, while implementing combinations of standards and principles.

2

An Introduction to Oracle Fusion – a Solid Foundation for Service Inventory

From this chapter onward, all discussions and examples will be based on the Oracle platform, products, and methodologies. In this chapter, we will discuss Oracle's technical implementation of abstract frameworks, as proposed in the first chapter. We will demonstrate that Oracle's technology stack can practically cover all the aspects of the service-oriented application's lifecycle, from development and implementation to monitoring and error handling. The degree of coverage essentially depends on the understanding and proper balancing of the SOA principles and the level of standards supported by Oracle's platform. Even though it has great coverage of service-oriented approaches and all SOA benefits, Oracle's platform also has certain limitations. These limitations will be discussed in the following chapters, which are dedicated to the patterns' realization in specific frameworks. The logical outcome of this chapter is the framework's requirements tables, containing all of Oracle's specifications.

The Oracle SOA technology platform

We have already defined ten SOA frameworks, of which six essential ones are quite SOA-specific. We will follow this notion until the last page, but we are not too eager to defend this logical segregation and will not force you to accept it blindly. If you already have a working framework model based on **TOGAF** (`http://www.opengroup.org/togaf/`) or **ITIL** (`http://www.itil-officialsite.com/`), that's perfectly fine. We chose this framework model for three reasons. First, this simplified model with concise but distinctive layers is quite well-accepted by both groups of practitioners—developers and operational personnel.

Second, and most importantly, this model is derived from Oracle's service implementation methodology (discussed later in the chapter). The third reason is purely empirical and based on Oracle's history of acquisitions and integration of different components into a solid application portfolio.

The Oracle SOA development roadmap – past, present, and future

Nowadays, Oracle offers probably the most complete family of products. This family has two distinctive SOA characteristics—each member of the family is the "best-of-breed" in its class (read: framework) and each product is hot-pluggable. Thus, the portfolio itself follows the principles of Composability, and because it has been shaped mostly through the chain of acquisitions, it's genuinely vendor neutral.

This type of SOA packaging would not be possible if Oracle didn't follow the service orientation principles. This fact alone makes it quite attractive for many industries and enterprises. For example, you would naturally choose a doctor who is capable of taking his own remedies. It is also quite natural that this status quo was not always this happy. We all have good and bad times during different phases of our SOA endeavors (`http://oracle.com.edgesuite.net/timeline/oracle/`). Please refer to the following table:

Oracle products timeline				**Industry standards timeline**	
Pre-SOA development					
Year	**Product(s)**	**Type**	**Description**	**Standard**	**Agency**
1979	Oracle v 2	DB	First commercial version		
1985	Oracle v 5	DB	Implemented in a client-server model		
1988	Oracle v 6	DB	PL/SQL engine added to DB		
1992	Oracle v 7	DB	Stored procedures and triggers and most mature traditional RDBMS	ISO 9075, Entry Level SQL92 Standard	ANSI
1997	Oracle v 8	DB	RDBMS with SQL object orientation, AQ introduced (persistent messaging)		

Oracle products timeline				**Industry standards timeline**	
			Pre-SOA development		
Year	**Product(s)**	**Type**	**Description**	**Standard**	**Agency**
1998	Oracle v 8*i*	DB	Integration with JVM (here *i* stands for the Internet)	XML	W3C
1998	JDeveloper 1.0	IDE	The first IDE release (based on Borland code)	SOAP	W3C
Building Contemporary Oracle SOA					
1999	XML Development Kit	XDK	XML parser, renderer, validator, and so on	XSLT, XPath	W3C
2000	Oracle OAS	OAS	Oracle Application Server	WSDL	W3C
2000	Oracle iFS	FSO	Oracle Internet File System (iFS)	UDDI	OASIS
2001	OC4J	OAS	Oracle Java container implemented with Orion acquisition and standalone J2EE application server	**Java Connector architecture** (**JCA**) and **Java message service** (**JMS**)	JCP
2002	TopLink	JPA	**Object-relational mapping** (**ORM**)	SAML	OASIS
2003	Secure Remote Access	FSO	Oracle acquired FileFish, which becomes part of Oracle's Collaboration Product Suite, aggregating remote **file system objects** (**FSO**)	WS-Reliable Messaging	OASIS
2004	Oracle v 10*g*	DB	First Oracle grid-oriented DB (*g*-grid)	**Service Data Objects** (**SDO**)	OASIS
2004	Oracle Identity Management	IAM	Oracle acquired Phaos, Identity management solution provider	WS-Security	OASIS
2004	BPEL Process manager	BPEL	Oracle acquired Collaxa, first release Oracle BPEL	WS-BPEL	OASIS

Oracle products timeline				Industry standards timeline	
Pre-SOA development					
Year	Product(s)	Type	Description	Standard	Agency
2005	**Business Activity Monitor** (**BAM**)	BAM	Oracle acquired PeopleSoft, BAM is one of the products in the stack	WS-CDL	W3C
2006	Oracle Data Integrator (ODI)	ELT	Oracle acquired Sunopsis, Extract Load Transform tool for data integration and master data management	WS-Addressing	W3C
2007	Enterprise Lifecycle Management	ESR	Oracle acquired Agile, Integrated Enterprise Product Lifecycle Management provider	XQuery	W3C
2007	Oracle Adaptive Access Manager	IAM	Oracle acquired Bharosa, provider of fraud prevention and authentication solutions (OAAM) and integrated IAM	WS-HumanTask	OASIS
2007	Oracle Role Manager	IAM	Oracle acquired Bridgestream, with Role Manager product, managing Role Based Access (RBA) integrated with IAM	WS-Policy	W3C
2007	Oracle Application Integration Architecture	AIA	Oracle integration approach for establishing seamless collaboration between numerous fusion applications—OEBS, Siebel, JD Edwards, and so on	ebXML Messaging Services 3.0	OASIS

Oracle products timeline				**Industry standards timeline**	
Pre-SOA development					
Year	**Product(s)**	**Type**	**Description**	**Standard**	**Agency**
2007	Oracle Coherence	XTP	Oracle acquired Tangosol, implementing reliable in-memory data grid technology, known as **Extreme Transaction Processing** (**XTP**)	WS-BPEL 2.0 WS-Context 1.0	OASIS
Maturing Oracle Fusion Middleware					
2008	Oracle ESB (OSB) and other tools	OFM	Oracle acquired BEA, moving to the more reliable JVM, App Server, Service Bus, and Repository	**Solution Deployment Descriptor** (**SDD**)	OASIS
2009	Oracle hosts JEE	OFM	Oracle acquired Sun Microsystems with a wide range of Sun's Middleware products (GlassFish, OpenESB, OpenSSO) and complete Java stack	Basic Security Profile v 1.1	WS-I
2009	OFM 11*g*	OFM	Complete SOA products stack running on Oracle WebLogic Server 11*g*	WS-Federation 1.2, WS-Discovery 1.1	OASIS
2010	Oracle SOA Management Pack	OEM	Oracle acquired AmberPoint with fully centralized management console (OEM), integrating BPEL console and BAM	**Extensible Resource Identifier** (**XRI**)	OASIS
2011	Oracle Service Gateway	ESB	Oracle partnership with Vordel, provides service security and API management solutions	Standard for Online Privacy, first draft	W3C
2013	OFM 11*g* 11.1.1.7 (PS6)	Only the journey is written, not the destination			

> The previous table does not pursue the presentation of the complete historical perspective of Oracle's acquisitions and development milestones, but merely links the key middleware products with the history of SOA's `WS-*` standards implementation by the most influential standardization agencies. For recent acquisitions in the Oracle Middleware products family, please visit `http://www.oracle.com/us/corporate/acquisitions/index.html` and refer to the *Middleware* section.

Middleware products were not the primary line of business for the company, which was initially prominent for its DB products. For almost two decades since the early eighties, Oracle was a **relational database management system** (**RDBMS**) flagship company, providing reliable relational DB and data processing commercial products literally for all industries. Again, mainly through the chain of acquisitions, practically all enterprise domains, such as ERP, CRM, SCM, and HCM, were covered by DB-centric applications: Oracle E-Business Suite, PeopleSoft, JD Edwards, Primavera, and Siebel. Business domains are truly enterprise wide, shown as follows:

- **Financial management**: This is useful in General Ledger, Accounts Payable and Receivable, Asset Management, and so on
- **Human capital management**: This is useful in Global Human Resources, Global Payroll, Benefits and Performance Management, and so on
- **Supply chain management**: This is useful in Inventory, Shipping and Receiving, Distributed Order Orchestration, Cost Management, and so on
- **Project portfolio management**: This is useful in Project Costing, Control and Billing, Performance Reporting, and so on
- **Procurement**: This is useful in Purchasing, Sourcing, Supplier Management, Spend and Performance Analysis, and so on
- **Customer relationship management**: This is useful in Customer Master, Sales, Marketing, Quota Management, Social Media, Portals, and so on

Common to all these suites are the Access Controls, Configuration Controls and KPIs, BI Dashboards, and Extendibility Foundation packs.

This massive application stack has been building gradually over the years, accommodating requirements from all industries, such as Telecommunication, Healthcare, Logistics and Transportation, and Governance, among others. Oracle's own DB-based flagship product, E-Business Suite, accommodated 1,000 lines of PL/SQL code in every business module, not mentioning C and Java.

The commercial value of all these products is out of the question. In 2001, Larry Ellison announced the following:

> *Oracle saved $1 billion by implementing and using its own business applications.*

Apparently, that was before SOA, and the whole stack was not that massive in scale. With more and more applications in the portfolio, seamless product integration becomes quite challenging, not only within an individual enterprise, but also within the cross-enterprise.

> Today, Oracle's application stack (`http://www.oracle.com/us/products/applications/overview/index.html`) consists of more than 200 hot-pluggable applications; this is probably the biggest commercial application farm.

Shifting from the client-server to the multitier infrastructure was quite evident with the dedication of the middleware tier as an integration medium. Oracle proactively worked on the accommodation of the middleware strategy before introducing SOA as a methodological approach. **Advanced queuing** (**AQ**) was implemented as a reliable message delivery mechanism and embedding JVM into DB opened the door for more interoperability options. New JEE middleware products were launched in order to improve interoperability, such as the following:

- **Oracle Application Server** (**OAS**): This was not the best application server at the time or fully compliant with many J2EE standards, and it suffered from memory leaks and poor performance. This was followed with **Oracle Integration Server** (**OIS**), but it didn't improve the situation much.
- **Oracle InterConnect**: This was the predecessor of the first Oracle Service Bus. Even this was not considered a total success, but it shaped the whole concept of ESB (of course, for Oracle) and demonstrated the necessity of the adapter layer (Interconnect Technology Adapters).

Soon it became quite obvious that the main problem with integration is integration itself. With a considerable number of disparate components, each establishing interoperability, putting all efforts into a single integration layer was similar to the search for the philosopher's stone. Hot-pluggability cannot be contracted by means of super glue. Some efforts should be put into the applications (services), such as Infrastructure, Methodology, and Governance, in order to make seamless interoperability more effortless, and needless to say that all these efforts must adhere to the already discussed principles and standards to make the effect profound.

Oracle has been taking a very active role in development and putting these standards into practice as a member of W3C since August 1995. Oracle has been a sponsor of the OASIS Web Services' technical committees since 2002 (and members of the board of directors since 2003). Oracle participated in practically all technical committees for all core SOA standards such as WS-ReliableMessaging, WS-Addressing, WS-BPEL, and so on. The previous timeline shows a strong correlation of product releases with standards acceptance and approval. Today, about 200 open standards and industry specifications are adopted in Oracle's SOA-related products. Surely, for such a huge company with such a comprehensive portfolio, changes and adaptations weren't as fast as some would expect with open source products (from the Apache community). In addition (and this would be the primary reason), the clients' reception of Oracle's initiatives was sometimes inadequate due to the products' roles and SOA principles being misunderstanding in general.

Probably one of the most noticeable examples would be the launch of the first release of Oracle BPEL Manager (originally from Collaxa). The Service Orchestration Engine, primarily nominated for its asynchronous services, was mistaken by many as the synchronous Service Bus. How many complaints about its synchronous MEPs performance did we witness (and produce, to be honest) at the time? All the complaints were because the Orchestration pattern was not clearly distinguished from the ESB pattern in various implementations. Public demand for a clear service collaboration strategy was quite noticeable, and after several years of initially combined SOA and integration efforts, Oracle responded with Fusion Strategy Roadmap (2006), where Fusion Middleware was declared as an SOA Enabler. In this sense, the term Fusion indicates a balanced approach, where the main focus is on SOA interoperability between components with the elements of integration, where service orientation is not feasible or too burdensome. Oracle SOA Suite 10*g* as a part of Fusion Middleware was introduced as a milestone of the Fusion roadmap including the following products:

- Oracle Service Bus, the first full-fledged Oracle ESB after InterConnect
- Oracle BPEL Manager with an extremely extensive Adapter framework
- Oracle Web Service Manager, with agents and gateways for secure policy-based message interchange
- Oracle Rule Engine, which supports various types of rules with rule SDK

The previously listed products are just a few of those available in the Fusion bundle. Oracle had much more to offer, and in general, these products met expectations from SOA architects and developers. The Fusion strategy started to pay off. The next steps according to the Fusion strategy were as follows:

- To improve application server reliability and performance as Oracle iAS10*g*, at the time, was already an obsolete OC4J-based server
- The Business Activity Monitoring tool was quite detached from the entire Fusion stack and based on a separate technology platform
- Rule Engine proved to be quite fast, but the rules' authoring and management was rather unfriendly, and that jeopardized its acceptance
- Oracle JDeveloper 10.1.X was noticeably slow compared to other IDEs and was not fully integrated with all SOA Suite components (unfortunately, jumping ahead, we can say that this is still the truth)
- In addition, Oracle's own Service Bus could not keep up with the closest competitors for some aspects

Oracle addressed these issues quite radically, acquiring BEA. The next generation of Fusion Middleware, 11*g* (2009), came equipped with AquaLogic Service Bus (OSB), BPM Studio, Enterprise Service Repository, and most importantly, **WebLogic Application Server** (**WLS**), one of the best J2EE servers available; all of these are based on the very reliable JVM. It took almost two years to reassemble and repackage all the new middleware components with the new WLS Application Server as the foundation and other BEA components. JDeveloper 11*g* proved to be more reliable and more integrated with most core frameworks, such as DB and SOA Suite, but it still did not cover the ESB development lifecycle. The standard of **Service Component Architecture** (**SCA**) was adopted in SOA Suite, where BPEL's role was rearranged and three other equally important components were added. Together with SOA Suite, three other suites were offered for **Event Driven Architecture** (**EDA**), BPM, and Governance, with overlapping functionality. Fusion's hot-pluggability option allows customers to select only the functionality that they really need at the moment, reducing resource wastage.

After the first radical turn, the next huge acquirement of Sun Microsystems was a completely logical thing in shaping the Fusion strategy. Now Oracle had the true foundation of all Java resources—the Java language itself. In addition, the Oracle family inherited Sun's Java-based, nonstrategic, but very attractive, products—Glassfish Application Server (`https://blogs.oracle.com/theaquarium/entry/java_ee_and_glassfish_server`; Oracle GlassFish Server will not be releasing a 4.*x* commercial version), OpenESB, and NetBeans IDE.

Despite their nonstrategic status, these products are quite capable of playing their parts in the construction of an SOA infrastructure of any size, and are quite cost effective as well. All these products are supported by open communities (`http://www.open-esb.net/index.php?option=com_content&view=article&id=90&Itemid=469`) and in some cases, are more advanced than the Fusion middleware stack (for instance, Glassfish is the first Java EE 7 Server). Anyway, from the customers' point of view, it's quite positive to have multiple options such as the three available ESBs. Out of the three, we have discussed OSB and OpenESB (refer to `http://www.open-esb.net/`, the Sun Microsystems and Seebeyond products, for more information on OpenESB). We will soon come to the third ESB.

Another strategic benefit of this acquisition is the availability of advanced hardware, capable of hosting the preconfigured complex Fusion solutions (both DB and MW). Clustering highly reusable Canonical software components logically leads to the implementation of the scalable Canonical Resource (hardware, infrastructure) pattern. Oracle responded to that with the Exadata and Exalogic combined solutions based on Sun machines (engineered systems) running on Oracle Linux. There are some debates regarding the elasticity of the *HW* + *SW* bundled approach and how it suits the public cloud. Clearly, this **platform as a service** (**PaaS**) solution is optimal for private and hybrid clouds, providing horizontal scaling, dynamic resource provisioning (with some limitations), and cloud burst-in and burst-out. Some of these challenges can be solved by inner-cloud load balancing and functional decomposition between the clouds. The fourth most critical cloud characteristic is **Multitenant Access**, which requires complex security measures and is also addressed in Oracle's Fusion roadmap.

In 2011, Oracle announced its strategic partnership with Vordel (now part of Axway), and the Oracle Enterprise Service Gateway product launched as a rebranded Vordel API Gateway. Today, it's also called Oracle API Gateway. Nevertheless, with the best-of-breed Secure Gateway, Oracle finally introduced a comprehensive security layer essential for any cloud model. As with most Service Gateways, the API Gateway is essentially an ESB, consolidating all common features for the service bus SOA patterns. Thus, now we have three full-fledged ESBs in the Oracle technology stack and something to choose from.

Cloud patterns and technologies are not the primary aims of this book, but to add some silver lining to emerging clouds, we could mention some of Oracle's other initiatives, again expressed via acquisitions, addressing both SOA and the cloud. For example, the quite recent (2013) acquisition of Nimbula—provider of the private cloud infrastructure management software; capable of managing the infrastructure resources of services delivery, quality, and availability; as well as workloads in private and hybrid cloud environments. It seems to be a perfect addition to the Oracle cloud-based Sun hardware resources.

It would not be entirely correct to state that all Oracle Fusion roadmap's milestones were based on acquisitions. Oracle primarily adopted **JavaServer Faces** (**JSF**) with lots of other JEE standards and patterns and came up with **Application Development Framework** (**ADF**), a foundation for all Web 2.0 and Enterprise 2.0 Oracle products mentioned as follows:

- WebCenter, a web-based collaboration suite, replacing Oracle forms, consolidating portals, social media management, and content management
- BAM Studio, proactive service monitoring, and dashboard creation
- BI Suite, analytics, reporting, OLAP, and scorecard management
- Enterprise Manager consoles

Needless to say that with the acquisition of Sun, ADF Java-based development is gaining a new boost.

So that was just a quick glance at some SOA and Oracle combined milestones, but history is not over yet. Distributed computing is setting new challenges, which we could address by the proper application of tools, principles and methodologies, and combinations of patterns. Oracle has nothing to prove really. It's obvious from the history line that only two options are available: Silo and SOA. Being the main consumer of its own Fusion technology, Oracle set the direction for cost-effective application collaboration based on the SOA principles and standards, moving from a Silo-based approach. Let's now see how these concrete tools can be fitted into Oracle's vision of a standard technical infrastructure.

Oracle SOA frameworks and technology layers

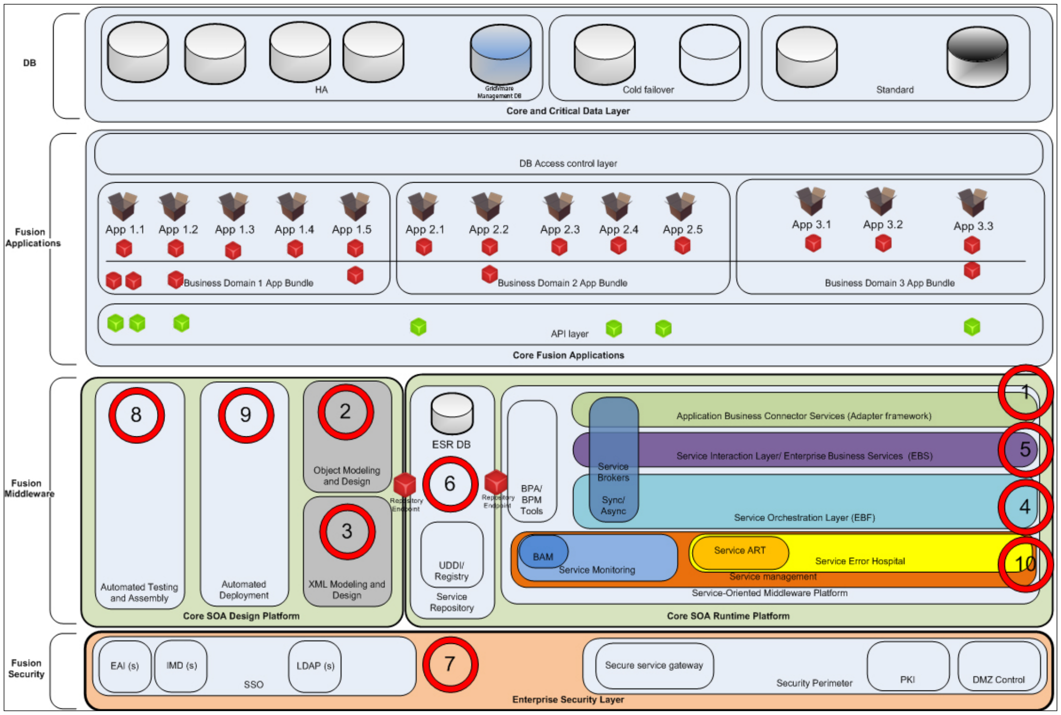
In *Chapter 1*, *SOA Ecosystem – Interconnected Principle, Patterns, and Frameworks*, we have already outlined the abstract layers related to the technical infrastructure, divided into 10 frameworks. At least six frameworks have concrete realization in four major technical layers. We will discuss the technical layers first, starting traditionally from the top, as demonstrated in the next figure.

The Data layer usually presented by data clusters is grouped into three types of **high availability** (**HA**) implementations such as zero data losses with automated failover, manual failover, and a single instance with a cold standby, suitable for manual recovery. The last one is not HA at all, but is still quite suitable for business applications with non-OLTP requirements and capable of staying offline during data recovery routines from backups. Only complete HA solutions are suitable to serve the orchestrated service compositions with state deferral requirements. Services with DB resources underneath cannot be reliable composition members if DB is not covered by HA capabilities.

In some cases, it can be addressed by the implementation of reliable messaging with asynchronous queuing, but that's not an option for fast-running compositions or service-oriented middleware itself, where the role of DB is crucial. Middleware DB, as we already mentioned, is a host of the Task Services state repository, operation monitoring reporting, service repository and registry, and error recovery. More than 20 different schemas will be installed in DB during the OFM installation, responsible for the various aspects of middleware runtime activities. Simply put, the Entity and Task services cannot be implemented without a reliable DB.

Interestingly, these days, DB itself becomes more and more Fusion-like. According to a statement made by Larry Ellison in October 2012, it was promised that the upcoming 12*c* Version would implement a new concept of a hot-pluggable DB, allowing multiple DBs to run under one copy of Oracle DB.

The following figure shows Oracle's Reference Architecture:



According to Oracle's Reference Architecture (`http://www.oracle.com/technetwork/topics/entarch/oracle-pg-soa-governance-fmwrk-r3-0-176707.pdf`), the second layer is an application infrastructure, similar to the DB layer divided by application bundles into several application domains (also covered by HA options).

From a technology standpoint, it is a pure application server implementation (on WebLogic or any other JEE compliant server) with all the required elements of the server infrastructure such as Node Managers, Admin Server, and Managed Servers in separate nodes. Three distinguishable sublayers can be mentioned (following the MVC pattern) as follows:

- Core application logic presented by the traditional Java modules (EJB, POJO, and so on)
- DB Persistence layer responsible for DB access, object-relational mapping for the relational DB, or object extraction for the object-related NonSQL DB
- The API and representation layer responsible for the exposure of the application interfaces

Application modules and APIs within an application package are presented as red and green boxes, respectively.

The Fusion Middleware layer is also based on the WLS Application Server and probably should be placed in between the DB and App layers, according to its name; however, we deliberately put it close to the southbound end of the Fusion perimeter, binding it with the security layer. This layer hosts most of the SOA frameworks, which in turn compose most of the patterns discussed later. Now, all abstract frameworks, discussed in *Chapter 1*, *SOA Ecosystem – Interconnected Principle, Patterns, and Frameworks*, are mapped to this layer and numbered. The layer itself can be vertically divided into Runtime and Design areas. Note that this division is rather arbitrary as ESR and Governance frameworks are actively used in both areas.
The runtime area holds four distinctive frameworks, namely:

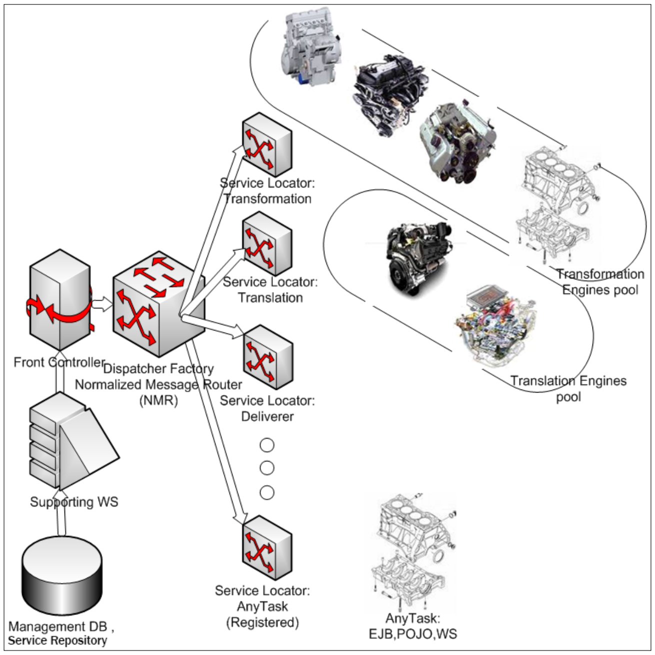
- Adapters (1)
- Service Bus (5)
- Orchestration (4)
- Service management (runtime part)

The Service Brokers, as agnostic controllers and subcontrollers, are the essential patterns acting across these frameworks. So, it's impossible to say which runtime framework in particular is used for their physical realization. **Business Activity Monitoring** (**BAM**) as a tool is a part of the broader Service Monitoring solution, incorporated into the Service Management (that is, Governance) framework; together they form a business and technical service monitoring solution. This solution, among other functions, feeds the error-handling facility with the information needed for service recovery, both manual and automated. The **Automated Recovery Tool** (**ART**) is a part of the error handler specific for particular service inventory, but based on Oracle's common error hospital pattern, which is essentially the implementation of the agnostic service controller SOA pattern linked to the Enterprise Service Inventory.

The Service Inventory framework during its runtime provides the discoverability of services and the service artifact to all runtime frameworks via unified API, constructed according to the Service Inventory (Repository) Endpoint pattern. Any service, service agent, or composition controller in these frameworks could perform the lookup and discovery of any service metadata element according to the **Service Repository** (**SR**) taxonomy.

It is highly important to realize the role of the Fusion technology layer for these runtime frameworks. This layer, based on WLS and JVM(s) (it's also true for the APP and DB layers) provides all the necessary **Service Engines** (**SEs**) essential for the functioning of all runtime frameworks and the hosting of our business-services logic.

Service Engines are the services that serve our services, so to say. As we can expect, BPEL as a language will be interpreted and executed somehow; the same for ESB, where proxy services must be decoupled from business services and the VETRO logic between them must somehow be fulfilled. Actually, the transformation, translation, and rule verification do not belong to a single framework and are provided independently to any layer that needs them. Moreover, their realization comes in the form of an engine, that is, how we call them—for instance, the Rule Engine. SE can be implemented in several ways, but the common requirements are portability, simplified binding of the engine and the service component, and the unified messaging model.


Service Locator: Transformation
Service Locator: Translation
Transformation Engines pool
Front Controller
Dispatcher Factory
Normalized Message Router
(NMR)
Service Locator: Deliverer
Translation Engines pool
Supporting WS
Service Locator: AnyTask (Registered)
AnyTask: EJB,POJO,WS
Management DB ,
Service Repository

Service Engines Runtime environment

The SE as a runtime environment for service execution and message exchange can be based on the **Java Business Integration** (**JBI**) standard (JSR-208). Oracle inherited the JBI specification with the aquisition of Sun. In fact, JBI is a SCA enabler as it permits the SCA components to be realized through a technology-agnostic generic programming model that decouples the components' implementation from their communication, allowing a high level of reuse. JBI SEs communicate with each other at the technology level, leaving business communications to the services. This communication model is defined as a WSDL contract that we mentioned earlier. All communications are decoupled by a message dispatcher, called the **normalized message router** (**NMR**), which physically support all MEPs, as declared in WSDL.

Ideally, we should be able to choose any engine for our abstract operation (that is, transformation or BPEL interpretation) and use a certain engine in any part of our frameworks. In fact, the Rule Engine and its SDK are unified across the Oracle Service Bus, orchestration, and even DB. (It is worth mentioning here that the Glassfish application server and OpenESB support JBI.)

From a service implementation prospective, architects should realize that the service models (such as entity, utility, or task) define the needs of certain engines. Therefore, the task-orchestrated services are rather special, as they strongly require the BPEL engine along with the DB infrastructure, as compared to the simple utility service, which can be implemented as a lightweight portable jar. Service layering within Service Inventory must be done with extra caution to avoid performance overhead and excessive infrastructure costs.

The design time part of Fusion Middleware hosts the rest of the compulsory frameworks such as the XML and Object Development frameworks and Automated Testing and Deployments frameworks. They are covered by several Oracle tools, linked to the source control and the Enterprise Repository. The two main developer tools are JDeveloper and Eclipse with the Oracle pack for OSB.

The security layer is a form of ESB as it basically serves similar features such as message screening (a form of validation), transformation, and exposure-only services designated for the outside world (that is, hiding and abstracting the internal Service Inventory). So, can it be implemented using Oracle Service Bus? Hmmm…yes! Firstly, OSB is integrated with **Oracle Web Service Manager** (**OWSM**). OWSM allows you to define policies, store them in the policy store, and attach them to a particular service. OSB can support message encryption and digital signature, the two main mechanisms for establishing message confidentiality, nonrepudiation, and integrity. Integrity guarantees that a message has not been altered on transit, confidentially ensures that only authorized people or processes have access to the message's content, and nonrepudiation signifies that the act of message transmission cannot be denied by the sender.

OWSM supports SAML, so brokered authentication and an Authorization pattern can be supported as well. It all sounds good. However, there are some other requirements that should be taken into consideration, shown as follows:

- OSB is an SE on top of WLS with connection to the DB (in the full version). With so many moving parts (*OS* + *JVM* + *WLS* + *OSB* + *OWSM* and DB somewhere nearby), will you consider moving it to the DMZ, as security perimeters should be in front of the firewall, not behind it?
- The conventional XML parsers and validators do the XML validation. They are quite well-known and have been exposed to very thorough scrutiny with not always good intentions. Naturally, the XDK development is mostly focused on functionality first, good performance after that and, honestly, security is not given highest priority as the performance's natural enemy. Would you consider putting a functionally brilliant but potentially insecure XML validator as your northbound XML Gateway?
- Talking about performance, we have to admit that conventional validators and transformation engines are not the best players. A secure perimeter should be considered as a corporate asset, common to all projects and products. It's not that uncommon to have about 10 K tps with a 5 K SOAP message per single node (VM 2CPU 8 GB).

Yet another architectural approach should be evaluated. The security perimeter is an ESB with all common features as we mentioned before. It can perform service brokering, mediation, and protocol bridging, both for message and transport protocols, and can also apply corporate policies (with security in mind first). Thus, for the external services, the presence of this framework makes the conventional **Enterprise Service Bus** (**ESB**) handling the **Enterprise Business Services** (**EBS**) layer quite superfluous, especially from a performance point of view. At the same time, services with outbound-only MEPs can reside in the conventional EBS layer and employ security features of OSB, whether it's possible from a functional and/or performance point of view.

Functionality specific to the **security perimeter** (**SP**) ESB is expressed by the following SOA security patterns, specific to a service message in transit for transport and message-based security:

- **Message screening**: We must prevent the infiltration of insecure message content through SP. One of the measures addressing it is XSD-based validation, which could be ineffective in the case of a conventional XML processor.

- **Exception shielding**: We must prevent the exposure of a service's error stack trace to the outside world. SP is not the optimal place for this, as the service itself must be designed to prevent this kind of leakage. It's our last resort, but functionality must be in place.
- **Full triple-A support**: Authentication, Authorization, and Accounting (triple-A support) should be based on the WS-Security specifications, combining a dozen `WS-*` standards, including digital signature (**DSS**; `https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=dss`), **encryption**, **portable trust**, **secure conversation**, and so on. In contrast to the XML engines are the security-related libraries for signature and encryption practicing the policy that can be expressed as the security algorithms being widely open with the key(s) strictly undisclosed (of course, except the public key). This is one of the ways that we can ensure that our security algorithms are secure enough.

All of these security requirements, and many more on top of that, are covered by the Oracle API Gateway. This concludes the overview of Oracle's Reference Architecture (`http://www.oracle.com/technetwork/topics/entarch/oracle-ra-soa-foundation-r3-1-176715.pdf`) layers and related tools. Now we will proceed with each tool individually, starting from the Fusion foundation.

Oracle SOA Foundation – methodology

> *"It's like if you want to buy a car. Would you get an engine from BMW, a chassis from Jaguar, windshield wipers from Ford? No, of course not. Right now with the software that's out there, you need a glue gun – or hire all these consultants to put it together. They call it best of breed. I call it a mess. We want to put an end to that."*
>
> *– Larry Ellison*

This emotional but hesitation-free statement made in the year 2000 pointed out the Oracle CEO's core procedural beliefs for the next few years as follows:

- Packaging the best-of-breed components without standardization is a dead end. You will run out of glue. The sole term *best-of-breed* without any criterion (read: principles) describes why it's actually the best and is clarity of meaning.

- Components standardization is impossible without rewriting the components in order to make it compliant with the SOA principles specifically responsible for the service/component design (such as autonomy, statefulness, abstraction, and reusability). However, this is bad news for those who believe that OSB and BPEL alone will solve everything.

Oracle had to establish a solid methodology in order to plug various Fusion applications together and give customers an option regarding what to choose from for their primary business components. The regular choice would be the ERP Oracle products from the Fusion application portfolio. But methodology, based on SOA principles, would be suitable to establish the collaboration between different products as long as their architecture is based on the same principles. In other words, you do not need to purchase OEBS, Siebel, PeopleSoft, and so on in order to follow the methodology proposed by Oracle. Most importantly, you do not need to buy methodology-related frameworks to practice it if you do not have a single Fusion application in your portfolio. This architectural approach, called **Application Integration Architecture** (**AIA**), is a perfect example of how to employ SOA principles and patterns grouped in clearly defined frameworks to work toward the integration of disparate applications.

Wait a minute, you might say. Earlier, we stated that SOA contrasts integration, and integration is the beast we would like to get rid of. Is there some kind of contradiction here? Not really. Firstly, most Fusion applications (and not only Fusion) emerged and were acquired well before the realization of the SOA approaches. All these approaches are evolutionary, so the transition from classical integration to SOA pluggability was gradual and the applications have been rewritten many times. **SOA Big Bang** is the last thing we need in our business, and thus traditional integration still has its share in this framework.

All these integration step-backs are clearly covered by recognizable (and therefore reusable) SOA patterns. Would these patterns be a good example for companies with the heavy burden of legacy applications, which are not exactly Oracle products-based? And last but not least, AIA provides a defined and practical way to implement SOA frameworks which are not really bound to Oracle's products. Ask yourself, how many times have you been arguing about what would be the practical realization of the TOGAF model (or any other model such as Zachman, P**F, and ITIL/ITSM)? For instance, why is the data in TOGAF's **Information System Architecture** (**ISA**) detached from technology and business? Enough is enough; here is an example, and it works!

AIA provides a clear decision tree on what approach, SOA or EAI, you should undertake depending on your requirements, where the key requirement is transformation. Technically, AIA is packaged as follows:

- **OFM products as a backbone**: SOA Suite, OSB, BPM Suite, SOA Registry and Service Management, and ODI for heavy batch processing
- **AIA Foundation Pack (FP) as a framework reference**: Reference process models, Common Objects, and Lifecycle and Governance
- **Pre-built Integration Packs (PIPs)**: Practical realization of ready-to-configure task-orchestrated services (business processes) for Oracle Fusion Applications

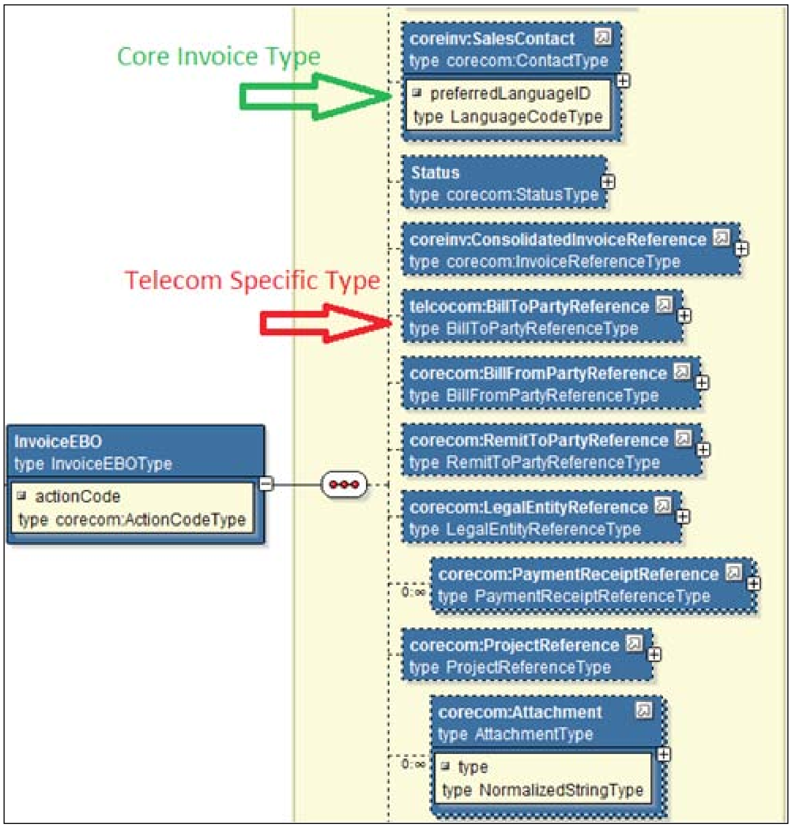
For companies with several Fusion applications installed, PIPs are a really interesting part of AIA as they provide fine-tuned, business optimal composite services such as Order to Cash, Agent Assisted Billing Care, and so on. You can learn a lot from them about how to implement all the SOA patterns. What if your version of an API is different and/or the business flow has its own specific versions? AIA's FP provides clear guidance on implementation as follows:

- Modify existing WSDL or create and register a new one
- Enhance **Enterprise Business Object** (**EBO**), a canonical data model representation
- Set **Enterprise Business Message** (**EBM**) according to the corresponding EBO
- Implement required transformations to accept/provide Application Business Message according to the API
- Alter PIP to add/remove elements of business logic

Oracle provided PIPs for several business domains and industries. The first industry that received a comprehensive set of task-orchestrated services was the Telecommunications industry. There are some extensions for utilities and insurance as well. You probably noticed from the very beginning that we used the AIA terminology to actively describe frameworks and SOA artifacts, and we will continue doing so. Therefore, we will illustrate the core components in a bit more detail in the following sections.

Enterprise Business Object

AIA provides standard canonical data models known as **Enterprise Business Object** (**EBO**). It is a standard business data object definition and is a reusable data component as shown in the following figure:


Core Invoice Type
Telecom Specific Type
InvoiceEBO
type InvoiceEBOType
actionCode
type corecom:ActionCodeType
coreinv:SalesContact
type corecom:ContactType
preferredLanguageID
type LanguageCodeType
Status
type corecom:StatusType
coreinv:ConsolidatedInvoiceReference
type corecom:InvoiceReferenceType
telcocom:BillToPartyReference
type BillToPartyReferenceType
corecom:BillFromPartyReference
type BillFromPartyReferenceType
corecom:RemitToPartyReference
type RemitToPartyReferenceType
corecom:LegalEntityReference
type LegalEntityReferenceType
0..∞
corecom:PaymentReceiptReference
type PaymentReceiptReferenceType
corecom:ProjectReference
type ProjectReferenceType
corecom:Attachment
type AttachmentType
0..∞
type
type NormalizedStringType

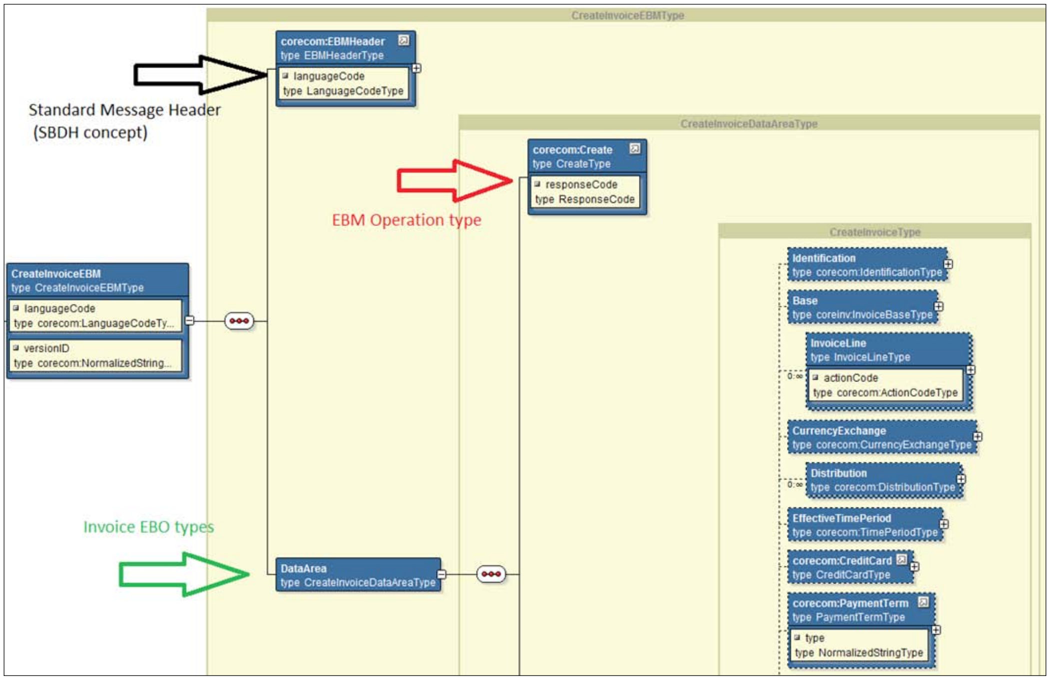
Here, we present the part of Invoice XSD specific to a Telecom PIP pack. It has the following characteristics:

- Represents the business concept of an invoice
- Defined using inputs from multiple applications and content standards
- Common service payload used by all applications
- Designed for extensibility

This is a perfect example of AIA's service contract standardization, and in this case, of data model canonicalization.

Enterprise Business Message

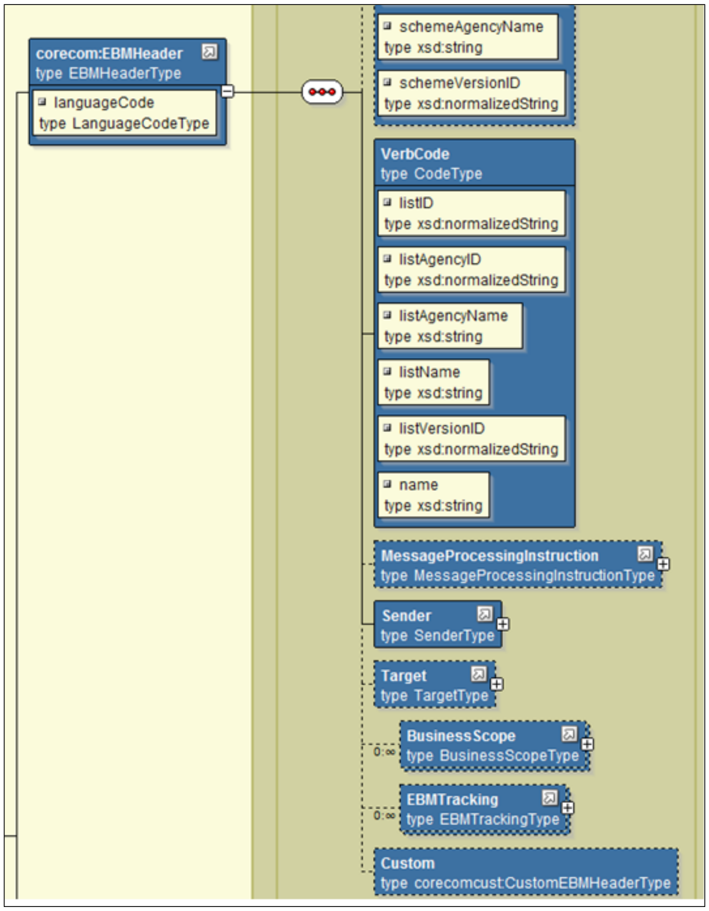
An **Enterprise Business Message** (**EBM**) represents the specific content of an EBO needed to perform a specific activity. For instance, in order to create an invoice, we may not require complete information about that invoice. So, EBM is the operational and transportational form of EBO.



This structure depicts the core parts of EBM to create an invoice based on Invoice EBO. The core EBM features are as follows:

- Application-agnostic encapsulation of an EBO
- Generally coarse-grained and operates either on the whole EBO or its subset
- Payload of a web service operation in EBS
- Semantically precise—performs a specific action, that is, one EBM for one verb (operation)
- A CRUD (create, read/query, update, and delete) operation or a special operation
- Comprises a MessageHeader, verb (that is, the operation name), and an EBO

- Transport protocol agnostic (for example, SOAP, HTTP, HTTPS, and JMS)



EBM Header structure

EBMHeader carries information that can be used for (but is not limited to) the following:

- Tracking important information
- Auditing for business and legal purposes
- Indicating source and target systems
- Error handling and tracing

The AIA EBMHeader implementation is correlated with the existing SBDH standards, and we advise you to ensure that the development of common enterprise structures adheres to this best practice. There are some critical elements in the EBMHeader implementation that are always needed. These are as follows:

- EBMID
- EBOName
- Version
- SenderSystem
- TargetSystem
- ProcessInfo
- ReferenceID
- CreationDateTime

Verb in EBM identifies the action that the sender/requester application wants the receiver/provider application to perform on the EBM. The verb also stores additional information pertaining to the action that needs to be carried out on the noun. Thus, the EBM verb practically implements the Canonical Expression SOA pattern, one of the core SOA patterns responsible for service contract standardization and maintaining discoverability at a desirable level.

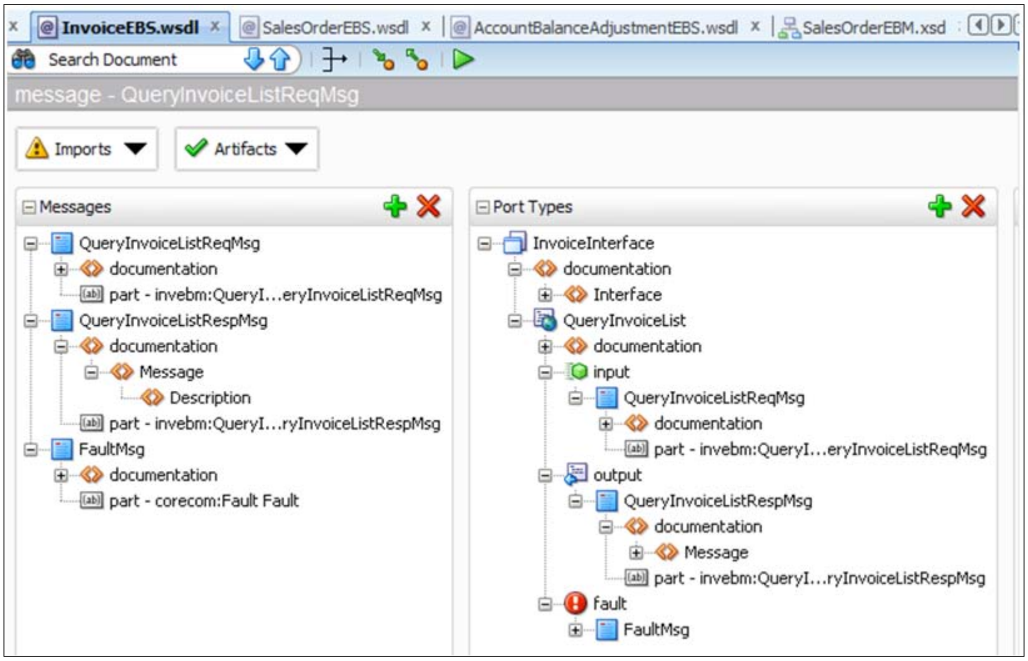
Enterprise Business Services

Enterprise Business Services (**EBS**) are the foundation blocks in AIA. EBS represents the application-independent web service definition to perform a business task. It is self-contained, that is, it can be used independent of any other services. In addition, it can be used within another EBS. EBSs are the standard business-level interfaces that can be implemented by the applications that want to participate in the integration.

EBSs are generally coarse-grained and typically perform a specific business activity such as creating an account in a billing system or getting account balance details from a billing system.

Each activity in an EBS has a well-defined interface described via WSDL (see the next screenshot). This interface description is composed of all the details required for the client to independently invoke the service.

Presented next is the abstract part of **InvoiceEBS.wsdl**:



Application Business Object and Message

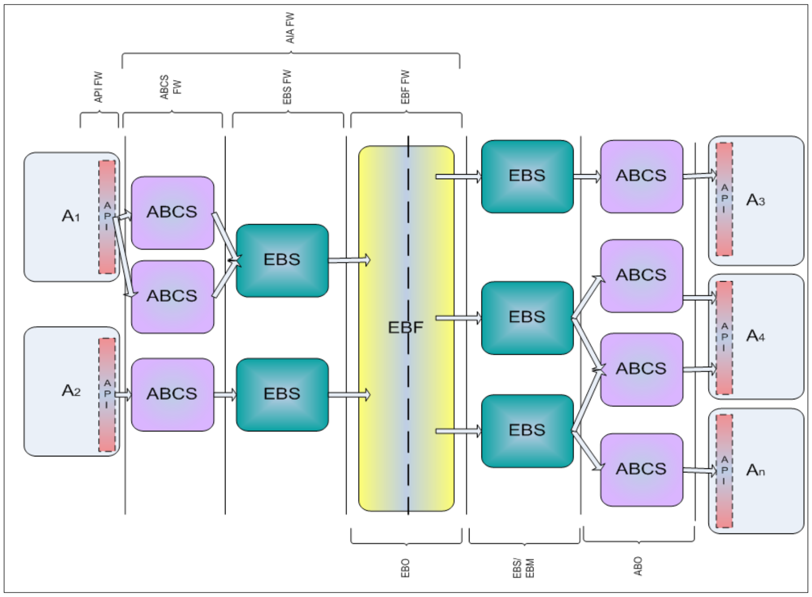
The **Application Business Object** (**ABO**) and the **Application Business Message** (**ABM**) represent the data model relevant to a specific application. The collection of ABOs represents one abstract business entity as a canonical EBO.

In conclusion, we can say that EBO / EBM XSDs together with EBS WSLs shape the Official Endpoint layer, centralizing the services logic and ensuring that the logic is always accessed via a standardized contract. If any alteration is needed, AIA FP provides clear guidance (`http://docs.oracle.com/cd/E23549_01/doc.1111/e17364/bldgintflows.htm#sthref126`) on how to extend the data model or alter the service WSDL.

Therefore, AIA defines integration architecture by adopting an SOA approach. The proposed framework has many out-of-the-box features that can be utilized to address any integration requirements and fulfill SOA principles.

Other SOA principles fulfilled by the framework are as follows:

SOA principle	AIA response
Standardized Service Contract	Enables extensive access to web services and accommodates the following standard MEPs: • Request/Response • Fire-and-Forget • Publish/Subscribe
Service Loose Coupling	This defines loosely bound services that are invoked through communication protocols that stress location transparency and interoperability. It also defines services that have implementation-agnostic interfaces.
Service Abstraction	This adopts an application's independent data model to accomplish the decoupling of a data format.
Service Reusability	This provides a general infrastructure for consistent integrations that are also extensible.
Service Autonomy	This replaces one service implementation with another with minimum impact on the framework.
Service Statelessness	This incorporates synchronous and asynchronous communication and provides partial state deferral (dehydration DB).
Service Discoverability	This uses the Business Process Repository to store business process models and related artifact and the Business Process Publisher publishes the models in HTML format.
Service Composability	This facilitates the use of services in orchestrated process flows and supports incremental adoption and implementation.



From an architectural perspective, AIA can be presented by three main and two supplementary frameworks. These frameworks govern the truly application-agnostic, highly reusable components architecture, where each component is scalable and stackable. This is the outmost evolution of **Receive Transform Deliver** (**RTD**), hub-and-spoke (opposite of ETL) integration patterns. How this framework can affect the design of other frameworks and the implementation of physical infrastructure is explained in the following table:

Framework	Description	Impact on technical infrastructure	SOA patterns employed
EBF	• This framework is BPEL based. • It describes the implementation of the optional PIP backbone related to the specific EBO. Other EBOs can also participate in this flow through connections to the other EBFs or EBSs. An EBF cannot be connected to the ABCS directly. • It is used to host a rule engine/ decision service. • It is present in the EBO/EBM level. • It is Stateful in general. • It is optional in AIA.	• As a part of BPEL, an EBF farm can be clustered using **Load Balancers** (**LB**). • It is installed on the SOA Suite server only. • AA considerations must take into account long-running processes. • It has a DB Dehydrated storage. • It is alternatively Stateful via ProcessHeader and BPEL-correlations. • It has a separate LB for this layer.	• Orchestration • Process centralization • Service Broker (asynchronous) • Partial State Deferral • Rule Centralization • Compensation Service Transaction

Framework	Description	Impact on technical infrastructure	SOA patterns employed
EBS	• This framework is OSB based, Stateless, and mandatory. • It is used to route synchronous invocations. • It is connected to ABSCS and/or EBF.	• It is installed as a separate OSB domain. • It is clustered as Stateless. • Separate LBs can be used. • Coherence installation with WLS	• Enterprise Service Bus • Reliable Messaging • Asynchronous queuing • Intermediate Routing • Rule centralization • Messaging Metadata (SBDH standard implementation)
ABCS	• This framework is BPEL based. • It is used to host adapters, compensation layers, and ABM-EBM transformations. • It is connected to API and EBS. • It is Stateless in most cases and is mandatory.	• It is installed on an SOA Suite server and is realized on BPEL Adapters in HA mode where possible. • Separate LBs can be used.	• Federated Endpoint Layer • File Gateway • Legacy Wrapper • Multi-Channel Endpoint • Partial message validation • Message Format/Model Transformation
BPR	• This is the Business Process Repository and is optional.	• It is implemented using Oracle Service Registry and Enterprise Repository • For runtime, discoverability must be covered by HA patterns.	• Enterprise Inventory • Inventory Endpoint

Framework	Description	Impact on technical infrastructure	SOA patterns employed
CAVS	• This is the Composite Application Verification system and is used as a test framework.	• This is conditional. • It has minimal impact on technical infrastructure.	• Process Abstraction • Entity Abstraction

> Generally, AIA does not recommend the reuse of individual ABCS components, which should be individual for every EBS operation.

It's a sound rule. Adapters are the highly tailored components bound to the individual application, especially when we are talking about Fusion applications. Nevertheless, adapters related to the protocol or technology could be reused by several services (such as JMS, AQ, or File). Therefore, it is your choice to share that type of adapters between different services.

We didn't discuss the Enterprise Business Flows in detail, but as the implementation of long-running processes is related to the single EBO, they are quite commonly known by the typical BPEL implementations. The Order Fusion Demo mentioned earlier could be seen as an example of EBF with all distinctive elements of SOA Suite involved in its assembly.

Even if you do not have Fusion applications in your portfolio, we advise that you to study AIA internal architecture and component implementation anyway, as they are probably the best example by Oracle of a working and very solid SOA approach.

Oracle SOA foundation – runtime backbone

The AIA methodology is a fusion of principles and patterns implemented on Oracle's technology stack, which is in its turn is a fusion of message-oriented middleware, database, application servers, security tools, governance suites, developers workbenches, and languages, which has matured over the years. Each of them presents its own universe, deserved to be explored in many separate books (and it is). We are not aiming to give you any guidance on them as it's simply impossible within the scope of this book, although links to the most recent documentation for each component will be provided in the related section.

Our goal is to demonstrate how certain tools can contribute to solving particular problems of service orientation and what strengths of these tools we should employ during patterns implementation.

We will not only focus on SOA Suite and the OSB components of OFM as they can hardly solve all the common SOA implementation problems alone, but we will also start from the foundation.

The Oracle database

In addition to the database being Oracle's roots and glory, there is nothing that cannot be implemented in the Entity service model by means of the Oracle DB (presently, 11*gR2* and we see that 12*c* has arrived). This is not an exaggeration. Literally, what you can do with Java or C#, you can do with modern PL/SQL. It won't be an exaggeration to state that some utility and task-orchestrated services can be (and probably must be) implemented by a sole Oracle DB. Among many other modern (and not that modern) DB features, the following items help to make this possible:

- Native XML support
- Native object-orientation with Java support
- Multitude of message delivery methods (protocols and MEPs)

Firstly, it's possible because of the full XML support along with the object-orientation support. The XML functionality is presented by the XDB features, available since 2002. Oracle DB has its own JVM with J2EE support, making it a fully capable application server with complete XDK support. Practically all W3C standards are supported with the core J2EE patterns, such as Front Controller (Servlet). The configuration of Oracle XML DB is defined and stored in an Oracle XML DB Repository resource, `/xdbconfig.xml`, and here is the section responsible for the Servlet configuration, as shown in the following code:

```
<httpconfig>
 ...
 <webappconfig>
  ...
  <servletconfig>
   ...
   <servlet-list>
    <servlet> ... </servlet>
    ...
```

```
      </servlet-list>
     </servletconfig>
   </webappconfig>
   ...
   <plsql> ... </plsql>
  </httpconfig>
```

The Servlet configuration section is a child of the protocol configuration described as follows:

```
<protocolconfig>
 <common> ... </common>
 <ftpconfig> ... </ftpconfig>
 <httpconfig> ... </httpconfig>
</protocolconfig>
```

From this section, you can clearly see that two protocols are supported by default: FTP protocol and HTTP protocol. Actually, one more is also supported, a WebDAV. Standard Local Listener is responsible for the handling of all XDB requests according to the parameters registered in the configuration sections as shown earlier, `<ftp-port>` and `<http-port>`. In addition, it is essential to add the related dispatcher entry to the `init.ora` file: `dispatchers="(PROTOCOL=TCP) (SERVICE=<sid>XDB)"`. You can always check the currently configured XDB Listener ports executing under the DBA privileges:

```
select dbms_xdb.gethttpport(), dbms_xdb.getftpport() from dual;
```

Thus, we have standard configurable Servlet features to listen and receive any data (including XML) into Oracle XML DB. But what about the HTTP posting? This functionality is extremely well-covered by the `UTL_* PL/SQL` packages, `UTL_HTTP` in particular. With this package, you can manage the following:

- Session settings
- HTTP requests/responses
- HTTP cookies handling
- HTTP persistent connections
- HTTP error conditions

We will cover this functionality in detail when discussing the SOA-oriented DB APIs.

While sending and receiving the XML data, we must be able to handle the payload inside the DB: store, register, transform, validate, and map to the relational structures. The key element here is XMLType, the abstract Oracle type capable of representing the XML data. Moreover, XMLIndex can sort XML data in addition to the B-tree and Oracle text indexes. XDB provides a complete range of XML-related functions for this data type to handle XML itself (add, modify, and delete nodes, siblings, and elements) and register it in DB. Oracle XML DB Repository provides a hierarchical way of storing XML documents, overcoming the disadvantages of a relational model. A repository provides all the necessary file-handling features including versioning, tagging, and access control based on the **access control lists** (**ACL**).

Despite the existing security mechanisms in XDB, we would not recommend direct access to the XDB features for external service consumers, thus bypassing the security perimeter and Service Gateways. There are many reasons for that as XDB simply does not provide the necessary security-related patterns, discussed earlier for layer 7, visualized in the first figure of this chapter but one reason has to be mentioned explicitly. The implementation of the Trusted Subsystem SOA security pattern requires the separation of security accounts for the service and the service's underlying resources. Nevertheless, for internal consumers, or for consumers isolated by means of **Secure Gateway** (**SG**) in layer 7, XDB provides an excellent opportunity for the entity-oriented service model's implementations. The principle of concerns' separation is maintained by the segregation of security features delegated to the SG and the entity service's basic operations delegated to the XDB. The main concern here should be the scalability because, for better results, we need a more atomic service realization.

Surely, for XDB Servlet creation, complex XPath operations, transformations, and so on, you can rely on something more robust than PL/SQL (although the oldie but goodie PL/SQL can do a lot). It will be quite right to say that PL/SQL nowadays can be seen as a wrapper for Java, thanks to SQLJ specification. You can create your own Servlet, compile it, and load it into your DB using the following:

```
loadjava -grant public –user <xdbuser>/xdbuserpwd>@<XDBSID>
CustomServlet.class
```

Then, register it in /xdbconfig.xml in a preceding section where we discussed the configuration of Oracle XML DB.

However, native DB Java capabilities are wider than just XDB functionality. You can do practically everything—from transaction control and complex XML operations to file handling—putting aside the UTL_FILE package. Again, despite being perfectly feasible, this solution should be clearly analyzed for scalability requirements.

From messaging infrastructure in addition to synchronous HTTP(S) calls, Oracle DB can offer the native support of reliable asynchronous communications compatible with JMS queuing, known as **Advanced Queuing** (**AQ**). AQ is a database-integrated messaging infrastructure. Thus, all the DBs' operational benefits, such as High Availability, Scalability, and Reliability, are applicable to the messages and queues in AQ. Standard database features such as backup and recovery and security and manageability are available for AQ as well. All AQ features can be accessed from the DB side by Java or PL/SQL (the two main packages are `DBMS_AQADM` and `DBMS_AQ`), where you can enqueue, dequeue (one or a group of messages), sort, propagate, and perform many other functions. The payload data type could be opaque (ANYDATA) or based on a predefined abstract data type (ADT). Messaging interfaces on the consumer side can be based on any popular programming interface, but naturally, in Fusion Middleware, the Java JMS API is used (the `oracle.jms` package).

All the previously discussed features present Oracle DB (we are talking about the classic RDBMS 11*g*) as a perfect platform candidate for DB-related Entity services and some utility, such as the File Gateway SOA pattern, responsible for fetching files, and transforming and persisting in single or multiconsumer AQ(s). However, the role of DB as a process state deferral store for task-orchestrated services is hard to underestimate. It's a primary feature of Oracle RDBMS in Fusion Middleware, employed in Framework 4. DB-based Rule centralization, Policy centralization, and storage for runtime-related XML artifact (XSLTs, XPaths, other XML fragments, and XQuery as well), together with reliable messaging persistence in AQ, cover all the types of runtime data persistence we need. Responsible for design-time, Discoverability Services metadata is also securely kept in an Oracle DB, Framework 6, and accessed via a unified inventory's endpoint. It is worth mentioning that the rule engine SDK is available for the DB, Java, and PL/SQL.

We have mentioned RDBMS several times as you may have noticed, and we did it on purpose. Yes, XDB with XML-relational model mapping is crucial to handle well-structured, query-intensive data such as Order, Invoice, Client, and so on. It is rather hard to justify why intensively queried data elements should be stored in CLOB or XMLType fields. But what if all we need is to pertain the object-oriented data and pass it further on for client-side processing? Yes, that's an AJAX with the JSON payload type of processing. Nevertheless, the payload data could also be the Lists, Sets/Ordered Sets, Hash maps, primitives, and so on. All types of message-object-related mappings are quite expensive and could be complex; moreover, we don't always need them on the **Message Oriented Middleware** (**MOM**) side.

Oracle offers good additions (we wouldn't describe it as a pure alternative) to RDBMS in the form of Oracle Big Data Appliance (optimized for Oracle Exadata Database Machine, but it can run on any hardware). This includes an open source distribution of Apache Hadoop (`http://hortonworks.com/hadoop/`), the Oracle NoSQL Database (frontend for well-known Berkley DB), Oracle Data Integrator Application Adapter for Hadoop, and Oracle Loader for Hadoop.

The last two components implement a MapReduce-distributed computing pattern (`http://highlyscalable.wordpress.com/2012/02/01/mapreduce-patterns/`), linking Hadoop Big Data realized as **Hadoop Data File System** (**HDFS**) to Oracle's external tables, which are transparent for regular SQL queries. The pattern's name `MapReduce` could be misleading as the object-relational mapping is not the primary goal of this solution. In this pattern, we try to achieve the highest level of processing parallelism for clustered multitenant data in HDFS. We are literally mapping chunks of data with unique numbers (key-value pairs), where the key is associated with processing nodes and responsible for the reducing of the overall workload by the parallel processing of related data chunks. Processing could be online or offline and multiconsumer AQs can be used quite extensively by job controllers in some pattern's realizations.

So, we could potentially have the classic RDBMS and NoSQL key-value distributed stores at the same time. All these types of data access, loading, and distribution put serious requirements on the data access layer, technically residing in the App Server layer (the next layer in the technical infrastructures hierarchy). Not only will components of the mentioned `MapReduce` pattern be implemented there, but practically the whole data layer must be abstracted in a way that data manipulation routines do not affect the application's logic, and apparently, old JPA is not enough. The latest JPA 2.*x* (JSR 338) covers most of aspects of access schema-less DB, in addition to the traditional implementations. Polyglot Persistence allows the abstracting of JPA's implementations even further, and with in-memory data-grid Coherence (JSR 107), we can achieve the highest level of data abstraction and availability.

When discussing a DB's role in an SOA technical infrastructure, it is quite interesting to mention the number of critical errors solely related to DB faults/mishandling. Does the following list look familiar?

- We dequeued the inbound message from the JMS queue, but failed to persist it in the DB because of tablespace problems. Where can we find the message for process recovery? (By the way, should we mention that the queue retention time was set to `0`?)

- During the night batch job, one longops runs out of rollback size as it was not big enough. About 25 percent of our active orders failed CRM-ERP synchronization and now applications are out of sync. Worse, policy-based automated recovery triggered automatically, but it also failed for the same reason. Now a manual clean-up is necessary. (The DBA forgot to assign a longops transaction to the one big rollback segment.)
- BPEL tried to dehydrate the process, but the tablespace was full. (Obviously, the SCA audit level was not tuned, but it seems that the DBA didn't monitor tablespace usage as well.)

From our experience, we can say that about 60 percent of all SCA crashes in production are caused by DB-related reasons.

The implementation of the State Deferral pattern for orchestrated services in terms of data persistence has significant impact on storage management, where one of the critical operations is the process of data cleansing. Oracle provides purge scripts together with purge strategies, depending on your operational requirements.

The Oracle application server

The core strategic assets of the Oracle WebLogic application server are interactively described at `http://www.oracle.com/webfolder/technetwork/tutorials/obe/fmw/wls/Poster/poster.html`.

WebLogic Application Server together with the DB is another keystone in SOA foundation. It's an engine of engines, so to say. All that you will build, deploy, execute, or consume as a resource will reside on WLS. An exception could be Secure Gateway (the Oracle API Gateway), which for good reasons we will touch upon in the *Securing service interactions – Security Gateway* section, is not based on WLS.

As we mentioned previously, we will focus only on some core features enabled with the SOA functionality, rather than on the WLS administration aspects. Firstly, we have to mention that WebLogic 11*g* supports JEE 5 and JAX-WS 2.1 for the web service development when the newest 12*c* release supports JEE6 and JAX-WS 2.2.

In the context of the SOA infrastructure, what is the most interesting for us is which resources can be securely provided for utilization and how they can be managed on runtime. Needless to say that resources such as JMS for asynchronous communications usually do not belong to the particular application (if the application that encapsulates the communication channel is down, we can hardly rely on that channel anymore), so we need the JMS server to handle our JMS messaging.

All these resources, and some more, are securely provided by WLS:

- EJB resources
- **Enterprise Information Systems** (**EIS**) resources
- **Java DB Connectivity** (**JDBC**) resources
- **Java Messaging Service** (**JMS**) resources
- **Java Naming and Directory Interface** (**JNDI**) resources
- Web service resources
- Work context resources

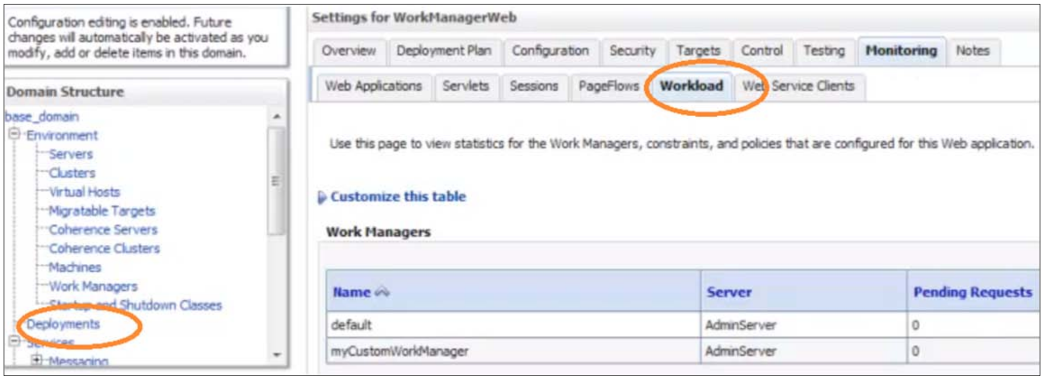
As WLS is compliant to JSR-255 JMX management extension standards, there are many ways to manage shared recourses. For instance, the standard sequence to create JMS Queue—`Creating a JMS Server -> Creating a Module -> Creating Queue ->`—can be fulfilled using the WLS console or the WLS scripting tool. The basic resources (such as JMS) can be combined to expose shared WLS services, which are also known as managed WLS resources. Among others, we would like to mention one managed service responsible for establishing a reliable messaging infrastructure—Oracle's **Store and Forward** (**SAF**) service.

The SAF service enables WebLogic Server to distribute messages reliably between applications that are spread across the WebLogic Server instances; effectively, the implementation of the Reliable Messaging SOA pattern. For example, with the SAF service, an application that runs on or connects to a local WebLogic Server instance can reliably send messages to an endpoint that resides on a remote server. If the destination is not available at the moment the messages are sent, either because of network problems or system failures, then the messages are saved on a local server instance and forwarded to the remote endpoint once it becomes available.

Oracle Work Manager is responsible for the second part of the question, that is, how resources are managed. To manage work in the installed applications, we define one or more of the following Work Manager components:

- Fair Share Request Class
- Response Time Request Class
- Min Threads Constraint
- Max Threads Constraint
- Capacity Constraint
- Context Request Class

Depending on your preferences, Work Manager can be assigned to any application (including the Web application), application component in the WLS domain, or OSB Business Service. You have to choose one of the four configuration files (please see the WLS documentation `http://www.oracle.com/technetwork/middleware/weblogic/documentation/index.html` for more information) to specify thresholds for the Work Manager components and assign them to your deployed application or component according to its deployment descriptor. You can always check the status using the WLS console—**Deployments** | **Monitoring** | **Workload**—as shown in the following screenshot:



Configuration editing is enabled. Future changes will automatically be activated as you modify, add or delete items in this domain.

Use this page to view statistics for the Work Managers, constraints, and policies that are configured for this Web application.

Work Managers

Name	Server	Pending Requests
default	AdminServer	0
myCustomWorkManager	AdminServer	0

WLS Work Manager's configuration

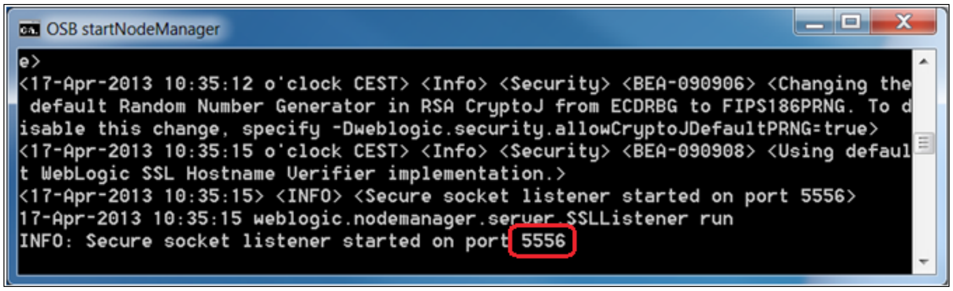
Work Manager uses a common execute queue (common thread pool), so it prioritizes work based on the rules we define in the configuration files. The rules can be set for response time, max/min threads, and run-time metrics, including the actual time it takes to execute a request and the rate at which requests enter and leave the pool. For example, simple math gives you the understanding of thread pool utilization and backlog size. If your web service can guarantee its performance for 20 concurrent calls (Max Threads) and Maximum Capacity for its work managers is 25, then backlog queue size will be 5 (*25-20*), and these requests will wait for a thread to become available. For rejected threads (over 25), you can assign a meaningful response message.

> We should be very careful with numbers that we put in the Work Manager's configuration files. For instance, it's definitely not a good idea to have the number of threads higher than that of the available connections in the DB connection pool.

Finally, from the Orchestration and Service Bus perspective, WLS is the host of the whole SOA infrastructure maintained in the high-availability mode. Start the node managers first using the designated script and then start your servers.

Needless to say that the node manager, which handles cluster nodes interoperability, should not be a single point of failure and must also be covered by high-availability options (please see the WLS documentation `http://www.oracle.com/webfolder/technetwork/tutorials/obe/fmw/wls/12c/10-NodeMgr--4472/nodemgr.htm`). In the following screenshot, you can see all our servers (SOA, OSB, and BAM) running on the same WLS. Although this is not recommended for production, please plan your infrastructure in a more segregated way according to the framework layers we explained earlier. In the following screenshot, you can see all our servers (SOA, OSB, and BAM) running on the same WLS on entering the following command:

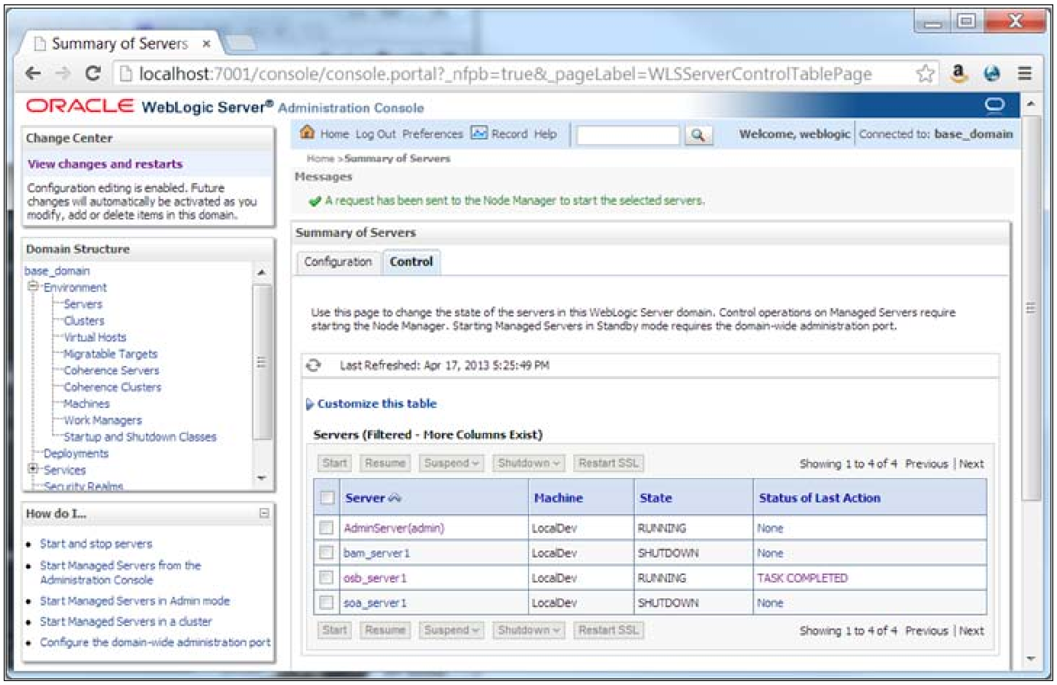
`/wlserver_10.3/server/bin/startNodeManager.*`



```
e>
<17-Apr-2013 10:35:12 o'clock CEST> <Info> <Security> <BEA-090906> <Changing the
 default Random Number Generator in RSA CryptoJ from ECDRBG to FIPS186PRNG. To d
isable this change, specify -Dweblogic.security.allowCryptoJDefaultPRNG=true>
<17-Apr-2013 10:35:15 o'clock CEST> <Info> <Security> <BEA-090908> <Using defaul
t WebLogic SSL Hostname Verifier implementation.>
<17-Apr-2013 10:35:15> <INFO> <Secure socket listener started on port 5556>
17-Apr-2013 10:35:15 weblogic.nodemanager.server.SSLListener run
INFO: Secure socket listener started on port 5556
```

Although this is not recommended for production, please plan your infrastructure in a more segregated way according to the two framework layers explained earlier. The WLS console with core OFM servers can be accessed after starting your server from the related WLS domain, see the following command (.cmd or .sh depending on your OS):

```
/user_projects/domains/<my_domain>/bin/startWebLogic.*
```



Summary of Servers
localhost:7001/console/console.portal?_nfpb=true&_pageLabel=WLSServerControlTablePage
ORACLE WebLogic Server® Administration Console
Home Log Out Preferences Record Help
Welcome, weblogic
Connected to: base_domain
Change Center
View changes and restarts
Domain Structure
base_domain
Environment
Servers
Clusters
Virtual Hosts
Migratable Targets
Coherence Servers
Coherence Clusters
Machines
Work Managers
Startup and Shutdown Classes
Deployments
Services
Security Realms
How do I...
Start and stop servers
Start Managed Servers from the Administration Console
Start Managed Servers in Admin mode
Start Managed Servers in a cluster
Configure the domain-wide administration port
Home >Summary of Servers
Configuration
Control
Customize this table
Start
Resume
Suspend
Shutdown
Restart SSL
Showing 1 to 4 of 4 Previous | Next

Configuration editing is enabled. Future changes will automatically be activated as you modify, add or delete items in this domain.

Messages

✓ A request has been sent to the Node Manager to start the selected servers.

Summary of Servers

Use this page to change the state of the servers in this WebLogic Server domain. Control operations on Managed Servers require starting the Node Manager. Starting Managed Servers in Standby mode requires the domain-wide administration port.

Last Refreshed: Apr 17, 2013 5:25:49 PM

Servers (Filtered - More Columns Exist)

□	Server	Machine	State	Status of Last Action
□	AdminServer(admin)	LocalDev	RUNNING	None
□	bam_server1	LocalDev	SHUTDOWN	None
□	osb_server1	LocalDev	RUNNING	TASK COMPLETED
□	soa_server1	LocalDev	SHUTDOWN	None

WLS core OFM Servers

Summarizing what we have just discussed, we can describe the WLS as the most generic way of covering all the SOA principles for our infrastructure, since the DB we discussed earlier only provides Statefulness (Process Dehydration) and Loose Coupling (AQ) directly. The application server covers Loose Coupling by maintaining JMS and SAF in an application-independent way. JNDI also contributes to Loose Coupling, isolating the resource name, for instance, the DB connection pool from the physical DB location; so when the database is moved or changed, you do not have to alter your DB-related service. JNDI's resource-naming unification and configuration centralization also positively impacts discoverability, but the main contributor to the realization of this principle is UDDI. The UDDI 2.0 Server is part of WebLogic Server and is started automatically when WebLogic Server is started. With complete JAX-WS 2.*x* support, WLS helps you to maintain Service Abstraction and Standardized Contract, but of course, you should put some effort into designing the services to realize these principles.

WebLogic as the integral component of all infrastructures, abstracting, sharing and monitoring resources, is the key contributor to Service Autonomy. Using the WLS Administration console or scripting tools, Work Manager can assign or revoke computing power to the installed application and components, thus increasing or decreasing their runtime autonomy. The clustered implementation controlled by WLS node manager(s) increases SOA applications' resiliency and high availability. Consequently, because of high availability, abstraction, and visibility, all the servers' resources are highly reusable in a very controllable manner (thanks to Work Managers), and as a result, are composable. The unification of the resource types and the ways of resource discovery and management open the door for the implementation of the Canonical Resource SOA pattern; this helps to present all components of the underlying technical infrastructure as unified blocks, easily shared and consumed by different services. Elastic resource provisioning based on the simplified replication model, specific for Cloud (mostly for 12*c*, but 11*g* also has some support), helps to avoid the decreasing of service autonomy during the implementation of the Canonical Resource pattern.

Again, we have to stress the fact that WLS cannot guarantee the employment of all of these principles and characteristics alone in your architecture. Only basic SOA patterns, such as Reliable Messaging and Canonical Resources for some elements, can be provided out of the box. We have to architect our components right from the start to achieve a desirable level of Composability.

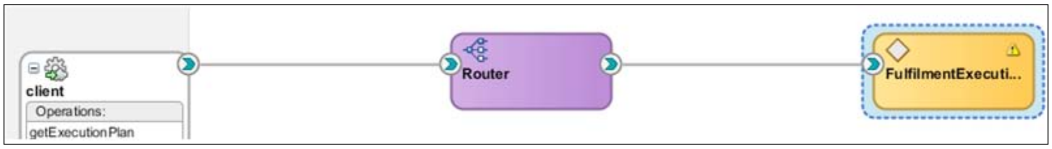
There are some more things on the technical side related to SOA, which are not supported by WLS (11*g* and 12*c*):

- No OSGi support. Even though it was said that WebLogic 12*c* uses OSGi for internal modularization and to deliver products such as Oracle's **Complex Event Processor** (**CEP**), there is no direct support for this standard. For instance, if you are developing components for unmanaged devices, and you need to implement the dynamic component model managed remotely, Fuse/ServiceMix ESBs could be a better choice.
- Limited JBI support. Same as previous.
- No OAUTH 2.0 support for RESTful services. You should rely on a secure perimeter solution.

The GlassFish application server could be the alternative for WLS, but we must remember that it isn't a strategic product; not all OFM products will be supported and some enterprise features will not be available.

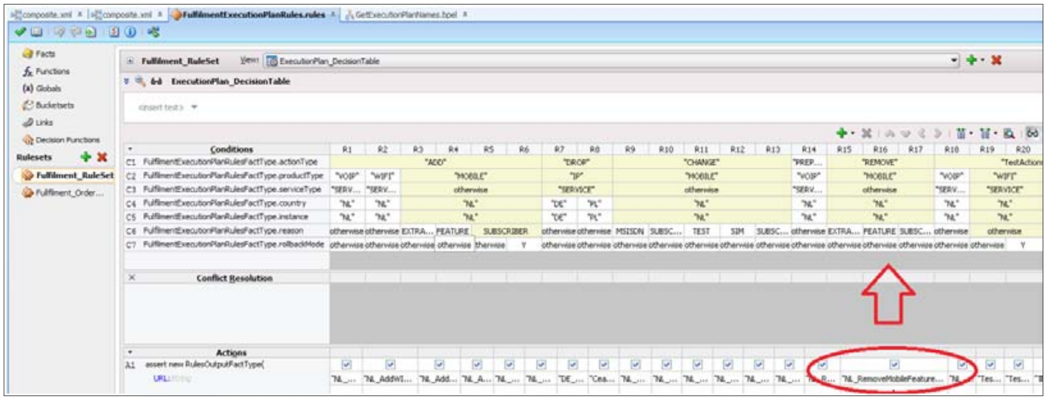
The Oracle Rule Engine

For many OFM developers, **Rule Engine (RE)** is visualized as a decision component in the SOA suite, shown as follows:



Indeed, this is the most common way of utilizing the power of business rules in composite applications, expressing very complex conditions, combined in collections (rulesets). The Decision component is the method of exposing the Decision services. The Decision component is an SCA component that can be used within a composite wired to a BPEL component or exposed directly as Service. In addition, the Decision components are used for the dynamic routing capability of Mediator and Advanced Routing Rules in Human Workflow, Case Management, and BPM. The Decision component can be seen as a web service wrapper for the decision function and ultimate endpoint for a subset of the rule dictionary. The decision function in the rule dictionary can be presented as rules or decision tables. This dictionary, acting as a central rule repository, contributes to the implementation of the Rule Centralization SOA pattern.

Business Rule Designer is the frontend to define and author various rules in SOA Suite and combines them in rulesets. The following screenshot demonstrates how a set of conditions based on Facts extracted from the message XSD (imported from message header elements, but also can be from Java class or created explicitly in designer) can be linked to a concrete action; in this case, the execution of a certain business process related to the user request. The following screenshot gives a view of Business Rule Designer:



	Conditions	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15	R16	R17	R18	R19	R20
C1	FulfilmentExecutionPlanRulesFactType.actionType	"ADD"						"DROP"		"CHANGE"					"PREP...	"REMOVE"			"TestAction		
C2	FulfilmentExecutionPlanRulesFactType.productType	"VOIP"	"WIFI"	"MOBILE"				"IP"		"MOBILE"					"VOIP"	"MOBILE"			"VOIP"	"WIFI"	
C3	FulfilmentExecutionPlanRulesFactType.serviceType	"SERV...	"SERV...	otherwise				"SERVICE"		otherwise					"SERV...	otherwise			"SERV...	"SERVICE"	
C4	FulfilmentExecutionPlanRulesFactType.country	"NL"	"NL"	"NL"				"DE"	"PL"	"NL"					"NL"	"NL"			"NL"	"NL"	
C5	FulfilmentExecutionPlanRulesFactType.instance	"NL"	"NL"	"NL"				"DE"	"PL"	"NL"					"NL"	"NL"			"NL"	"NL"	
C6	FulfilmentExecutionPlanRulesFactType.reason	otherwise	otherwise	EXTRA...	FEATURE	SUBSCRIBER		otherwise	otherwise	MSISDN	SUBSC...	TEST	SIM	SUBSC...	otherwise	EXTRA...	FEATURE	SUBSC...	otherwise	otherwise	
C7	FulfilmentExecutionPlanRulesFactType.rollbackMode	otherwise	otherwise	otherwise	otherwise	therwise	Y	otherwise	otherwise	otherwise	otherwise	otherwise	otherwise	otherwise	otherwise	otherwise	otherwise	otherwise	otherwise	otherwise	Y
×	Conflict Resolution																				
	Actions																				
A1	assert new RulesOutputFactType(	☑	☑	☑	☑	☑	☑	☑	☑	☑	☑	☑	☑	☑	☑	☑	☑	☑	☑	☑	
	URL:	"NL_...	"NL_AddWI...	"NL_Add...	"NL_A...	"NL_...	"NL_...	"DE_...	"Cea...	"NL_...	"NL_...	"NL_...	"NL_...	"NL_...	"NL_...	"...R...	"NL_RemoveMobileFeature...	"NL_...	"Tes...	"Tes...	"T

Business Rule Designer

Compared to the decision tables, the rule functions are more `IF-THEN` like, where `IF` is a set of conditions or pattern matches and `THEN` is the list of actions. A rule might perform several types of actions. An action can add, modify, or remove facts and can execute a Java method or perform a function, which may modify the status of facts or create facts. Here you can see the problem. If a rule action can modify the fact and if a decision is based on this fact, then it's quite possible to create a never-ending loop within the ruleset or lead to the so-called "combinatorial rules explosion." When a rule adds facts and when these facts run against the rules, this process is called an **inference cycle**. Luckily, Oracle implements the JSR-94-compliant RETE logic (`http://docs.oracle.com/cd/E15523_01/integration.1111/e10228/intro.htm`) for optimization of a rule's execution, avoiding unnecessary checks for facts when they were altered or deleted during the rule function execution. The RETE algorithm provides the following benefits:

- Rule orders independence.
- Optimization across multiple rules.
- High-performance inference cycles; typically, each rule firing changes just a few facts. The cost of updating the RETE network is proportional to the number of changed facts, not the total number of facts or rules.

Yet we have to warn you that as long as all the rule executions for the highest performance is memory-based, combinatorial explosion is still a threat and the RETE algorithm's compliance alone cannot prevent memory depleting. Plan all your rulesets wisely and do not follow the Rule Centralization too rigorously. The Rule Centralization must be strictly observed for rule development, authoring, and testing, but physical implementation can be divided between frameworks and application layers depending on the rulesets' complexity. As any centralization, the rule centralization can implement the single point of failure and definitely affects the Service Autonomy principle negatively, taking away business logic (or part of it) from the business service. However, the last bit is the whole idea of increasing business agility and Composability, and there are numerous positive customer use cases where Oracle RE has been implemented in mission-critical, online-fraud prevention and money laundering detection applications that handle and scan billions of transactions or records daily. Thus, there are no doubts that Oracle RE can provide fast performance and the highest level of reliability.

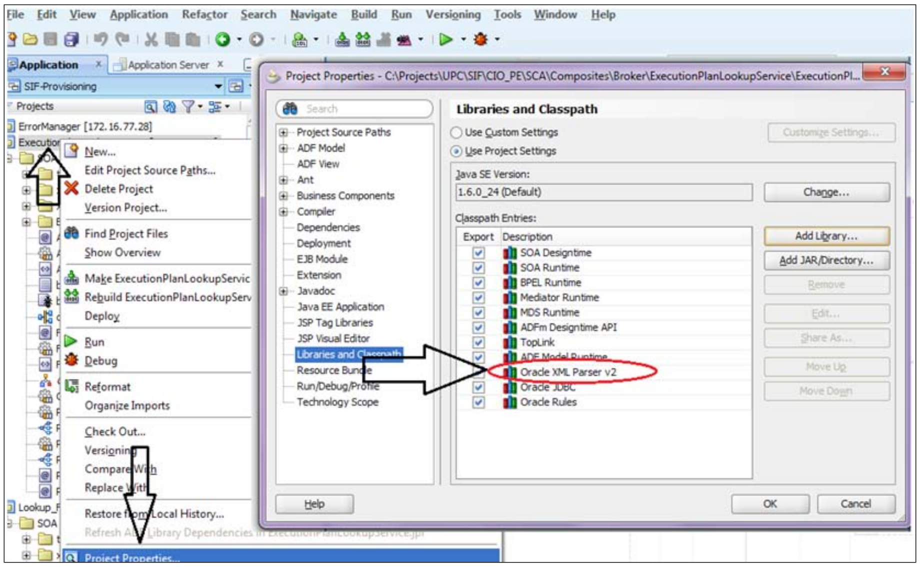
In general, Oracle RE is a standalone Java application that provides a very comprehensive SDK and API for the rules' utilization from any element of infrastructure: DB or OSB. Even if we do not have a rule-related activity for the request and response pipes in OSB, it is quite possible to implement rules using Java calls on the RE API. Oracle rules can be expressed by **Rule Language** (**RL**), which is a subset of Java and is relatively easy to use. Oracle DB 11*g* has packages that present built-in rule engine functionality; primarily, `DBMS_RULE_ADM` and `DBMS_RULE`.

Oracle transformation and translation engine

As XML is the core standard in service-oriented computing, common to all frameworks, the last but not least shared technical component discussed in the *Oracle SOA Foundation – runtime backbone* section is Oracle XML Development Kit—a set of tools, utilities, and modules, bundled with Oracle DB, JDeveloper, OSB, and SOA Suite. All operations with XPaths, XQueries, XML nodes, XSDs, and XSLTs are possible because of the XDK functions. Some core functionalities related to the latest versions of XML standards are listed here:

- The JAXB-compliant XML class generator to generate classes from DTDs and XML schemas on runtime and design time
- The DOM v 3.0 and SAX-compliant XML Parsers, full support for JAXP 1.3 interfaces, and implements and access XMLType in Oracle DB
- XSLT v 2.0 processors for transformation or the rendering of XML
- XML schema processors for runtime and design time schema validation
- XML SQL utility, essential for XDB to convert SQL queries into XML

All core XML libraries are associated with an SOA project upon creation, but you can always verify the existing libraries or add new ones following **Project Properties** | **Libraries**. The Oracle XML Parser v2 library is mainly responsible for XML parsing and validation, as shown in the following screenshot. TopLink mentioned in the previous section is the main O-R mapper. Refer to `http://www.oracle.com/technetwork/middleware/toplink/overview/index.html` and `http://www.oracle.com/technetwork/middleware/toplink/overview/index.html` for more details.



Project Properties - C:\Projects\UPC\SIF\CIO_PE\SCA\Composites\Broker\ExecutionPlanLookupService\ExecutionPl...

Libraries and Classpath

○ Use Custom Settings
● Use Project Settings

Java SE Version: 1.6.0_24 (Default)

Classpath Entries:

Export	Description
☑	SOA Designtime
☑	SOA Runtime
☑	BPEL Runtime
☑	Mediator Runtime
☑	MDS Runtime
☑	ADFm Designtime API
☑	TopLink
☑	ADF Model Runtime
☑	Oracle XML Parser v2
☑	Oracle JDBC
☑	Oracle Rules

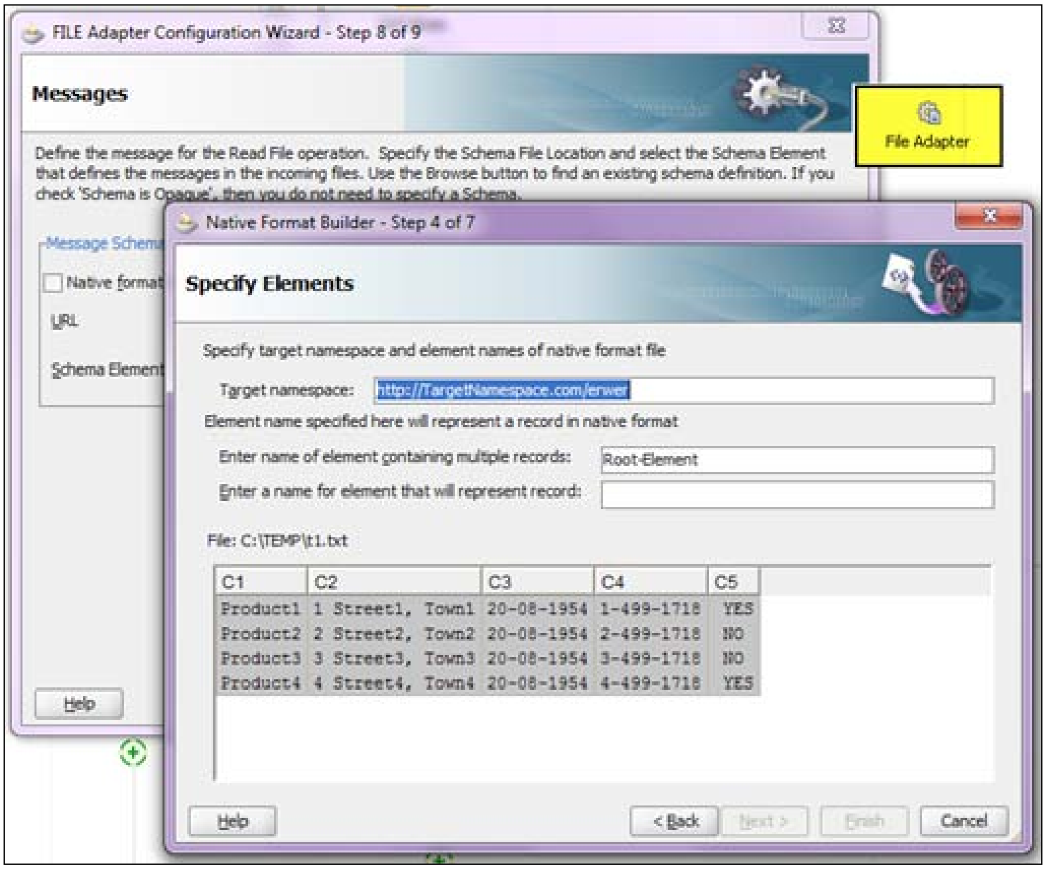
Oracle XML Parser configuration

Thanks to the mouse's right-click menu functions—**Reformat** and **Validate XML**—you can always check the consistency of your document and keep it in a readable form. The same validation can be done using the following command line:

```
>java oracle.xml.parser.v2.oraxml -schema PurchaseOrder.xml
```

The input XML file is parsed without errors using the Schema validation mode. Similar to this, to apply style sheets in the command line, the `oraxsl` utility is handy.

The translation capability of Oracle XDK is provided by **Native Data Format Translator** (**NDFT**) and this feature is highly useful in the adapter framework when we map non-XML files into XML and vice versa, as shown in the following screenshot:



Define the message for the Read File operation. Specify the Schema File Location and select the Schema Element that defines the messages in the incoming files. Use the Browse button to find an existing schema definition. If you check 'Schema is Opaque', then you do not need to specify a Schema.

Specify target namespace and element names of native format file

Target namespace: http://TargetNamespace.com/erwer

Element name specified here will represent a record in native format

Enter name of element containing multiple records: Root-Element

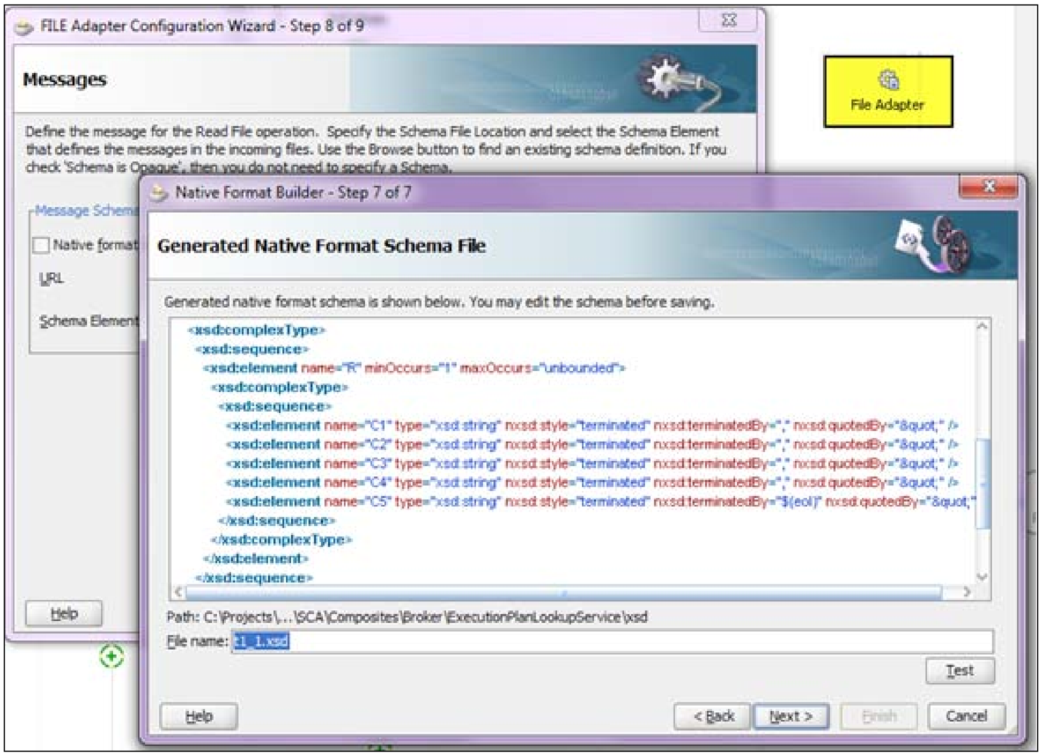
Enter a name for element that will represent record:

File: C:\TEMP\t1.txt

C1	C2	C3	C4	C5
Product1	1 Street1, Town1	20-08-1954	1-499-1718	YES
Product2	2 Street2, Town2	20-08-1954	2-499-1718	NO
Product3	3 Street3, Town3	20-08-1954	3-499-1718	NO
Product4	4 Street4, Town4	20-08-1954	4-499-1718	YES

Native Data Format Translator

Here, simple comma-separated files were translated into simple XML files suitable for further processing. XSD for the complex type, generated by Native Format Schema generator from initial CSV file you can see in the following screenshot:


FILE Adapter Configuration Wizard - Step 8 of 9
Messages
Message Schema
Native format
URL
Schema Element
Help
File Adapter
Native Format Builder - Step 7 of 7
Generated Native Format Schema File
Test
Help
< Back
Next >
Finish
Cancel

Define the message for the Read File operation. Specify the Schema File Location and select the Schema Element that defines the messages in the incoming files. Use the Browse button to find an existing schema definition. If you check 'Schema is Opaque', then you do not need to specify a Schema.

Generated native format schema is shown below. You may edit the schema before saving.

```
<xsd:complexType>
 <xsd:sequence>
  <xsd:element name="R" minOccurs="1" maxOccurs="unbounded">
   <xsd:complexType>
    <xsd:sequence>
     <xsd:element name="C1" type="xsd:string" nxsd:style="terminated" nxsd:terminatedBy="," nxsd:quotedBy="&quot;" />
     <xsd:element name="C2" type="xsd:string" nxsd:style="terminated" nxsd:terminatedBy="," nxsd:quotedBy="&quot;" />
     <xsd:element name="C3" type="xsd:string" nxsd:style="terminated" nxsd:terminatedBy="," nxsd:quotedBy="&quot;" />
     <xsd:element name="C4" type="xsd:string" nxsd:style="terminated" nxsd:terminatedBy="," nxsd:quotedBy="&quot;" />
     <xsd:element name="C5" type="xsd:string" nxsd:style="terminated" nxsd:terminatedBy="${eol}" nxsd:quotedBy="&quot;"
    </xsd:sequence>
   </xsd:complexType>
  </xsd:element>
 </xsd:sequence>
```

Path: C:\Projects\...\SCA\Composites\Broker\ExecutionPlanLookupService\xsd

File name: c1_1.xsd

Native Format Schema Generator

However, what if a situation is not as simple and we cannot use the preconfigured wizards for files separated with a predefined delimiter of fixed-length files? XDK's Native Schema translator provides constructs that can cover practically all possible complex cases, helping with conversion to canonical XML. A few of them that are extremely useful in our opinion (from practical experience) are listed as follows:

Native schema translator	Usage
`startsWith`	This looks for the specified string in the native data.
`surroundedBy`	This looks for the native data being surrounded by the specified string.
`terminated By`	This looks for the native data being terminated by the string specified.
`skip`	This skips the specified number of bytes or characters.

Frankly, by using these four translators, you can find any piece of information in a very complex mutilated file and map it to your canonical model. We advise you to look at the full list of constructs and related examples in Oracle's *Native Format Builder Wizard* documentation, related to the adapter framework. Finally, if this approach doesn't work (yes, the IBM EBCDIC encoding with all possible mixes of formats are still in use), use Java callouts from your framework (OSB or BPEL) and implement your chopper using the Tokenizer patterns with any extensions you need.

Summarizing this, we simply do not know how to express more the importance of a reliable and performing XML framework for the whole Enterprise architecture (not only SOA). All six core frameworks rely on the robustness of Oracle XDK, but the Security and Adapter frameworks, our gateway keepers, are explicitly responsible for maintaining our canonical data models and reducing the number of transformations and translations.

> Some complex products, such as Oracle E-Business Suite, comprise several XDKs, sometimes in different versions, as DB/XDB and the Application server can have their own instances. Be careful with the complex XML constructs and transformation schemas and always pay attention to the parser you are using within your application. There may be some problems with compatibility and definitely, the performance will be different.

How Oracle products compose the SOA framework

We discussed the foundation of Oracle SOA, the components that will always be present in your infrastructure. Even if you decide to install OSB in a lightweight mode (no DB and **Repository Creation Utility** (**RCU**)) without reporting the functionality, the Derby DB will be installed anyway. Combined together, these products cover all SOA principles and introduce all `WS-*` standards. When properly maintained, they will help you to regularly address the common challenges you face during the implementation of the SOA infrastructure. Here are just some of the problems and patterns that address what we have already mentioned:

Common problem	Pattern addressing the problem
Latency on the service provider's side negatively affecting the service consumer's functions	• **Asynchronous Queuing**: Covered by WebLogic's implementation of JMS Server with queues and topics.

Common problem	Pattern addressing the problem
Unreliable communication channels between the service consumer and service provider	• **Reliable Messaging**: Addressed by WebLogic's Store and Forward Service (SAF), with agents securely delivering messages with all the necessary acknowledgements.
Making stateful (and stateless as well) services constantly available in a fault-tolerant way and ready to scale up rapidly	• **Service Grid**: Insured by Coherence, in-memory grid computing solution, hosted by WebLogic (JSR-107 compliant). Working together with OSB, provides a simplified service deployment on the clustered environment and rapid high-volume processing. Combined with Oracle Event Processing implements ultra-fast event processing network.
Computing resource unification and management harmonization	• **Canonical Resources**: Covered by WebLogic's JNDI, Resource Management, and Work Managers implementation of unified resource management.
Increasing the reliability and availability of services	• **Redundant Implementation**: Guaranteed by WebLogic clustered implementation, including Coherence and Node Manager in a redundant mode.
Controlling all business rules centrally and avoiding business logic from creeping in	• **Rule Centralization**: Provided by Oracle Rule Engine with SDK, APIs, and management console.

In general, what we are missing here is the services collaboration and operational support for long- and short-running services (Orchestration and ESB, respectively), security enforcement, and service governance (including Enterprise Repository). However, we would first like to discuss the positioning of the development, testing, and deployment tools in these frameworks.

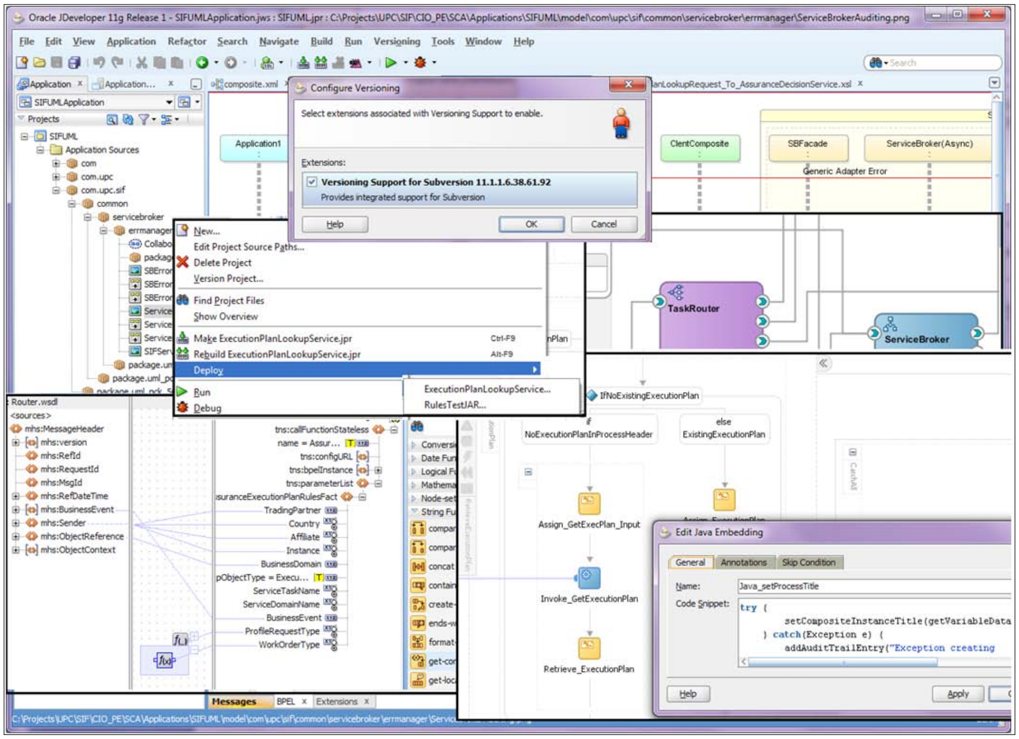
Service creation – Object and XML Design frameworks

Oracle JDeveloper is probably the most all-encompassing tool among the whole range of developers' workbenches, covering the entire development workcycle. Please see the most common steps in the next screenshot.

UML designing is the complete UML coverage for use cases, activities, classes and sequences diagrams. Sequence diagrams have an effective autoplay feature, simplifying the concept demonstration. UML diagrams are part of your SOA Suite project, which is the closest thing to your code, so you and a business analyst can sit together during prototyping, a very critical stage of the project, in order to eliminate any possible design gaps.

Needless to say that all artifacts are the subjects of the version control (subversion is displayed, but Rational ClearCase, MS Team System, and Serena Dimensions are also supported) as mentioned in the following list:

- First is the **service composition visual development** (**SCA**) of all components such as BPEL, all types of rules (discussed previously in *The Oracle Rule Engine section*), SCA mediators, and Human Workflow routines. The contact-first concept is fully supported, you can create BPEL or Mediator right from the WSDL and deploy it straight from JDeveloper on any deployment target in your preconfigured servers list. The standard Java development, including embedded coding with outlining and code insight, is greatly simplified, but some insight features are probably better to switch off, as code completion can be annoying for some. The BPEL visual designer is brilliant in the latest versions and the number of crashes are considerably reduced.
- Second is the Visual XML development; XSLT mappings are really mature and the number of XML-related function is substantial. Still, when some complex XML manipulation is added into the source, visual development can become unavailable. That can be irritating; however, in the defense of JDeveloper, this is true for some other XML development tools such as Altova's XMLSpy.



Configure Versioning

Select extensions associated with Versioning Support to enable.

Extensions:

☑ **Versioning Support for Subversion 11.1.1.6.38.61.92**
Provides integrated support for Subversion

Help OK Cancel

Edit Java Embedding

General | Annotations | Skip Condition

Name: Java_setProcessTitle

Code Snippet:

```
try {
        setCompositeInstanceTitle(getVariableData
    } catch(Exception e) {
        addAuditTrailEntry("Exception creating
```

Help Apply

JDeveloper OFM capabilities

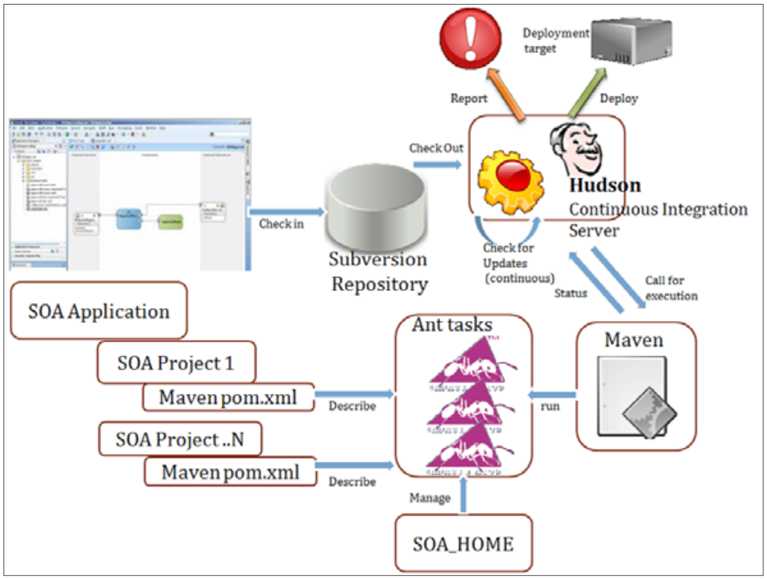
When discussing JDeveloper, we would really like to avoid the holy wars of JDev versus Eclipse versus NetBeans. It has no relation to the SOA patterns, the subject of this book. We have to admit that JDeveloper is the perfect tool for SOA Suite and almost any other type of development, except for the **Oracle Service Bus** (**OSB**). We either use the OSB console or the Eclipse plugin (OEPE) to construct our OSB flows. Is this a drawback? Certainly! A single tool that covers SCA/SOA Suite and OSB would unquestionably simplify our life, but we have to remember that EBF (SCA composites and orchestration) and EBS (service bus) are different frameworks; they are almost different universes. We already have some OFM 12*c* products at our disposal and, hopefully, with the full release of OFM 12*c*, we will have these tools merged.

One of the problems of using Eclipse is that it is as good as the quality of the plugins we have, and from our experience, we know that OSB and WLS admin pack plugins are really good; the old plugins for BPEL worked only for quite simple flows. So, splitting OSB and SCA development may not be that bad after all.

The last piece in the development and modeling stack is SQLDeveloper, which in many ways is better for DB development and administration than JDeveloper.

Service development – automated test and deployment

The open OFM development architecture allows for the creation of the **Continuous Integration** (**CI**) framework, where the central role will be on the Hudson build server. Please see the following figure:



Hudson is an open source tool supported by Oracle. It's a continuous integration server capable of executing practically any scripting tasks, such as Ant targets, Java compiler tasks, and Maven goals, and in doing so, performs almost any type of assembly and deployment. It can also be easily integrated with any type of version control system and equipped with a powerful scheduler. All connections can be secured and dedicated credentials can be established for every subsystem (code repository or deployment targets). All Hudson's projects can be established in a hierarchical way with the master project on the top and subtasks below. The build process itself can be distributed as Hudson supports the master-slave topology for distributed assembly and compilation. Master (central Hudson server) and Slaves (agents performing build tasks) can be dispersed between on-premise and cloud(s) installations. The previous figure illustrates the simplest (yet most effective) way of implementing this framework.

Maven (`http://maven.apache.org/what-is-maven.html`) and its plugins will be necessary when your project has different heterogeneous components—not only SOA Suite SCAs but also DB schemas, separate Java modules, and Oracle **Meta Data Services** (**MDS**). In most cases, Ant can do all the jobs required for the task's execution against the targets. The mandatory part here is the versioning system (subversion) in connection to JDeveloper and Hudson, deployment server (probably WLS), and Hudson CI server. When a developer submits the project's code, Hudson detects the change in versioning system and run the scripts associated with these projects. Alternatively, it will be more effective to schedule these Hudson tasks at a certain interval. Developers must observe and respect the code build and submit culture diligently. When code is assembled and compiled successfully, Hudson can proceed with the deployment on the JIT server and execute the test scripts. If an error occurs during the assembly and build or during testing, a report will be generated. Build Manager and the developers will always see the status on the Hudson console as `sunny` or `rainy`.

This framework probably does not produce an SOA pattern, but its role together with the Service XML and Object development frameworks in maintaining the contract-first development paradigm is crucial. That's the core of the Agile development, and without establishing a proper development and deployment culture, we will not be able to move further toward complex Orchestration and Service Brokering.

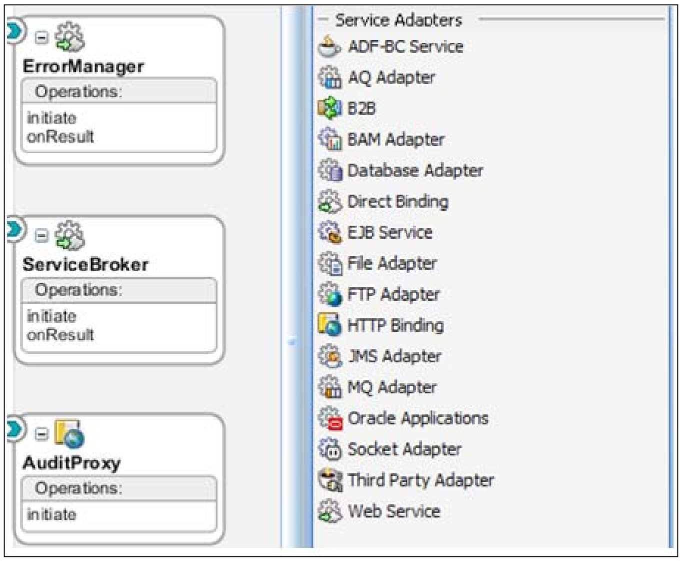
Establishing the adapter framework

When discussing AIA, we emphasized the role of the adapter framework in establishing the compound SOA pattern named Federated Endpoint Layer (in AIA terms, ABCS). This pattern is composed of several atomic patterns, where they directly govern the following canonicalizations:

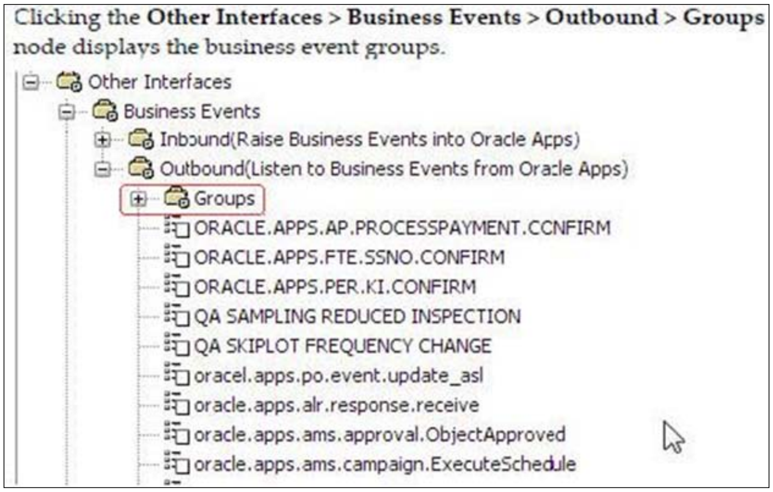
- **Canonical Expression**: This indicates expressing contracts' capabilities (operation) in a consistent and comprehendible way, improving discoverability, and consequently, reusability.
- **Canonical Protocol**: This means reducing the number of communication protocols to the optimal level. It is essential to maintain composability.
- **Canonical Schema**: This denotes establishing canonical data models, and reducing transformations and as a result, improving performance and the overall reusability.

According to the AIA methodology, the adapter framework is usually realized by BPEL. This advice has very strong historical reasons because the very first version of BPEL had dozens of different connectors presented as partner links to DB, Files, FTP, MQ, and so on (see the following screenshot). Adapters are arguably the strongest part of Oracle BPEL. The framework is extremely extensive and provides the possibility to link to third-party adapters.

Oracle Application Adapters have also been one of the strongest selling points for BPEL. With these types of adapters, we can natively communicate with OEBS PL/SQL concurrent APIs, shadow tables, and custom interfaces:



The latest releases allow adapters to subscribe to the OEBS business event or a group of events (see the following screenshot). We will discuss how it fits the whole SOA paradigm in *Chapter 6, Finding the Compromise – the Adapter Framework*. What's important now is to stress that Oracle looks like a primary platform for the adapter framework.



Clicking the **Other Interfaces > Business Events > Outbound > Groups** node displays the business event groups.

- Other Interfaces
  - Business Events
    - Inbound(Raise Business Events into Oracle Apps)
    - Outbound(Listen to Business Events from Oracle Apps)
      - Groups
      - ORACLE.APPS.AP.PROCESSPAYMENT.CONFIRM
      - ORACLE.APPS.FTE.SSNO.CONFIRM
      - ORACLE.APPS.PER.KI.CONFIRM
      - QA SAMPLING REDUCED INSPECTION
      - QA SKIPLOT FREQUENCY CHANGE
      - oracel.apps.po.event.update_asl
      - oracle.apps.alr.response.receive
      - oracle.apps.ams.approval.ObjectApproved
      - oracle.apps.ams.campaign.ExecuteSchedule

From a vendor-neutral perspective, we would not advise you to entirely lock on the BPEL realization. As repeatedly mentioned, BPEL is the orchestration tool. Fast service collaboration can be affected by latencies of this platform. On the other hand, data format/data model transformations and protocol bridging can be easily implemented on Oracle Service Bus. These two patterns are the building blocks of the Service Broker pattern, together with asynchronous queuing, which is part of the classic Enterprise Service Bus. Thus, if your adapter does not require complex data extractions with possible multicommits from DBs with degraded performance, all you need is to transform and validate the inbound XML and consider OSB as your adapter layer. However, the general rule is always the same—minimize the adapter framework in your enterprise as much as you can. The standardization of service contracts across the Service Inventory is the primary goal.

Providing orchestration – enterprise business flows

With all the complexities of this layer and numerous SOA patterns encapsulated into it, the technical foundation of this framework can be described precisely as Oracle SOA Suite. In more detail, it consists of Oracle DB (XE; standard or Enterprise is also possible with some configuration amendments regarding max cursors and processes) with SOA DB schemas installed (by the Repository Creation Utility, RCU), and Oracle WLS with `soa_server`, a domain dedicated for orchestration.

From a component perspective, Oracle SCA consists of the following:

- BPEL
- Decision Service
- Mediator
- Human Workflow

Interestingly, after five years of exploitation, SOA Suite 11*g* is sometimes called BPEL, where BPEL actually represented just 25 percent of the overall functionality. In fact, it plays a clearer role now, acting as the glue between other services and the components. SOA Suite itself presents a very good example of the separation of concerns. Probably the most disputed component is Mediator. Its role is sometimes mistaken for the **lightweight service bus**. We do not believe that the weight matters here; Mediator can also be dehydrated as it's stateful, and so it's not that lightweight. Mediator relieves the BPEL processes from the implementation of complex if-else / routing functionality: parallel, sequential, static, dynamic, and rule-based.

> The execution of parallel rules requires the enqueuing of messages in the DB dehydration store. As a DB is involved, performance tuning and performance monitoring are required.
>
> Oracle Mediator's dynamic rule-based routing is an effective feature, but we are limited only by an asynchronous MEP (nonsynchronous or one-way MEPs) and we cannot alter the payload (no transformation). Also, only SOAP binding is currently supported.

Thanks to the BPEL-based artifacts, we can finally focus on the grouping task-centric logic related to particular EBMs, presenting enterprise-centric business logic in a transparent and manageable way. Clearly, that's the Process Abstraction SOA patterns' implementation, which aims at the creation of a task service layer in our service inventory. A new SOA pattern-candidate has been introduced recently—Entity Linking. According to its name, the purpose of this pattern is to maintain the desirable level of the Loose Coupling principle between the isolated Entity services, giving them a possibility to natively communicate as some business relations always exist between them (`Order <-> Invoice`, `Schedule <-> Vehicle`, and so on). In this context, Mediator can be seen similar to this pattern's implementation, but this is for task services within one composite because, task services are also quite often related to a single EBO.

In the SOA modeling and analysis practice, there is one rule that concerns the functional decomposition of business logic, that is, isolate manual tasks and do not try to automate them (at least see the automation of manual tasks as a second priority). Human Workflow is actually a very elegant way to address this designing dilemma.

So, the simple conclusion about this framework is that Oracle SOA Suite covers it pretty well, following the BPEL 2.0, WSDL 2.0, XSD 1.1, and XSLT 2.0 standards. Some XPath 2.0 functional arrears are supported as well. Therefore, we can give positive, as well as practical, answers for all questions in the following table dedicated to the EBF aspects from *Chapter 1, SOA Ecosystem – Interconnected Principles, Patterns, and Frameworks*. In terms of SOA patterns, the most important composite pattern, Orchestration, is covered in great detail.

Common problem	**Pattern addressing the problem**
The isolation of business-centric (nonagnostic) services from agnostic, highly reusable components, and visually representing them in a comprehensive and manageable form	• **Process Abstraction**: SOA Suite is quite cleverly organized according to **Service Component Architecture** (**SCA**), allowing the abstraction of assembled components and hiding implementation details. • Mediator, Human Task Services, and Decision Services reduce the complexity of BPEL processes, increasing modularity and composability of complex business solutions. • Agnostic services (Entity and Utility) can be easily recognized and filtered out for separate implementations. Surely, this separation cannot be done by SOA Suite alone. We have to architect our service layers cleverly.
Task services are a special kind of services, with specific requirements for the service engine. In fact, as we have four separate components, then four separate engines will probably be necessary in our technical infrastructure to support this framework.	• **Process Centralization**: Oracle SOA Suite provides all the necessary engines to support orchestrated task services. All these service models are centralized under a harmonized environment (WLS `soa_server`), covered by HA measures. Nevertheless, it must be realized that all engines (BPEL, Mediator, Rule, and ADF runtime for hardware) are a burden for the infrastructure and operational support, so please model your services carefully and avoid hybrid models in order to save costs and effort. • Back to the discussion of development tools unification; the physical isolation of orchestrated task services from other infrastructures does not justify the IDEs' disparity, but at least it explains the complexity of unification. • Another type of centralization pattern covered by SOA Suite is Rules Centralization. Decision services, connected to the central rule repository, allow us to abstract business rules and make our task services more flexible. The risks associated with the realization of this pattern were discussed earlier in *The Oracle Rule Engine* section.

Common problem	Pattern addressing the problem
What makes the orchestration layer quite special is the necessity to persist process data during an inactive state for days, or even weeks.	• **State Repository**: We already discussed this pattern when talking about DBs. This is a crucial pattern for the whole orchestration, and the point here is not the performance deficiencies that are quite natural in (almost) any persisting technique. The question here is the safekeeping of storage and data consistency. • Choose your storage purge strategy wisely; Oracle supplies us with online and offline purge scripts, but local DBA's attention is highly advisable. And, of course, back up, back up, back up… • Recently (in the PS6 release), Oracle presented the **Table Recreation Script** (**TRS**) that can be used as a corrective action in addition to the standard purge script. You should also be aware that purging strategies could be different for different DB vendors (MS SQL Server or DB2). • Last but not least, select your SOA audit level appropriately. The trace/debug mode in production is probably not the best idea.
Even with the implementation of transaction management (Atomic Service Transaction SOA pattern), it is not always possible to maintain long-running transactions such as ACID.	• **Compensation Service Transactions**: The SOA Suite compensation flows present the BASE transaction management that cover this pattern.

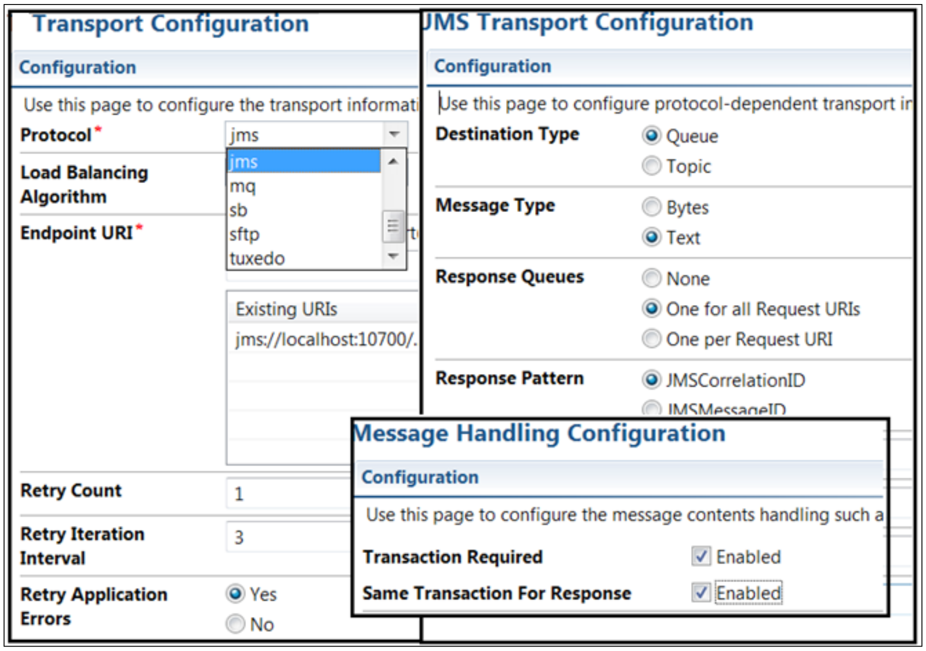
Setting up Service Bus – enterprise business services

Inherited from BEA, Oracle Service Bus is probably one of the most well-equipped commercial ESBs and is similar to the Orchestration layer. We can provide positive responses to all questions for the table under *The Enterprise Business Services framework* section in *Chapter 1*, *SOA Ecosystem – Interconnected Principles, Patterns, and Frameworks*.

Also compound, OSB shares several SOA patterns with Orchestration:

- We also need Rule Centralization in ESB, and although the connection to the decision service in OSB will require a bit more effort than that in SOA Suite, we can use the Rule Service SOAP API to call the decision service prior routing, get the endpoint URI based on our conditions, and route accordingly (in this case, we could use the SOA Suite SCA component-wrapping decision service, but this is not always advisable). Alternatively we can use the RE Java API to fire the rule/ruleset from the Java callout.
- Data Model Transformation is also common to ESB and Orchestration. As both compound patterns (and frameworks) are candidates for the adapter framework, the implementation of transformations is almost inevitable, but it is better to be done with them in ABCS.

Asynchronous Queuing and Reliable Messaging are the strong sides of OSB, as shown in the following screenshot:



Transport Configuration

Configuration

Use this page to configure the transport informati

Protocol* jms

jms
mq
sb
sftp
tuxedo

Load Balancing Algorithm

Endpoint URI*

Existing URIs
jms://localhost:10700/.

Retry Count 1

Retry Iteration Interval 3

Retry Application Errors ● Yes ○ No

JMS Transport Configuration

Configuration

Use this page to configure protocol-dependent transport ir

Destination Type ● Queue ○ Topic

Message Type ○ Bytes ● Text

Response Queues ○ None ● One for all Request URIs ○ One per Request URI

Response Pattern ● JMSCorrelationID ○ JMSMessageID

Message Handling Configuration

Configuration

Use this page to configure the message contents handling such a

Transaction Required ☑ Enabled

Same Transaction For Response ☑ Enabled

OSB transport configuration

The classic ESB SOA composite pattern has more atomic patterns inside, such as Event-Driven Messaging (actually, Event Delivery Network in the Oracle realization is more SOA Suite-oriented), but Service Brokering and Intermediate Routing will be our main discussion subjects in *Chapter 4, From Traditional Integration to Composition – Enterprise Business Services*.

Discovering enterprise – enterprise service repository

With so many technical layers, frameworks (we are focusing on a core ten, but that's a very simplistic model), and components, even in a single composite, it is highly important to keep control centralized on all the projects' artifacts; therefore, Oracle supplies all OFM products with **Metadata Services** (**MDS**) by default. This infrastructure with the support of customization is common to all types of artifacts, initially stored locally in all 11*g* SOA projects in order to avoid clashes. The common location can be configured on the application level by setting the correct path to the shared MDS location `$application_home/.adf/META-INF/adf-config.xml`, as follows:

```
        <metadata-store class-name="oracle.mds.persistence.stores.file.
FileMetadataStore">
         <property value="C:\Users\SergeyPopov\workspace\CIO_PE\SCA\MDS"
              name="metadata-path"/>
         <property value="soa" name="partition-name"/>
        </metadata-store>
```

Of course, you can set in a more clever way as `${oracle.home}/integration/`... Certainly, you could have several metadata stores defined in `adf-config.xml`, each with its own metadata storage usage namespaces and paths. The storage in the previous example is file-based, but DB can be used as well with some additional benefits (for instance, faster and more complex runtime queries). So, in general, it looks like some sort of version control for the following:

- BPEL, BPMN (`http://www.oracle.com/ocom/groups/public/@otn/documents/webcontent/172298.pdf`)
- XML-related artifact such as XSLT, XSD, XQuery, any XML fragments, and XPaths
- Human Workflows
- Business Rules
- Web artifacts such as Portlets, JSF pages, and ADF components

The beauty of this approach is that you can reference and access your MDS-shared artifacts using the `oramds:/apps/`... path from any SOA components (BPEL, Mediator, and so on) for **domain value maps** (**DVM**), policies, XML mappings, and so on. For instance, here is an example of fault policies binding in the SCA project's `composite.xml`:

```
     <property name="oracle.composite.faultPolicyFile" type="xs:string"
        many="false" override="may">oramds:/apps/OraSOAPatterns/
faultPolicies/fault-
```

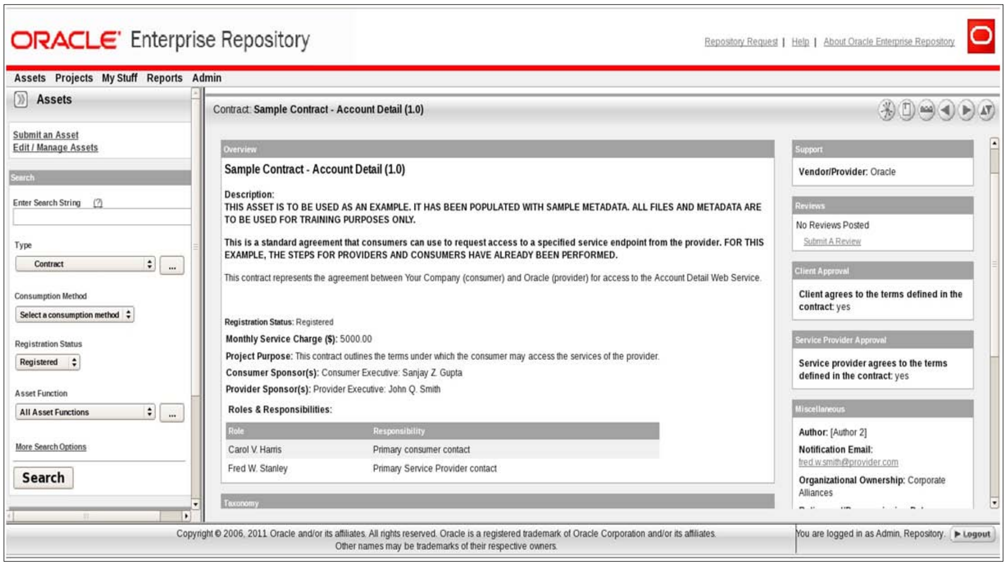
```
        policies.xml
      </property>
      <property name="oracle.composite.faultBindingFile" type="xs:string"
        many="false" override="may">oramds:/apps/OraSOAPatterns/
faultPolicies/fault-
        bindings.xml
      </property>
```

MDS is actively used not only in JDeveloper in conjunction with SOA Suite, but also in the Oracle ADFs MVC patterns' development, the WebCenter Web 2.0 development platform, and Fusion applications. Fusion Middleware control provides complete management capabilities for MDS repositories such as registration, deployment artifacts, migration, exporting, labeling, versioning, backups/recovery, and deletion. The same management functionality is also available through WLST.

Despite all the brilliant MDS repository's design and runtime features (even Design Time @ Runtime lookup/discovery capabilities), it tends to be more of a developer's tool than the enterprise service lifecycle support suite. To some extent, it can be seen as a UDDI, which is also brilliant for Service metadata description and discovery. However, it can mostly be viewed through programming interfaces built into UDDI APIs, providing a little support for metadata classification, taxonomy, analytics, and end-to-end control. Oracle has Enterprise Repository to cover all these requirements. OER is a part of Oracle SOA Governance Suite, natively connected to MDS, UDDI instances, and the developer tools. It would be quite difficult to find a single element of service lifecycle that is not covered by OER. Please see the following list that details how OER covers common SOA Governance requirements:

- First is Code Compliance Inspection with automated design-time compliance evaluation against WS-I interoperability tests.
- Next is the Native Contract support, helping negotiate terms of use between the service consumer and provider (see the figure under the *Establishing the Adapter framework* section). Both the technical and SLA aspects of service contract are covered in great detail. We probably do not need napkin drawings anymore. This capability together with the Code Compliance Inspection is covered by the AIA harvester and workbench.
- Once the contract and related Trading partners are defined, every single step of the service lifecycle can be clearly defined.
- Business and technical policies can be defined and centrally stored in support of the defined service lifecycle.
- The lifecycle service events can be recognized, and automated notifications can be delivered to access subscribers.

- What if we have legacy services or bulk-delivered projects? The Asset Harvesting tool can discover the artifacts and populate OER. Harvesting also supports clustered environments.
- Most of the OER features are accessible from SOA IDEs.
- The bi-directional synchronization with UDDI registries.
- Oracle BI Publisher supports a whole range of OER standard and custom reports.



Contract: Sample Contract - Account Detail (1.0)

Sample Contract - Account Detail (1.0)

Description:
THIS ASSET IS TO BE USED AS AN EXAMPLE. IT HAS BEEN POPULATED WITH SAMPLE METADATA. ALL FILES AND METADATA ARE TO BE USED FOR TRAINING PURPOSES ONLY.

This is a standard agreement that consumers can use to request access to a specified service endpoint from the provider. FOR THIS EXAMPLE, THE STEPS FOR PROVIDERS AND CONSUMERS HAVE ALREADY BEEN PERFORMED.

This contract represents the agreement between Your Company (consumer) and Oracle (provider) for access to the Account Detail Web Service.

Registration Status: Registered
Monthly Service Charge ($): 5000.00
Project Purpose: This contract outlines the terms under which the consumer may access the services of the provider.
Consumer Sponsor(s): Consumer Executive: Sanjay Z. Gupta
Provider Sponsor(s): Provider Executive: John Q. Smith

Roles & Responsibilities:

Role	Responsibility
Carol V. Harris	Primary consumer contact
Fred W. Stanley	Primary Service Provider contact

Vendor/Provider: Oracle

No Reviews Posted

Client agrees to the terms defined in the contract: yes

Service provider agrees to the terms defined in the contract: yes

Author: [Author 2]
Notification Email: fred.w.smith@provider.com
Organizational Ownership: Corporate Alliances

Copyright © 2006, 2011 Oracle and/or its affiliates. All rights reserved. Oracle is a registered trademark of Oracle Corporation and/or its affiliates.
Other names may be trademarks of their respective owners.

You are logged in as Admin, Repository.

Oracle Enterprise Repository is a vital element of any service inventory, ensuring runtime and design time discoverability and, ultimately, maintaining composability. Really, what good will even the best of our services do if nobody knows of their existence and capabilities? Or, how do we know that the service is really good if we do not know its vital metrics and runtime statistics? How can we assure service consumers about a service's vitality without providing stress test results and a description of test harnesses? How can we collect money for service usage? How can we centralize our business policies?

With or without OER, we will have to answer all these questions. The realization painfulness will be directly proportional to the size (and complexity) of our enterprise: a small company with 10 employees will probably do well with one Excel spreadsheet, but any sizable firm with IT staff of about 20 with the same spreadsheet will constantly be in a "Oh shhh…!" situation.

Again, OSR, OER, UDDI, and MDS are only part of the complete Governance suite and what Oracle can offer.

Service governing – monitoring, error handling, and recovering

A complete governance infrastructure includes the following components of Oracle's SOA Governance Suite in addition to OER and OSR:

- Runtime security policy enforcement (OWSM)
- Service monitoring (Oracle Enterprise Manager SOA Management Pack Enterprise Edition)
- Business Activity Monitoring (BAM)
- The Centralized Fault Management facility (partly Error Hospital in SOA Suite)

Here we would like to reiterate that you do not need to implement *all* the elements of the governance infrastructure from the very beginning. It's definitely expensive, depends on your company's size, and is ultimately useless if you do not have a clear SOA strategy in mind (building a strategy was discussed in *Chapter 1*, *SOA Ecosystem – Interconnected Principles, Patterns, and Frameworks*). Although some components will be prepacked, like Error Hospital as a part of SOA Suite, a separate purchase won't be necessary. It's part of a policy-based fault management framework (yes, another one) within SOA Suite, so we cannot consider it as enterprise wide. Nevertheless, you can assign custom actions to your error conditions and use retry and notification mechanisms. Some effort will be required to make it common for several frameworks, and we will discuss this in *Chapter 8*, *Taking Care – Error Handling*. No wonder Enterprise Repository will be an essential part of the unified Error Handler.

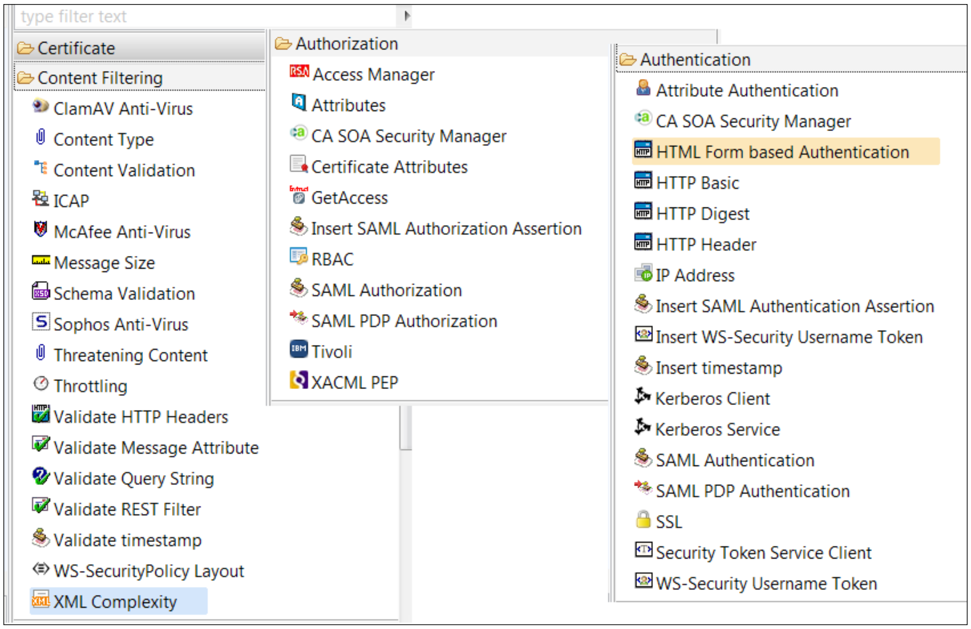
All together, these elements of the Governance framework contribute to the realization of discoverability and the Composability principles. More about governance infrastructure can be obtained from the Oracle documents at: `http://docs.oracle.com/cd/E28280_01/doc.1111/e16581/install.htm#sthref9`.

Securing service interactions – Security Gateway

This is the last framework to discuss, both graphically (the seventh layer in the very first figure in this chapter) and chronologically, according to Oracle's purchase history, but undoubtedly, a highly critical one. From the SOA realization perspective, this framework should be based on the ESB, as we discussed earlier, because of its message filtering, screening, and routing functions. Indeed, it was for a quite long time, where OSB played the role of a Service Gateway and Oracle Web Service Manager was responsible for policy enforcement, providing a common gateway and individual agents to secure service interactions.

This approach is still valid and you can use it in your enterprise between the internal SOA domains, but apparently, some more security-related features have to be added for Message Screening (Content Filtering), Authentication, and Authorization.

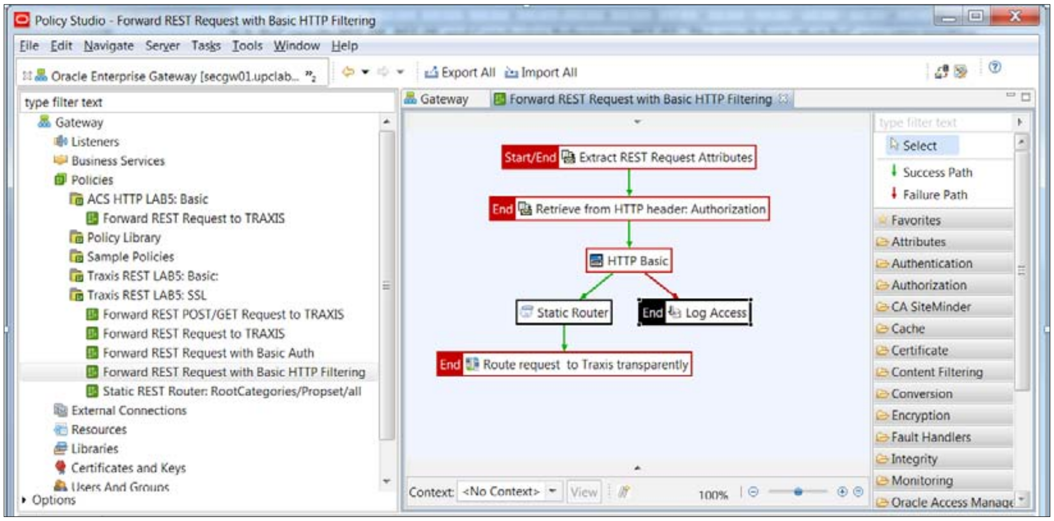
The Oracle API server has all these requirements nicely covered. See the following screenshot:



Oracle API Gateway Policy Studio

Originally from Vordel, this gateway has quite extensive workflow capabilities (see the following screenshot) and is specifically designed to operate in very hostile environments, such as **DeMilitarized Zone** (**DMZ**). The application is well-layered and runs as a managed code on JVM, but no WLS or OSB is required. So it's very lightweight on one hand and reasonably protected from buffer stack overflow attacks by JVM features on the other. Surely, the latest Java security updates must be applied promptly as certain risks related to JVMs' security holes will exist, but this calculated risk is justified by the possibility to run on the newest versions of OS and JDK as unmanaged gateways, because versions sometimes lag behind due to longer implementation time. Obviously, older OS releases are exposed longer to the evaluation of vulnerabilities, giving more opportunities to hackers. Although we do not really want to participate in the managed versus unmanaged code clashes, the unmanaged Intel Expressway Service Gateway in our tests produced a stunning 10 K tps 3 K SOAP message on two really modest dual core CPU servers.

At the same time, Oracle Gateway was almost as good, showing pretty similar figures. Thus, the choice is yours.



Oracle API Gateway Policy Studio

The option to choose the physical realization of this technical layer again comes from the following principles, opening the door for vendor-neutral security implementation. What must be strictly observed though is the compliance with common security standards:

Requirements	Standards

Transport layer security	SSL and TLS
	`http://tools.ietf.org/html/rfc6101`
	`http://tools.ietf.org/html/rfc5246`

Confidentiality enforcement XML Encryption	XML Encryption `http://www.w3.org/standards/techs/xmlenc#w3c_all`
Integrity, Non-repudiation, and origin assurance	XML Signature `http://www.w3.org/TR/xmldsig-core/`

WS-Security	(SAML, Kerberos, X.509 token profiles)
	`https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=security`
	`http://www.ietf.org/rfc/rfc4120.txt`

Requirements	Standards
Generic application of rules and conditions	WS-Policy `http://www.w3.org/Submission/WS-Policy/`
Subset of the policies, related to security	WS-SecurityPolicy `http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/ws-securitypolicy-1.2-spec-os.html`
Brokered trust management (claims, assertions, tokens)	WS-Trust `http://docs.oasis-open.org/ws-sx/ws-trust/200512/ws-trust-1.3-os.html`
Governing secure context exchange (tokens during handshake, and so on.)	WS-SecureConversation `http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512/ws-secureconversation-1.3-os.html`
XML-based Access Control Language	XACML `https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml`
Key management specification	XKMS `http://www.w3.org/TR/xkms2/`
Cryptography standards	PKCS#1, PKCS#7, PKCS#12 `http://www.emc.com/emc-plus/rsa-labs/standards-initiatives/public-key-cryptography-standards.htm`
Securing rich/multipart content	S/MIME `http://tools.ietf.org/html/rfc2311`
Open delegated authorization standard	OAuth `http://tools.ietf.org/html/rfc6749`

It's not that uncommon when parties are unable to communicate having the best-of-breed Secure Gateways in place just because of incompatible encryption protocols or the unsupported enveloped/enveloping signature concept.

One very interesting topic here is the necessity of the EBS layer for external communications, handled by a security perimeter (essentially, Service Bus as well). We will not discuss it in this chapter, as it is dedicated only to products linking to the abstract frameworks, but rather keep it for future discussion.

The names of the security standards supported by the API Gateway are very similar to the functionalities we expect from it. With this product in the portfolio, Oracle now covers all eight generic SOA Security patterns.

Summary

Oracle has come a long way from ad-hoc apps that link to enterprise integration and finally, to the full-fledged service orientation. With no doubt, products from Collaxa, BEA, and Sun turned the single DB-company into one of the strongest Middleware players with the most advanced products in the portfolio. Importantly, we see that Oracle SOA's product stack is evolving and this evolution is guided by the open standards committees where Oracle is the one most active contributor and is influenced from partners and customers who bring business demands and challenges. At the same time, it is important for us not to follow this path blindly, trusting somebody else's strategy, but rather choose our own way of achieving the strategic goals. In this sense, Oracle is setting a good example by assembling its own and acquired products following the Composability principle and giving us a wide range of the tools, enabling service-oriented computing.

Some architects accuse Oracle of offering products with overlapping functionalities in application suites and packages. Indeed, this claim has factual context. Apparently, overlapping can hardly be avoided in a particular market's expansion strategy. But honestly, most of the enterprise architects have to admit that in our own companies, the level of redundant denormalized functionality in applications/services is sometimes far from optimal, and that's normal. Architecture is a living mechanism—it has to evolve—and as long as we do not want to get into the disastrous Big Bang approach, we have to move gradually, allowing some level of functional denormalization along the way. So Oracle does as well.

What matters is the way we normalize our functional and technical boundaries. More rationally, this could be done by applying the SOA principles and standards in identified frameworks. After all, most of the Oracle components in software packages are optional. If you do not have long-running services, go for OSB (EBS) first and turn to SOA Suite (EBF) later, when necessary. If your security requirements are particularly high and performance must not be compromised, consider the API Gateway as your single service collaboration platform with no extra layers. If you run mostly heavy-batch jobs during the night hours, give ODI and **master data management** (**MDM**) a general thought; however, bear in mind that these tools are also parts of the EDN and AIA SOA implementation. Nevertheless, what almost certainly will be presented in your infrastructure are the core ten frameworks, and as we discussed in this chapter, Oracle has good candidates to employ.

As a final example to conclude this chapter, we can look at the industry-specific case of an SOA platform realization based on Oracle products. The telecom industry is rapidly moving these days, both technologically (emerging 4 K TV + H.265 HEVC codec and to accommodate it at home -802.11 ac routers) and commercially (*any content* + *any protocol* + *any device* + *any place*). To accommodate this business shift technologically, another term was proposed—**Service Delivery Platform** (**SDP**)—and all the key telecom providers rushed to offer their platform realizations: Huawei, Amdocs, Ericsson, Alcatel-Lucent, and Telcordia. According to Gartner (market analysis 2012), Oracle is among the strongest three SDP providers. However, what is Oracle SDP? Again, Oracle's Telecom-specific products are the result of the latest acquisition (from Convergin): **Oracle Communications Converged Application Server** (**OCCAS**), **Oracle Communications Services Gatekeeper** (**OCSG**), and **Oracle Communications Marketing and Advertising** (**OCMA**).

Generally speaking, the controller and gatekeeper are the elements of the telecom-specific adapter framework, handling telephone network protocols (OCCAS: SIP, ISC, INAP, and so on) and telecom enablers (OCSG). Gatekeeper is also capable of handling home devices, such as REST and SOAP calls. Unsurprisingly, these ABCS elements are connected to the enterprise backbone, where Oracle Service Bus, SOA Suite, Service Repository, and API Secure Gateway are playing the same roles as we discussed for each enterprise framework. Oracle put considerable effort into the Contract Standardization of each component of the Telecom SOA platform, abstracting specific protocols to increase SDP Composability and demonstrating a high-level of reusability of its own assets to achieve strong merits.

Now we are ready to look at the most widely recognized SOA framework, **Enterprise Business Flows** (**EBF**), and see how SOA patterns can solve typical problems there.

3

Building the Core – Enterprise Business Flows

Service Orchestration is one of the terms that is most commonly associated with the Enterprise Business Flows framework and SOA implementation in general. At the same time, Orchestration is one of the two most common compound patterns (the other is ESB) employed for addressing reoccurring problems related to application integration. In this chapter, we will discuss these common problems and the ways of mitigating them using different patterns that are related to Orchestration, and find ways to turn integration into service collaboration. Not all debated patterns will be related to the Orchestration realm directly.

However, as long as they are quite universal and applicable to every framework, we will put them here; this is because the Orchestration layer can be a good demonstration ground to begin with this exercise. Some basic SOA Suite skills (for 11*g* composites: BPEL, Mediator, and RE) and some Java skills (MDB and JMS) would be useful, as we simply do not have enough room here to provide a complete technical tutorial. The main focus in this chapter will be on patterns, ensuring reusability and composability. Therefore, architecting the agnostic Composition Controllers and subcontrollers will be the most interesting part.

Another reason to start with Orchestration first is that due to the popularity of Oracle BPEL Process Manager (initially Collaxa and later 10*g* and 11*g*), in the last 9 years, quite a few enterprises have maintained a considerable amount of orchestrated services with different degrees of service orientation. It is not too common for an enterprise to start something from scratch these days (such as the *St. Matthews Hospital* example in *Oracle SOA Suite 11g Handbook*, *Lucas Jellema*, *McGraw-Hill*), practicing a pure top-down approach. Thus, the patterns related to Service Inventory Analysis and Modeling will be essential for understanding the presented example.

Oracle SOA's dynamic Orchestration platform

Orchestration is always related to task-orchestrated services and non-agnostic workflows that are devised for covering a single (but complex) business operation, which we discussed in *Chapter 1*, *SOA Ecosystem – Interconnected Principles, Patterns, and Frameworks*. This master service is a chain of service invocations and simple request-responses and/or complex transactions that involve several services at a time (ACID- or BASE-style). As mentioned earlier, this master service fulfills the runtime role called Composition Controller, which controls the service's composition members or other subcontrollers. This is a pure runtime role, and the next time you decide to include this master-controller into a more complex composition, the new role will be different (becomes a subcontroller). We also know that Composition Controller does not always start the service interaction (or service activities) within the composition. More often than not, a composition is commenced by a composition initiator that is not part of the composition itself.

In relatively static business models (big savings/deposit banks and some governments institutes), compositions could be pretty static as well with fixed data models and business rules. Some conservative businesses can rely on traditional BPEL processes, especially if geographic areas are well set and related service endpoints are permanent. As we all know, the static business model is rarely used these days, even in government structures. This is because there are many aspects such as technological, commercial, and organizational that shake our organizations. Interestingly, even small companies with a single line of business also like to stay more and more agile and adaptable in modern competitive surroundings. Task services that are placed at the top of the whole IT stack, most closely related to concrete operations and business people, must not pose bottlenecks for adapting the enterprise principle. How can we achieve that? Yes, you are right, all eight principles must be applied, but with some preconditions as follows:

- Composition members must logically encapsulate a concise business context and introduce a standard service contract (among the other seven principles), that is, stay composable
- Controllers that compose these members into a business process fabric must be agnostic

The second outcome is possible when all the business rules are abstracted from Composition Controller (remember that theoretically, we are compromising the controller's autonomy), and it can be accessed and consumed by the controller dynamically using the message context. Following this logic, we can assume that Composition Controller can recursively act as a subcontroller for just another instance. That's it! Just by changing the rules, we can change the sequence of the invocations and consequently implement new processes on the fly. So simple, isn't it? Who would think about it? Here, we can conclude the chapter and probably the entire book.

> Again, as you can see, suggestions are based on common sense; most certainly, these logical outcomes are not new to you. We now see our task as a demonstration of Oracle SOA Suite's capabilities to implement the suggested outcomes.

Talking about it seriously though, dynamic Composition Controllers are not new at all, and we have all coded and implemented them many times. However, here we are going to choose examples that are specific to Orchestration, where we have two distinct properties: processes should be long running and asynchronous and the controller must be truly agnostic for dynamically brokering other service members. Therefore, the problems we are facing are obvious too:

- There is no problem in abstracting the rules, centralizing them, and accessing them dynamically. What we must remember is that these rules could evolve while the process is running for days and weeks. Thus, rule versioning must be handled seamlessly by Composition Controller together with Rule Engine. A single process cannot start using one ruleset and finish with another.
- The process can fail for many reasons. With a static non-agnostic controller, you can set precise compensations and error handler(s), anticipating concrete faults. An agnostic controller alone has no idea about what to do in error situations, which means that you will again need to employ Rule Engine in order to compensate for the error.
- A compensation itself is rather specific as we cannot always apply generic compensation flow(s) in a generic controller. We should also be careful in propagating the error to the upper level (initiator or master-controller) for many reasons.

- Actually, abstraction and centralization of the rule is the problem as we technically present a single point of failure, and the infrastructure must provide a redundant solution for this.
- While abstracting the controller itself, we must not forget that these types of controllers are more suitable for synchronous and fast-running compositions. Long-running compositions will require persisting the process state, and agnostic controllers are not the best candidates for this. Actually, BPEL was invented to solve this problem, presenting a task-orchestration service as a non-agnostic Composition Controller and one of the means for the implementation of the Process Centralization SOA pattern.

So, why would we step aside from the Process Centralization pattern based on a very comfortable and easy-to-use BPEL and then try to decentralize the processes using an agnostic controller? The following practical example based on a counterfeit, yet quite realistic, telecommunication primer will demonstrate the reasons for this approach. Jumping ahead a little, we would like to substantiate that we are not actually breaking the concept of the Process Centralization pattern with the implementation of an agnostic controller, as we are enforcing the process of centralizing composition's rules. Secondly, we are not going to abandon the idea of using BPEL as the Orchestration language/engine or reinvent BPEL's syntax. Instead, we will use SOA Suite tools (11*g* PS6) in this example; since SOA patterns have very generic solution approaches, you can roll it up on any other platform.

The telecommunication primer

It is difficult to find a business sector with more rapidly changing commercial and operational environments than telecom these days. Reasons such as active mergers of mobile operators with cable and content providers, new technologies and geographic diversities, and high demands from customers for new products set the highest requirements for the agility and adaptability of business processes. Intense competition will easily put any company of any size out of business if it cannot keep up with these types of shifts. Let's see how the agnostic controller can address these challenges, and at the same time, mitigate the problems described in the bulleted list in the previous section.

Basic facts about the telecommunication enterprise

Some of the basic facts about our fictitious telecommunication enterprise are mentioned in the following table:

Telecommunication enterprise	Business domain	Governing and type of ownership
CableTelUnlimited Inc. (CTU), HQ: Brasilia	Telecommunication	Public

History of CTU

CTU was started 20 years ago as a cable company, providing TV/video for customers in one geographic area. Quite soon, through a series of acquisitions, they became a large telecom enterprise, delivering various combinations of three main traditional-for-cable-company products:

- Voice/phone (landline VoIP)
- Video entertainment
- Internet provisioning

Following this business development strategy, geographic operations were expanded all over South America during the last ten years, effectively covering ten countries and turning the company into one of the biggest telecom market players.

Currently, business operations are still cable-oriented and mobile products are not in the company's portfolio yet, but there are some plans for business expansion in this area.

Traditionally, this telecom enterprise is not a "software house", so all the development work is done by external vendors. The two main factors, namely the technically diversified products' portfolio (video/voice/Internet) and past acquisition, have shaped the company into three main departments (offices) with a relatively low level of collaboration.

Also, local affiliates in all the countries where the enterprise operates have a considerable level of freedom with regards to standards implementation; additionally, the burden of legacy applications is also quite substantial. Some legacy applications have been in operation for almost ten years due to budget constraints or lack of life cycle governance, so the level of adaptation to the current business environment is below expectations.

Technical infrastructure and automation environment

CTU has a massive inventory of products from various vendors. In operational HQ, these products are unevenly distributed between three main operational departments:

- Network (responsible for Internet and voice), handled by the CNO
- Technology (content delivery and Video on Demand), handled by the CTO
- IT and internal IT systems (responsible for order management and provisioning, billing, and customer management), handled by the CIO

The last department was also involved in providing internal system integration to some extent to the other two departments, especially the CRM domain. Only a few solutions were developed in-house, mostly for integration and service abstraction. The number of applications in the application's portfolio is unknown but can be roughly estimated at 50 in CIO, 200 in CTO, and 100 in CNO.

The HQ's application farm is deployed, maintained, and administered on two business data centers with more than 600 virtual machines, each in clustered environments, handling multitenant and individual accesses for regional offices. The level of multitenancy is low at the present moment; about 90 percent of all VMs are country-specific, although their business logic is similar with very small variations.

Regional offices have their own local application infrastructure that supports business applications in regional offices and maintains integration with core HQ apps related to the affiliates.

It must also be mentioned that in some countries, a corporate entity has more than one affiliate, depending on the line of business. The level of an affiliate's technical efficiency varies, reflecting business proficiency. Each of the three HQ administration offices has an IT division with its own departmental structure and organizational hierarchy. There are regular meetings between the IT managers from all divisions, but outside of that, there is infrequent communication or coordination.

The vendor selection process and new product procurement routines are formalized by policies, which are specific for each department. The standard RFI/RFP process could take up to 10 months according to these policies; at the same time, the requirements for new product/projects' implementation are about 6 months. IT resources are rarely shared between departments.

Business goals and obstacles

The recent annual financial report demonstrated the best corporate earnings over the last 10 years. At the same time, a detailed analysis revealed that operational costs have increased considerably compared to the last year by almost 10 percent. The reasons can be identified as follows:

- Two new strategic products were released last year with the first stage covering one third of the countries in operations
- Some applications have been migrated from local premises to a private cloud that is built on a corporate data center

These reasons are naturally positive, but some considerable drawbacks should also be mentioned:

- New products were under development for three years and the implementation was out of budget (40 percent) and time (the deadline was moved three times).
- Due to the delay, some first-on-market advantages were lost, and during the same period, competitors managed to deliver two to four products more in different geographic areas.
- The architectural delivery models for these products were presented in the application-silo style at the last stage of the products' delivery with very limited service collaboration efforts. Most of the applications were integrated in a **point-to-point** (**P2P**) manner.
- Although migration of some applications to a private cloud was considered successful, it didn't reduce the operational cost, as none of the cloud delivery models (**Infrastructure as a Service** (**IaaS**), **Platform as a Service** (**PaaS**), and **Software as a Service** (**SaaS**)) were presented clearly. VM provisioning still takes up the same amount of time that it used to with on-premise deployment.

Considering all of the stated reasons, the Board of Directors decided to transform the IT process, aiming to initiate the following:

- Streamline the RFI/RFP process for optimizing PoC time and narrowing down the vendors list. They decided that having two approved vendors would be the common rule for all departments.
- Bridge the gaps between architectural groups in the main operational departments, and establish a new enterprise architecture office. A new **Chief Architecture Office** (**CAO**) head was appointed.
- This new department was made responsible for establishing standards, policies and design rules, and controlling E2E delivery from the enterprise architecture standpoint. Even common terminologies were supposed to be consolidated, agreed upon, and conveyed down to all the related parties, from the project management and delivery office to the coders from a number of vendors.
- The physical implementation of this approach was required to be materialized in the PoC prototype for every key delivery.

Suppose that the new CAO designates you as the lead architect for the first SOA initiative. This will be a highly visible project that your colleagues and superiors will be observing with high interest. The goal of this project will be to assess the services currently in development and to upgrade the CTU runtime platform using proven SOA design solutions in order to demonstrate that SOA can solve a series of problem areas. In support of this goal, you will be provided with funding to implement a modern enterprise service bus platform.

As the starting point of this challenging IT transformation program, the Operations System support domain is selected. Traditionally, setting up the **Order Management** (**OM**) functionality is the primary objective for the implementation of service-oriented computing within this domain.

Oracle Enterprise Business Flows SOA patterns

Challenges will first be described, and after a detailed analysis, we will formalize the patterns that are most suitable for solving the problems on hand. Not all SOA patterns related to Orchestration will be covered here; we will focus only on candidates who are most frequently (in our opinion) engaged in the practical implementation of Service Inventory. We have mentioned some of these candidates several times already—it's a Service Broker together with Intermediate Routing that acts as an agnostic Composition Controller.

> These patterns are related to ESB primarily. The telecom example presented in *The telecommunication primer* section will demonstrate that these patterns are very important for Orchestration as well. There are some more patterns that are not directly related to Orchestration, but without them, it will be impossible to build a reliable EBF framework. Our intention is not to follow the SOA pattern catalog, but rather demonstrate the practical values of the patterns.

Establishing a Service Inventory

To reuse something, first we need to establish the easily accessible inventory of reusable entities and make sure that these entities are truly reusable. From *Chapter 1, SOA Ecosystem – Interconnected Principles, Patterns, and Frameworks*, you are aware of the eight principles we must follow to achieve the four desirable characteristics. Simply put, it is your (any) solution's evaluation spreadsheet that contains eight rows and four columns for each component you are planning to evaluate. Surely, you can place your requirements at the top, but bear in mind the tangibility of your criteria. Obviously, you will have to pay some price for reusability as well. You need to balance your tactical and strategical goals wisely, but what is obvious from the presented telecom example is that strategic targets were sacrificed.

Initial analysis

Your first task as a lead architect is to meet the development team and review their design of the new Order Management System service. You must evaluate the possibility of improving the current Orchestration layer in order to reduce the operational costs (firstly by minimization of service installations). Another obvious choice would be to scrap the Order-oriented orchestration entirely and select a commercial product, **Oracle Communications Order and Service Management (OSM)**, for instance.

Operations and Business System Support (**OSS/BSS**) form the core of the telecommunication domain, handling **custom relations management** (**CRM**) and order management and procurement among other functionalities.

TMForum (`www.tmforum.org`) is the main standardization authority in telecommunication. Its eTOM specification was used for implementing the first release of the Pan-American service layer, suitable for providing a unified order management.

Physical realizations presented by three tightly coupled Orchestrated services are as follows:

- **COM**: Commercial order management
- **TOM**: Technical order management
- **SPC**: Service provisioning

Obviously, the realization is on BPEL, as the development was initially carried out on the previous Version 10*g* and then migrated and refactored for 11*g*.

Service Request is received as a delimited file, containing one or more ordered items according to the advertised CTU products. The COM module verifies a client's profile and the current products registered for this customer.

A further Service Request in the TOM module is converted into working order with lines that are sorted according to business logic and enriched with the client's information. For instance, if a client needs TV channels in higher quality over the Internet, the currently provisioned Internet bandwidth must be adjusted to these requirements. Consequently, copper or optic cables must be considered and verified; some newly advertised features cannot be supported by older versions of the **Set Top Box** (**STB**).

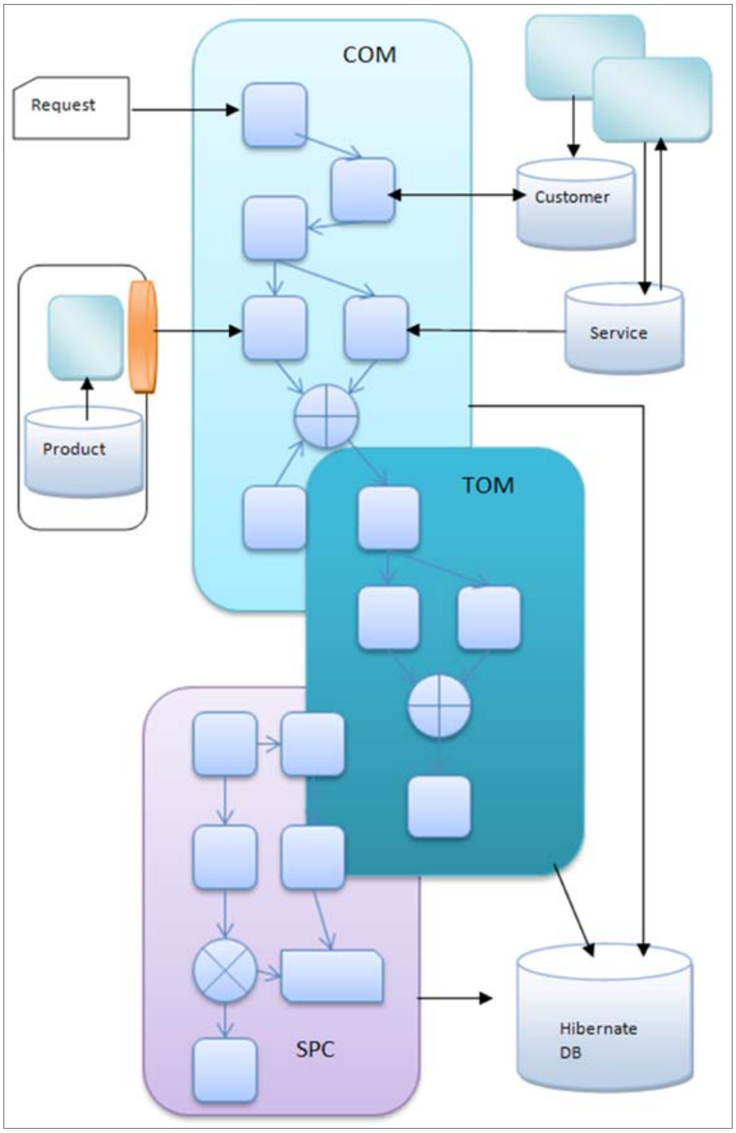
There are certain interdependencies between core telecom products such as VOIP and Internet, and Order provisioning must adhere to these rules of practical fulfillment. From the field force management perspective, for instance, you cannot send a technician to install the cable modem or STB if the cable requirements are not met. All of this must be carefully analyzed for converting the received commercial order into its technical implementation plan.

Therefore, Order lines are grouped for parallel or sequential processing and are passed to the service provisioning module for actual processing. As you can understand, every single line of commercial order could be potentially converted into multiple lines of technical tasks that are related to certain application endpoints or other compositions. Thus, there are a lot of `IF-ELSE` controls included in the last module in an attempt to cover all the possible business combinations.

This last module presents a lot of adapters and partner links to endpoint systems, acting as service providers for IP Provisioning, DTV Provisioning, registration/alteration in CRM, and so on. After completing the list of technical tasks that are related to a single Order line, its status must be returned to the caller of the process. The last successfully executed Order line will update the Order status if all the line invocations were successful. A failure in the execution of the group of invocations (or single operation in the group) related to a single Order line leads to complex compensation activities. The complexity could be very high as the number of combinations of technical tasks related to one Order line is massive. Thus, in most cases, developers just park the error order in the error queue for manual recovery. In an attempt to help operational personnel with order recovery, developers decided to log error information with excessive details. Regretfully, due to the complexity of the compositions and the number of telecom subsystems involved, there are also several logs and information related to errors that are not centralized.

To summarize this, Service Request Input is in a delimited file format where delimiters could vary, the number of lines could potentially be unlimited, and line terminators are not always in place. This is the result of some inconsistencies in the legacy system that is responsible for the construction of this file. This fact complicates the implementation of the adapter that is responsible for data format transformation at the receiving endpoint. It appears that this adapter is part of a COM process.

The development team confirmed that an XML object, constructed after the data format transformation by the receiving adapter, is in full compliance with the TMForum data model (Telco CDM Order), implementing all the declared elements. Most of these elements are not in use at the moment, thereby presenting placeholders for further business implementation. Not all elements are presented in an optional way. Therefore, the whole structure is propagated to the ultimate service provider's adapter, passing all the three orchestrated services to it without moving on to the second transformation.



Access to the **Customer** DB and **Service** catalog (commercial service term) is implemented using DB adapters, as these resources do not provide public interfaces that are suitable for the new OM system. These resources are also consumed by other applications that perform read-write operations. For a couple of hours every day, Customer DB handles excessive workload caused by some DWH OLTP operations; thus, to address latency during this period, developers increased the timeout for this particular adapter.

The **Product** catalog (and its DB) is presented as a separate service with a standard contract. The XSD for this service was autogenerated, ensuring that the data model was presented very precisely in the XML schema.

The level of standardization of the presented contract is also ensured by the fact that it was generated from an entity object based on an existing data model, which means that all the basic operations such as `GET` and `SET` are in place.

Most importantly, developers confirmed that these three Orchestration services are truly universal and ready to serve multitenant requests. It was achieved by placing complex branching logic into TOM and COM with separate branches for each country/affiliate. For some use cases, which are not formalized yet, separate placeholders are preserved for later implementation. At some places, additional transformation will be implemented in the next phases to make it rapidly available for adapters.

The development team is not concerned about the size of the composition, as they believe that the partial state deferral DB will be capable of handling the process hibernation state. Error-handling routines are covered by throwing SOAP fault errors with maximum details, as we mentioned previously.

A summary of the initial solution

Branching logic with dynamic expansion of branches depends on the root conditions that lead to an avalanche-like physical implementation. Starting from obvious business fractions, it would be very hard to stop. At the time of initial analysis, three coupled processes had nearly 20,000 lines of BPEL code (the code was written over several years by a very big consulting company). It not only makes the code unreadable (by other developers), but also unmanageable (by ops, as opening the failed flow in the **Enterprise Manager** (**EM**) console could take five minutes). By the way, did we mention that the company has had operations in ten countries, and the BPEL code branches have to accommodate each of these countries' specific logic as well?

We believe that the situation looks pretty familiar to many of us (certainly for the telecom architects), and that strong temptation to turn to Oracle OSM is only restrained by the tiny feeling that the same logic had to be deployed on another, more secluded tool, adding another silo to the CTU farm and increasing the dependency on the vendor.

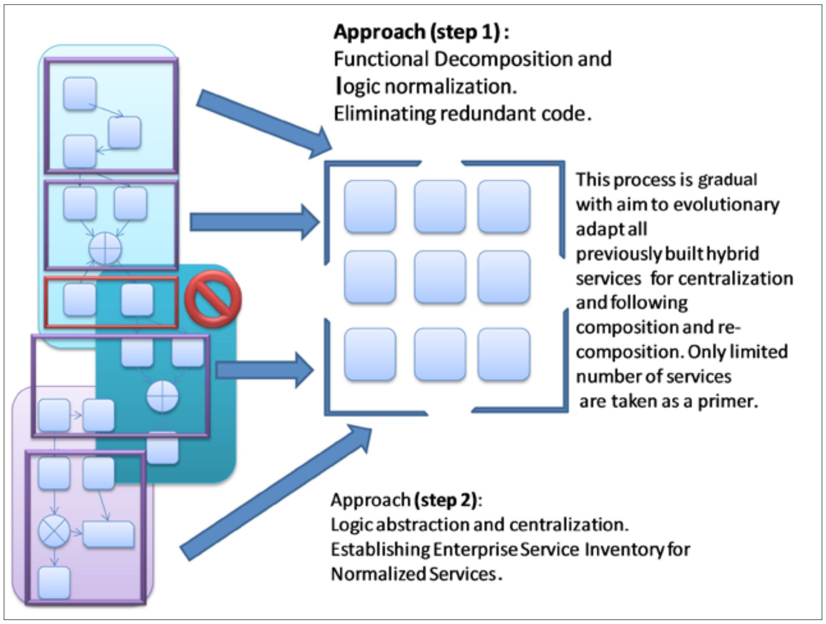
Detailed analysis – functional decomposition

The detailed analysis pattern together with Enterprise Inventory and Logic Centralization (see the second figure under the *SOA Service Patterns that help to shape a Service Inventory* section in *Chapter 1*, *SOA Ecosystem – Interconnected Principles, Patterns, and Frameworks*) shapes the boundaries of our reusable service catalog in a way that is similar to the separation of concern principle. The "divide and conquer" rule is the gradual adaptation of bulky silo-like BPEL processes to the idea of dynamic assembly and service consumption. The initial step here is to separate the automated logic (something we can code and run) from manual operations. It's done already in general, but some calls to frontend GUI services still must be isolated for proper human task implementations. The next step is to identify business cases that are related to the provisioning of core telecom products (TV, Internet, and Voice) and their bundles. Technical cases that support business cases also have to be separated and then arranged into dedicated services (ACS, STB control flows, and so on).

The common parts of all Orchestrated services (also as Orchestrated services) must be extracted and segregated as a result of this analysis. You do not have to implement them physically in the first place; however, as long we are dealing with BPEL, it would not be a problem to identify the `IF-ELSE` branches in a copy of analyzed process and take them out, storing the new flow under a new version or/and name. Eventually, we can drill down all our processes to the realization of the fact that all branches' diversity is mostly related to the geographic specialties presided by the following:

- Different endpoints of similar applications in regional installations
- Different combinations of similar services/applications due to regional business specifications

The two-steps approach based on the initial functional decomposition and following the logic abstraction and centralization is presented in the following block diagram. These patterns must be applied gradually for the most obvious `IF-ELSE` branches first, avoiding big bang implementation.



Approach (step 1):
Functional Decomposition and
logic normalization.
Eliminating redundant code.

This process is gradual with aim to evolutionary adapt all previously built hybrid services for centralization and following composition and re-composition. Only limited number of services are taken as a primer.

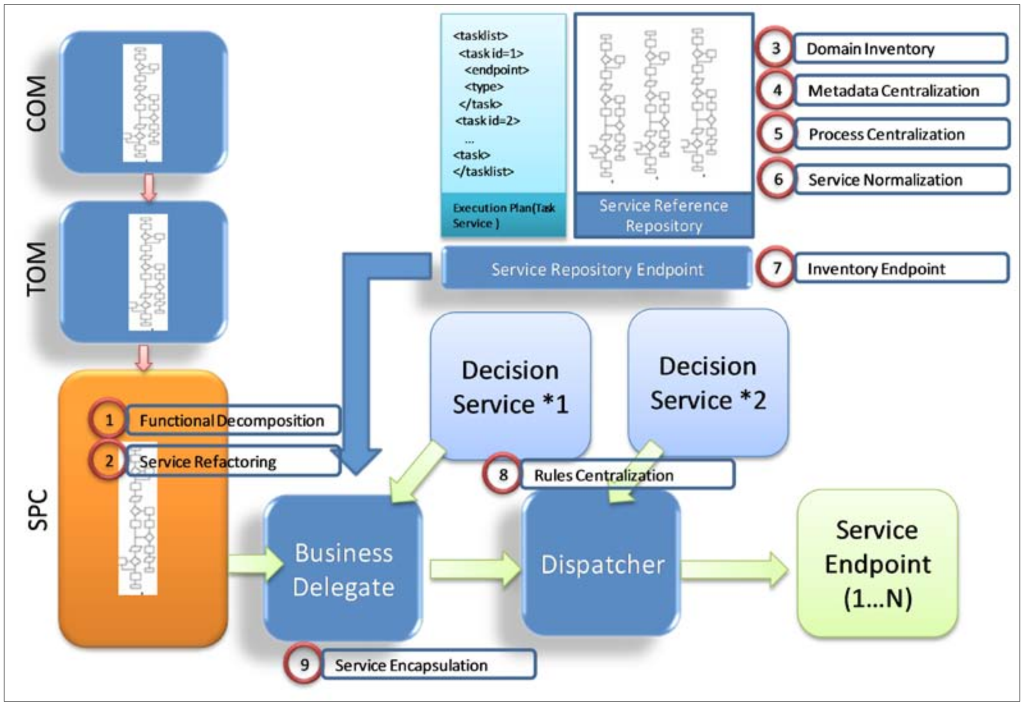
Approach (**step 2**):
Logic abstraction and centralization.
Establishing Enterprise Service Inventory for Normalized Services.

This segregation leads to the following:

- We have tree products, and each has around 20 to 25 different business flows. (We have Internet that provides high speed, ceases high speed, selects the bandwidth, and so on. We have TV to request the relevant TV package, cancel a TV channel, add a channel to a package, and so on.)
- We have ten countries where each country could have its own variation of business flows.
- In each country, we could have two (sometimes more) affiliates, providing individual or aggregated services.
- We have seven telecom-specific applications for all those countries that participate in our OSS/BSS business flows and one or two specific applications for some countries.
- Variations within an individual process for different countries are insignificant. Usually, it's just a sequence of invocations of similar applications. Anyway, it makes processes different.
- A single branch (or decomposed process with concise and formalized functional boundaries) has up to 10 invocations (one application can be called several times).

No wonder the realization of all of these in practically every single BPEL process was problematic; simple math can provide us with this: *3 products * 20 business cases * 10 countries * 2 affiliates = 1200 different combinations of single order provisioning* (this is a quite modest number; we could have more combinations).

Thus, potentially, we have 1,200 services (task-orchestrated models) to maintain in our Service Inventory, and it's definitely clear that it cannot be implemented as a single Order Management process (sorry, Order Fusion Demo). However, what's wrong with 1,000 BPEL processes? Firstly, it's quite a big number to maintain on the server. It will require quite a powerful server farm and considerable Governance efforts to control versions, DB utilization when a state is dehydrated, and significant memory resources for services running in parallel. It seems that this extreme level of functional decomposition is not good either as so many similar services will complicate Governance and error recovery, plus, again, the hardware resources' consumption will be considerable.



```
<tasklist>
 <task id=1>
  <endpoint>
  <type>
 </task>
 <task id=2>
  ...
 <task>
</tasklist>
```

Balance must be maintained during **Functional Decomposition (1)** and **Service Refactoring (2)**. Not every use case (business process) is equally demanding. You should focus on refactoring 20 percent of the services, leaving the remaining 80 percent to handle the task on hand (yes, common sense again). Do not rush to decompose the hybrid Entity services. Try to separate the agnostic and non-agnostic parts first, keeping in mind Occam's rule (`http://en.wikipedia.org/wiki/Occam's_razor`).

Quite soon, we will have our core atomic business processes, related to certain use cases of Order provisioning, decomposed and stored separately in Service Inventory. Actually, we just started shaping our Service Inventory and are not really concerned about its taxonomy. All we need to know at this moment is that we will have two distinctive layers:

- Task-orchestrated layers for decomposed and refactored BPEL processes, presenting atomic processing legs
- Entity services (Product, Service, and Customer), which for the time being are also presented as BPEL processes, are primary candidates for rewriting an appropriate language (staying with Oracle: *Java* + *Spring JPA*).

You may think that BPEL processes in the task layer will remain untouched, and that would be a mistake. We will come to this quite soon. The **Application of Process Centralization (5)** pattern is just a beginning; we will focus on **Service Normalization (6)** shortly. Now what's interesting is how we will invoke our centralized services.

It is obvious that SPC will now act as a composition initiator, and we are quite close to realization of our Composition Controller (not agnostic yet). This controller will isolate the initiator from service(s)-actors, hide the complexity of the composition, and provide dynamic invocation together with the service locator component. From the previous figure, you can clearly see that this is the description of a classic J2EE Business Delegate, the way it's described in `http://www.corej2eepatterns.com` and `http://www.oracle.com/technetwork/java/businessdelegate-137562.html`. Refer to the following points:

- We want to promote Loose Coupling between the initiator and service-worker

- We want to minimize service calls from the composition initiator for fulfilling this complex task
- We would like to shield the initiator from composition or invocation exceptions, thereby providing a clear and understandable error context and completion status
- We want to keep this process dynamic, allowing logic to mutate without affecting the initiator
- We want to maintain this delegation as highly manageable, allowing us to easily change the business rules within the composition

Thus, we place the classic Business Delegate as a part of the solution where its role in SOA terms is slightly wider, covering service brokering and mediation at the same time. So, potentially, it would be the SCA mediator, as a router and message transformer. Potentially, it can detect the use case using a message header provided by the component-initiator (SPC in the earlier figure), and route the message to the decomposed BPEL process. We have already mentioned a drawback of this decision:

- A composition could be enormous if we place all the decomposed services in one SCA. Even the addition of certain aspects of atomic BPEL processes will make it rather heavy. Decomposing a big process and then assembling the atomic pieces in SCA using a mediator is a very common approach, but not for the discussed number of services.
- A mediator with a static routing table can be overinflated by the number of conditions and rules. Adding transformation to it will not make it better than the original solution. A rule-based mediation has certain limitations that we mentioned in *Chapter 2*, *An Introduction to Oracle Fusion – a Solid Foundation for Service Inventory*, and we will discuss it a bit further.

- Most importantly, quite a few of the decomposed processes are almost identical from the business logic and involved applications' perspective (Canonical Expressions and Canonical Schema are either in place or can be maintained).

The last bullet item means that for some processes, we could continue with the decomposition down to the endpoints' definitions, aiming to apply the dynamic partner links technique as an option or generalize the processes to simple XML configuration files in the form of routing slips, describing the endpoints that should be called. This process is similar to the ultimate DB's structure normalization, where we strive to eliminate redundant constructions and minimize dependencies. This activity is described as the SOA Process Normalization pattern where we eliminate redundant logic and establish the atomic process as a single carrier of business logic. Of course, the primary candidates for normalization are as follows:

- Processes of the same logic for different geographic units (only endpoints are different)
- Processes similar for one type of product or the same service-provisioning group

For these Processes/Service Normalization and Capability Composition (assembling capabilities outside the service boundaries in order to fulfill complex business logic; a typical role of task-orchestration services), business logic can be really "dehydrated" to routing slips. The concept of routing slips is quite common and mature; it's one of the core EAI patterns (`http://www.enterpriseintegrationpatterns.com/`) that is inherited by SOA as well. Products such as Apache Camel and ServiceMix (and the commercial Red Hat version, Fuse ESB) use it for dynamic and static routing. The whole concept is highly related to the WS-Addressing standard notation that we discussed in *Chapter 1, SOA Ecosystem – Interconnected Principles, Patterns, and Frameworks*: "The message came from A, then must go to the endpoint B and sequentially to C. The reply must be delivered to service D. In case of error, service E must be notified".

We would like to repeat it here again, just for comparison.

WS-Addressing	Static Routing Slip: ServiceMix
The code for WS-Addressing is as follows:	The code for Static Routing Slip is as follows:

`<SOAP:Header>`
`   <wsa:MessageID>uuid:`
`aaaabbbb-cccc-dddd-`
`                       eeee-`
`ffffffffffff`
`    </wsa:MessageID>`
` <wsa:ReplyTo>`
`   <wsa:Address>`
`      http://CTU.Order/CRM/`
`Clarify`
`   </wsa:Address>`
` </wsa:ReplyTo>`
` <wsa:To SOAP:must`
`Understand="1">`
`          mailto:support@CTU.`
`Order`
` </wsa:To>`
` <wsa:Action>`
`     http://CTU.Order/`
`Provision/Activate`
` </wsa:Action>`
` <wsa:FaultTo>`
`  <wsa:Address>`
`  http://www.w3.org/2005/08/`
`addressing/anonymous`
`  </wsa:Address>`
`  <wsa:Reference`
`Parameters>`
`   <ctuns:ParameterA`
`       xmlns:ctuns="http://`
`CTU.Order.namespace">`
`       FAULT`
`  </ctuns:ParameterA>`
`  <ParameterB>`
`ServiceBroker</ParameterB>`
`  </wsa:Reference`
`Parameters>`
` </wsa:FaultTo>`
`</SOAP:Header>`
	`<eip:static-routing-slip`
	`     service="ctuOrder:`
	`routingSlip"`
	`     endpoint="endpoint">`
	` <eip:targets>`
	`    <eip:exchange-target`
	
	`service="ctuOrder:Service1" />`
	`    <eip:exchange-target`
	
	`service="ctuOrder:Service2" />`
	`    <eip:exchange-target`
	
	`service="ctuOrder:Service3" />`
	`  </eip:targets>`
	`</eip:static-routing-slip>`

Thus, this ultimate implementation of Normalization of an SOA pattern leads to the conversion of BPEL artifacts into some primitive XML constructs, fulfilling a part of the WS-Addressing paradigm.

"Stop" you say? Two points must be clarified straightaway:

- Are we reinventing another more lightweight BPEL?
- Intermediate routing within SCA is perfectly covered by a mediator. What's wrong with it, and why can't we employ it instead of routing slips?

Firstly, BPEL 2.0 is a full-fledged language with quite an extensive syntax. It's no longer a simple approach with essential "Assign-Invoke" commands. A complex syntax requires a very smart Interpretation Engine. Look at the construct on the right-hand side of the previous table. You do not need an engine to execute this; any language with a standard XDK can execute it gracefully. Even without any XDK, you can execute it using less than a dozen lines of code. Secondly, it's transportable and portable. WS-Addressing is just fine, but it's part of the SOAP header, and it's still the `WS-*` standard; however, we will not always call WS, and WSIF is not always an option.

Also, you probably noticed that the routing slip is nothing more than a description of Endpoints. Endpoints (API) are one of the key (probably most crucial) elements of our service infrastructure, and from the Governance standpoint, we must simply register and maintain them in our Service Repository (and Registry as well). The Registry structure will be discussed in a separate chapter, but for now, we can assume that endpoints are already there after the completion of decomposition and logic centralization. It's also not a big deal to extract a consolidated endpoint descriptor in the form of a routing slip from the Repository. We have to stress that we are not opposing WS-Addressing routing slips and BPEL as an Orchestration language; further, you will see how they gracefully coexist. Now we come to the realization of two new SOA patterns:

- **Metadata Centralization (4)**
- **Inventory Endpoint (7)**

Metadata is data about data (probably the shortest description possible, and we love it). The endpoints and types of Service Engines in use, which include runtime roles of services and service models, are data elements that describe the service. They will be in our Inventory eventually (the sooner the better) but accessed through the unified Inventory Endpoint. So far, we have plans to keep the service endpoints and extract them using the WS API, which we described in the development phase.

> By cautiously practicing the philosophy "one baby step at a time," we cannot boldly proclaim an emerging inventory as an Enterprise. This Service Inventory currently belongs to the OSS/BSS domain; thus, it's a **Domain Inventory (3)**.

Returning to the first point of concern, we would like to confirm its validity. Keeping the routing slip reasonably simple is the key; otherwise, we will just reinvent the wheel. It would be much easier than presenting the routing slip as the BPEL 2.0 XML artifact and the routing slip parser as a subclass of the BPEL engine, reacting only on the constructs invoke and transform. By doing this, universalism will be maintained, but we will have to forget about portability.

> Select a very limited vocabulary for your routing slip if you decide to go this way (as if you have a choice). Technically, we would like to execute some tasks by calling public endpoints. Later we will call it **Execution Plan** (**EP**), with some similarity to DB's execution plans.

Thus, we have no plans to reinvent the BPEL. Even more, by balancing the Normalization of different processes, we openly declare that not all processes can (and should) be decomposed down to a simple sequential EP. We deliberately would like to keep complex processes untouched and invoke them from the same EPs. Even more, we will denormalize some processes due to the presence of complex parallel processing or difficulties that arise with error compensation. Another reason for the implementation of the Contract Denormalization pattern is the needless multiplication of the services and high performance demands associated with it. For sequential processes though, Execution Plans (structure will be explained in detail with connection to Enterprise Repository in *Chapter 5*, *Maintaining the Core – the Service Repository*, but critical elements will be presented here) can solve this problem more gracefully.

Next, all of the logic related to BPEL is applicable to the Mediator concern as well. The fact is that the Oracle Mediator is not lighter than BPEL, and its engine is also rather heavy. Yes, the syntax and vocabulary is a bit more modest, but Mediator is also a stateful component that is capable enough to perform the processing sequentially and in parallel with the predefined priorities. Thus, its XML metalanguage is not the best candidate for performing a simple sequential EP. However, similar to what we did with BPEL, we are planning to use at least two (initially) mediators: one for dynamically extracting EP or routing to the complex denormalized BPEL and another for routing to the specific endpoint (this decision will be revised soon).

Gradually, we will come to the dynamic extraction of EPs, which can be realized on Rule Engine; now we are moving on to the **Rule Centralization pattern (8)**, enforced by Oracle SCA Decision Service. Here, another concern can be expressed. Would it be more logical to use Rule Engine to identify the next task in the dynamic workflow instead of extracting the execution plans by the same Rule Engine? The answer is already in the question. For a sequential composition, we will need up to 10 Decision Service invocations instead of one (see requirements). Most importantly, for dynamically branching a process, we can invoke RE from EP with no problems, or we can call another SCA that could encapsulate another Decision Service.

Functional analysis would not be complete without defining the message structure that is consumed and propagated between the controller, composition members, service providers, and composition initiator. Obviously, the message payload is the Order in the CDM form according to the TMForum specification. As mentioned in *Chapter 1*, *SOA Ecosystem – Interconnected Principles, Patterns, and Frameworks*, discussing the *Standardized Service Contract*, the optimization of its structure is extremely important, but we cannot approach it just yet due to compatibility reasons (external applications are used to it) and because all those enormous BPEL processes are our primary concern at the time. However, we will have to address message normalization as soon as possible. Due to this pending task, we have to construct our message in such a way that the alteration of payload would not affect Composition Controller's functionality, that is, make it universal (or process agnostic). It could be possible if we isolate the payload and a dynamic part of it, and expose the payload object's particulars in the message header. The Execution Plan(s) must also be presented in the message, and mandatory routing slip(s) and statuses of object alterations must also be preserved for the controller's convenience.

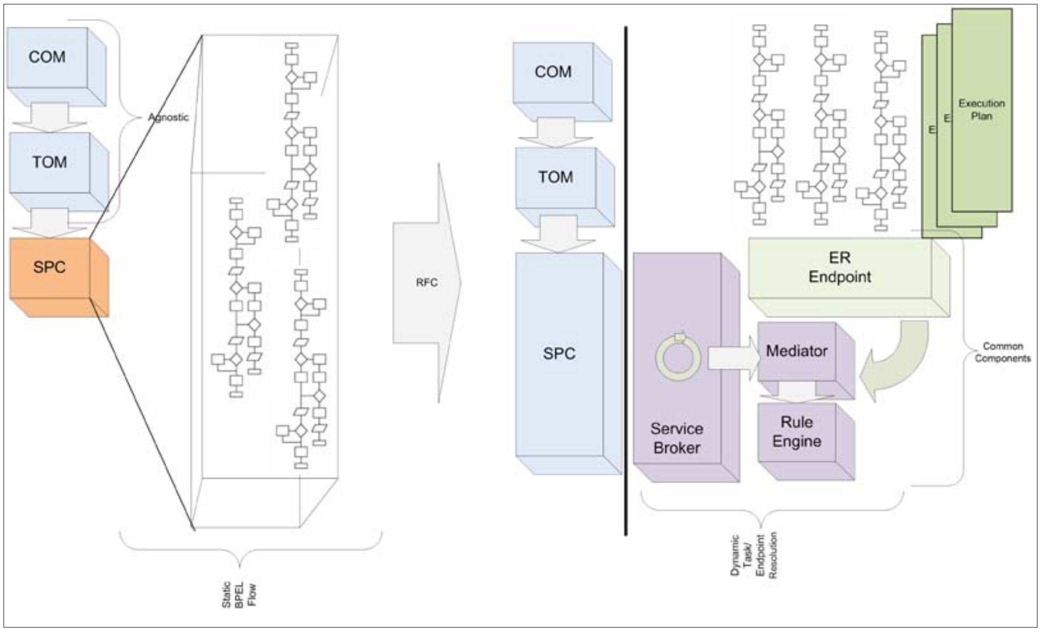
All these message components are best discussed in conjunction with the Service Repository taxonomy, and we will do that in the coming chapters. We would just like to stress on the fact that the implementation of a universal message-container will convert our Composition Controller into a truly agnostic service that is capable of serving not only Order messages, but all types of payloads in a very dynamic and lightweight fashion. This will inevitably lead to the following consequences:

- As you will realize, a universal message container can be designed by the implementation of the `<any>` element in the payload section. (Yes, by `[CDATA]` as well, but we are not talking about this extreme case for obvious reasons.) We mentioned this point in *Chapter 1*, *SOA Ecosystem – Interconnected Principles, Patterns, and Frameworks*, as rather undesirable: security considerations and excessive XML processing at the backend. This could be fairly and successfully mitigated by the following:
    - We are in the EBF framework behind security gateways in DMZ and the conventional ESB. If the enemy is already present, it's too late anyway.

    - Besides, our payload is not really `<any>`; it's still a corporate EBO (Order, Invoice, Client, Device, and Resource) and is strictly compliant with CDM's XSD. If we would like to perform message screening in EBF by means of an XDK, we can do that easily (although, it's rather unusual and ineffective all the same).

- The message header will be in place despite the presence of any type of message structure. Should we also mention that it will be SBDH-compliant (see *Chapter 1*, *SOA Ecosystem – Interconnected Principles, Patterns, and Frameworks*)? Thus, all objects' particulars will be in the header anyway, simplifying payload processing. Even more, this approach could make our payload lighter, as we can delegate some common objects' elements to the message header.
- A universal message-container denotes the universal agnostic Composition Controller, and consequently, the promotion of Domain Inventory to Enterprise Inventory. This would be a very positive outcome.

Now we can summarize the results of our analysis and draw the functional diagram of the proposed solution. We have an initial understanding of the tools we are going to use, and all of them are in the SOA Suite bundle (both 11*g* and 12*c*) and quite well within the EBF framework.



The summary of the results is as mentioned in the following table:

No.	Component	Purpose	OFM SCA
1	Decision Service (RE)	This component is used to replace the hardcoded "if-else" business logic and make it portable, centralized, and detached from the composite application and make the business logic more visible for business analysts. This is also used to establish dynamic routing logic based on the previous step and maintain the endpoint URL resolution on the fly.	Oracle Rule Engine as Decision service within Composite. As Separate activity invoked within Service broker in order to obtain execution plan based on message parameters. Execution plan can be a preconstructed object or can be constructed by Rule Engine on the fly. Pros: Truly detach business logic from SCA and make it highly dynamic and manageable. Implements Process centralization and Functional decomposition patterns. Cons: Gives most positive results when logic is complex. Increases maintenance costs. Dynamic routing based on Decision Service is a standard Oracle SCA pattern. Mediator is required. Downsides are discussed.
2	Service Repository (ER)	This is a placeholder where all service references are stored (such as URL) with supplemental information in some Service Profile form. Task lists' references could also be stored here as Orchestrated Task Services.	Oracle Service Repository Custom DB Local XML files
3	Service Inventory Endpoint	This is a Service Registry Lookup service and is used to accept CDM elements as parameters and return the Execution Plan object.	Depending on the realization, it will act as an adapter to a DB or file. Alternatively, it can act as a solution, as in the previous step.

No.	Component	Purpose	OFM SCA
4	Execution Plan Object (task/ endpoint list)	Orchestrated Task service sequential representation; a collection of tasks/endpoints in the form of execution plans. These are dynamically discovered and are an array of tasks with associated compensation/rollback tasks (service endpoint IDs).	XML Part of the universal CTU message-container
5	Business Delegate	Dispatcher to the actual worker. Looping through the Execution Plan and sequentially invoke services from a list. Store response and dispatch to associated compensation task/endpoint if response is negative.	BPEL process within composite with four activities: • Assign MH elements. • Using MH elements obtain Execution Plan. • Loop over Execution Plan and invoke tasks/ services. Depending on response, proceed to next or compensation. • Wrap up response.
6	Dynamic Endpoint Resolution (Mediator)	Isolate Business Delegate from actual service implementation. Dynamically resolves service URI.	See the previous step
7	Transformation Service (optional)	Transform and Enrich Messages between service invocations.	Optional service/activity: • Can be atomic BPEL process for each XSLT. • Can be generic service with XSLT as a parameter from Execution Plan.

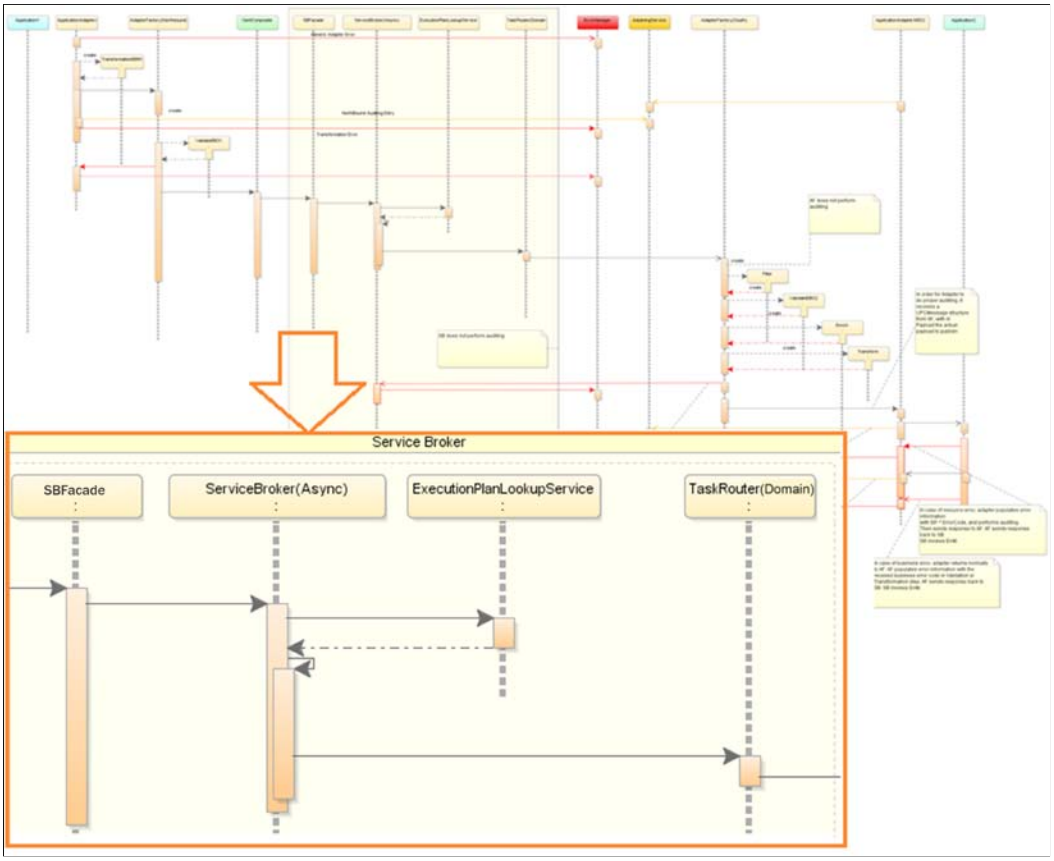
Asynchronous agnostic Composition Controller

The block diagrams shown previously present the complete solution that consisted of the order management part (non-agnostic task-orchestrated services together with entity services), encompassing the following generic parts as well:

- Business delegates as service brokers, both synchronous and asynchronous (currently we will focus on asynchronous), fulfilling the role of an agnostic Composition Controller

- A Service Broker facade, responsible for wrapping an inbound Order message into a message container
- An Inventory Endpoint in the form of Execution Plan Lookup Service, as we have only EPs in our Metadata Repository for the time being
- A Task Router that is presented as a mediator for a synchronous service broker

The next step after you're done with drawing a comprehensible solution block diagram is presenting detailed sequence diagrams, covering all aspects of the components' runtime behavior. Needless to say that it can only be done with close cooperation with the developers. Here, we would like to present the complete sequence diagram with focus on Service Broker's activities shown as follows:



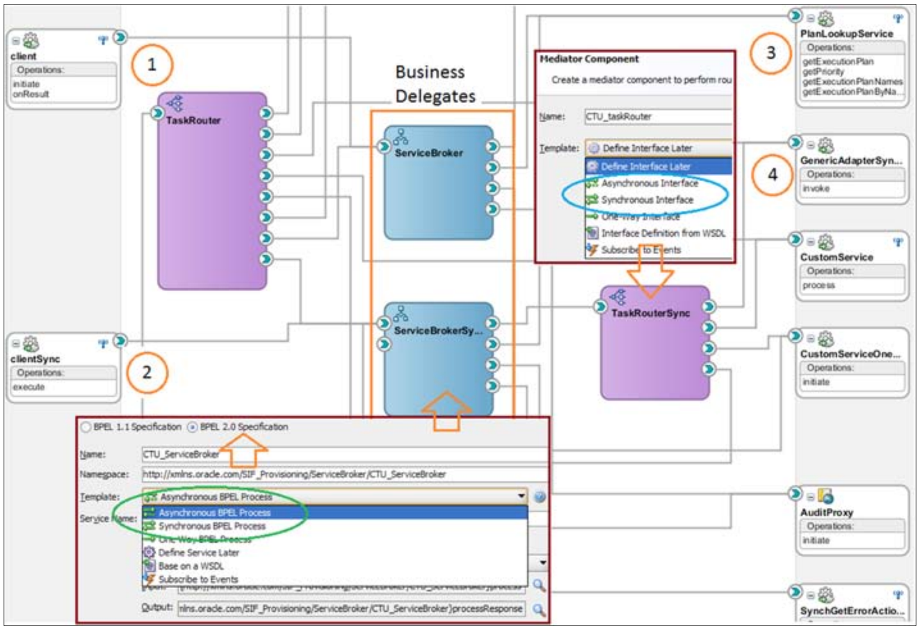
Service Brokers as a core of Composition Controller

Usually, sequence diagrams should contain all the elements of the solution to visualize the service activities that involve the following:

- An audit service
- Error handlers
- Inbound and outbound adapters
- Common Order-handling components

We will discuss each of them later. In the previous figure, we depicted the part that acts as an agnostic Composition Controller. This part can be created in any environment (again, SOA patterns are vendor-neutral) for any of your projects where an agnostic controller's functionality is required; however, for good reason, you should plan to implement it in BPEL. Now we will drill into some details that are essential for its physical implementation, although we will stay focused on the positive execution scenario first (rollbacks will be discussed later in the chapter).

All controllers' SOA artifacts will be packed into one composite for simplicity. This simplicity will be illusive from the very beginning as we will have to handle synchronous and asynchronous operations in one place. Thus, we will have two pairs of BPEL (as a service broker and business delegate) and Mediator as a task router. However, that's the whole idea: gradually convert a bulky solution into something composable. The four main steps in the realization of this idea's realization are presented in the following SCA diagram:



Mediator Component

Create a mediator component to perform rou

Name: CTU_taskRouter

Template: Define Interface Later

○ BPEL 1.1 Specification ● BPEL 2.0 Specification

Name: CTU_ServiceBroker

Namespace: http://xmlns.oracle.com/SIF_Provisioning/ServiceBroker/CTU_ServiceBroker

Template: Asynchronous BPEL Process

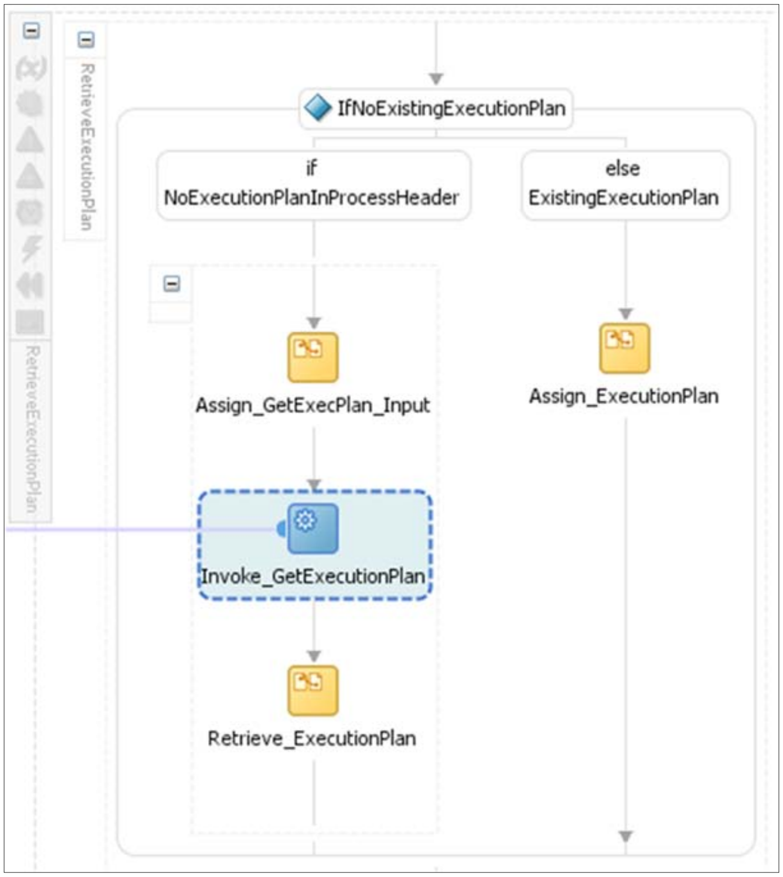
Service Name:

Output: nlns.oracle.com/SIF_Provisioning/ServiceBroker/CTU_ServiceBroker}processResponse

Composition Controller's realization on SCA

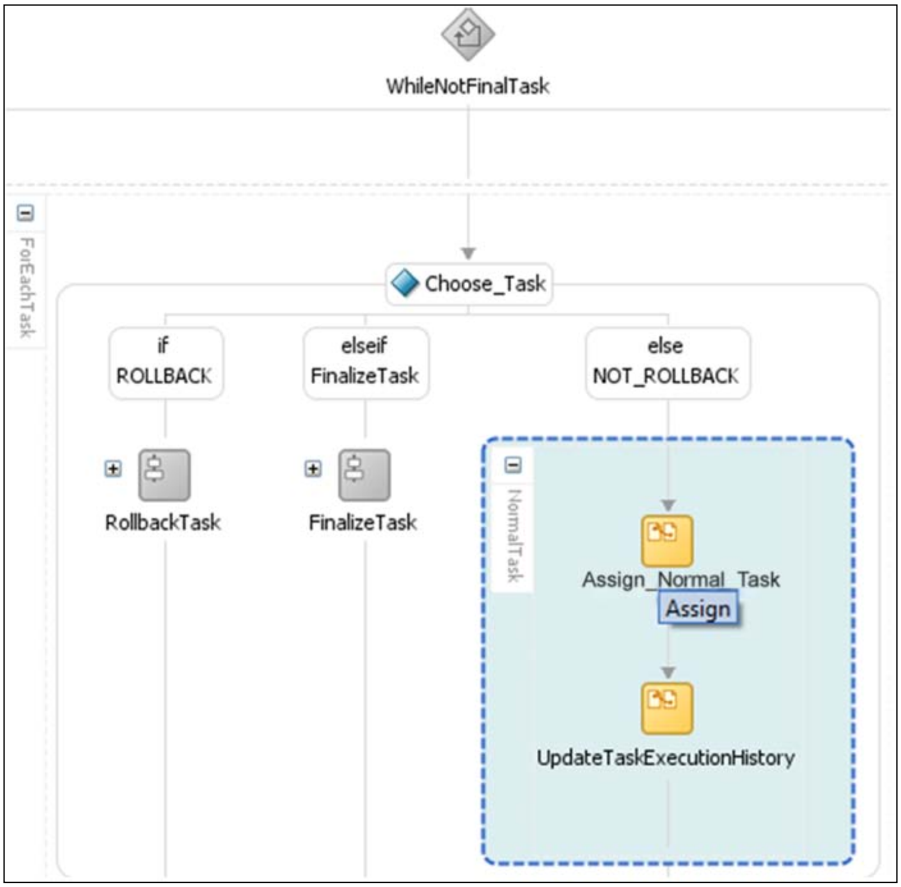
Naturally, two types of invocations could be possible: one from the async (**1**) initiator and the other from sync (**2**). Then, Execution Plan will be extracted for every call (**3**), and every sequential task will be passed to the Adapter framework (**4**).

Create the asynchronous Service Broker as an asynchronous BPEL, exposing its WSDL as a SOAP service. We are omitting some obvious initial steps only to return to them later. The first important step is to invoke the Inventory Endpoint and extract EP as a routing slip, as shown in the following screenshot:



Obviously, EP must be implanted into the universal message container for further traversing in the loop during the execution/invocation. Universality means that a controller can be nested so that it acts as a subcontroller; therefore, the extraction of an EP is not always necessary. So, we should recognize the presence of an EP provided by a master-controller and invoke the lookup only if it's necessary. Logically, EP can be provided by the Adapter framework, where an individual adapter handles a particular use case. EP's extraction is based on message header elements that are initially known: sender, affiliate, object (Order), and event (requested product and operation). The Message Header is SBDH-compliant as discussed in *Chapter 1, SOA Ecosystem – Interconnected Principles, Patterns, and Frameworks*. The final step here is to store the EP tasks array in a global variable.

We got the list of tasks as an XML array, and now we can invoke them sequentially in the loop shown as follows:



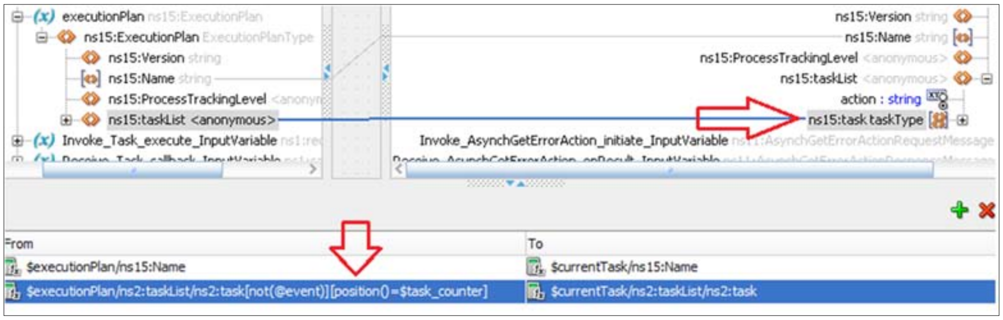
This is the essence of the constructed BPEL process and the core of the execution logic, although later, we will have to add some extra functionalities to keep it agnostic and task-oriented at the same time. As explained during the analysis phase, each order line could dynamically spawn a number of subtasks, which should be logically grouped, where the last task in a group should perform a specific operation on the service context. This task is the last one in the group of EP and the middle lane of the control is responsible for handling it. The right lane is for the standard task; it also updates the task execution history, but only the ones that are related to tasks such as `status`, `isRolbackRequired`, and `currentTaskcounter`, which are essential for traversing and looping. The left lane is responsible for executing the compensation task when the response from the actor is negative; it's possible only if current and compensative tasks are included in EP in relation to the primary task.

Using the `While` looping is quite easy; with the following code, we just simplified the condition for the sake of demonstration:

```
<while name="WhileNotFinalTask">
     <condition>
            (...   ($finalizeTask_counter <= $finalizeTask_total )...)
   </condition>
   <scope name="ExecuteTask" exitOnStandardFault="no">
```

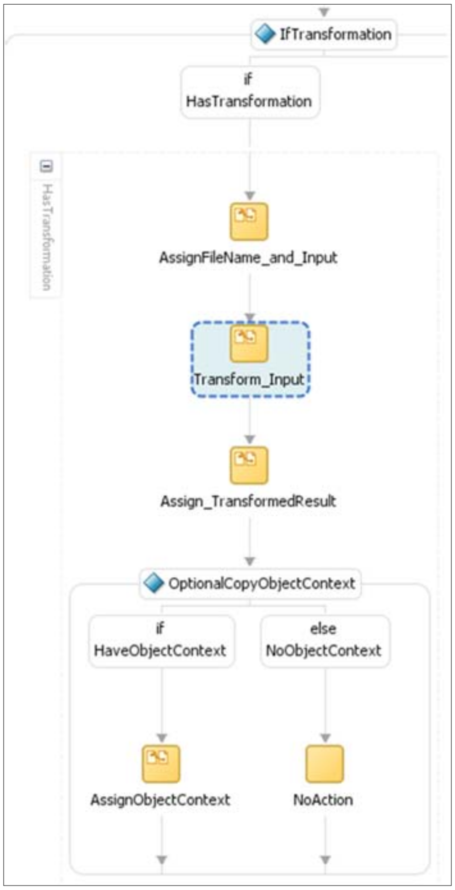
There are many ways in which you can loop over XML nodes in Oracle BPEL. All of them involve the Assign element, counting the number of XML nodes using the `count` function in XPath and XPath's `position[]` function. The one we can use in this case was extracted in the previous step where the execution plan was stored in a global variable, which is common for all BPEL scopes. In the `ForEachTsk` scope, to the single EP task element that currently exists in the universal Message Container (Process Header) part, we assign the value of the task node from the global variable with the position that is equal to the current task counter. The task counter is incremented after each task is invoked, and we will perform a reassignment in every loop.

This is probably not the fastest technique, but it works; additionally, we have advantages in terms of memory utilization, keeping only one node at a time in the Process Header within the message container.


executionPlan ns15:ExecutionPlan
ns15:ExecutionPlan ExecutionPlanType
ns15:Version string
ns15:Name string
ns15:ProcessTrackingLevel <anonym
ns15:taskList <anonymous>
Invoke_Task_execute_InputVariable
ns15:Version string
ns15:Name string
ns15:ProcessTrackingLevel <anonymous>
ns15:taskList <anonymous>
action : string
ns15:task taskType
Invoke_AsynchGetErrorAction_initiate_InputVariable ns11:AsynchGetErrorActionRequestMessage

From	To
$executionPlan/ns15:Name	$currentTask/ns15:Name
$executionPlan/ns2:taskList/ns2:task[not(@event)][position()=$task_counter]	$currentTask/ns2:taskList/ns2:task

Extend the task types further to make the controller more universal.



With no intention of making our routing slip heavy or close to BPEL, we have to extend its syntax a bit further. It is obvious that different tasks in our task sequence will require different data subsets; even if we use a canonical XSD for our Order (actually, we are not going to use it yet the TM Forum model is too "all-weather"). This fact will require additional transformations between invocations. We can perform these executions in the adapter framework layer and relieve the Business Delegate from this functionality; however, some of our service-workers will be other composites, standalone BPEL(s), or other services that are not covered by the ABCS layer yet.

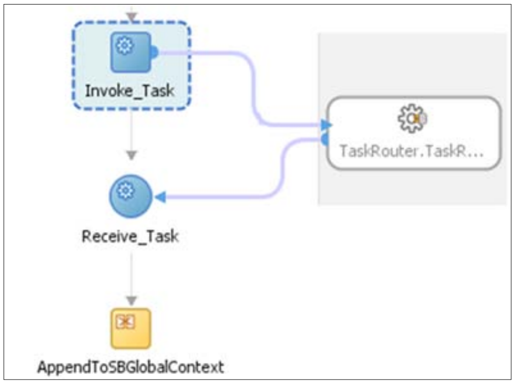
A separate lightweight Servlet can do it gracefully, making the solution truly universal; however, because we have decided to stay in a single composite for now, the standard BPEL functionality will be enough:

```
ora:processXSLT($TransformationFileName, $TransformationInput)
```

Here, `TransformationFileName` is a task parameter, assigned in BPEL as shown in the following code snippet:

```
<assign name="AssignFileName_and_Input">
    <copy>
         <from>
         string($currentTask/ns15:taskList/ns15:task/ns15:transform/@
location)
      </from>
      <to>$TransformationFileName
     </to>
  </copy>
```

We must also supply the task worker (service provider) with additional information about the transported business object. This information resides in the SBDH message header, the Object Context part, and presented in the name/value pairs form, storing all objects' supplemental information. Now we are ready to invoke the actual Service Provider.



Agnostic Task Invocation

The only task associated with the designated provider is maintained in the Execution plan, and this task is exactly what we want to execute. Thus, the extraction of this variable is simple, and we pass it to Task Router (SCA Mediator). It is now the Mediator's job to substitute the actual endpoint URI with the variable from the task list.

This operation is similar for synchronous and asynchronous service brokers; only Mediators are different.

That's it! We can extract the XML task array from the metadata inventory; this array has endpoint references, and we can loop through and invoke them synchronously or asynchronously, get the result back, update an object's payload and its contexts, keep track of the execution in a special placeholder of our XML container, and even react to errors with predefined tasks in the same tasks array.

The mentioned endpoint reference term is part of the WS-Addressing vocabulary we briefly discussed in *Chapter 1*, *SOA Ecosystem – Interconnected Principles, Patterns, and Frameworks*. Let's look at it a bit closer now. The endpoint reference is the XML construct that allows us to select one of the available services in a WSDL. We can also define the service endpoint at runtime. As we just saw, BPEL is very good at assigning variables at runtime, practically for all XML entities in a composite. With the very first version of the Oracle BPEL manager, we got a brilliant *DynamicPartnerLink* example, demonstrating the substitution of a dynamic partner-link address; it was turned into an excellent chapter in *SOA Best Practices: The BPEL Cookbook*, *Oracle*.

One might think that as we focus on SCA (and BPEL in particular), why not really implement everything in BPEL and use the same old technique (with no Mediator)? Actually, the old example is similar to Mediator, how it exists today. We initially defined a dummy service endpoint and created a WSDL for it with all the predefined real endpoints; after that, we used the `wsa:EndpointReference` type variable in assignment to the `partnerReference` variable, pointing to the real endpoint in WSDL. This approach could be taken even further with the substitution of an actual WSDL with different addresses for the same reference. This approach is not purely dynamic, as we will have to redeploy the new WSDL. What was proved is that we can call any of the services even without addresses registered in WSDL, as long we know the actual address at runtime. For addresses that are not specified, the default endpoint in WSDL will be used. All of these benefits though are already in the SCA Mediator (with lots of extra benefits), and we do not need to modify BPEL. We delegate the endpoint reference and partner links' handling to the Mediator without affecting BPEL and WSDL alterations. Here, we are clearly following the separation of concern principle.

Before getting a bit deeper into Mediator, we would like to check some other features we would like to include into the BPEL Service Broker/Business Delegate component. We already added the transformation and clearly separated it from the task invocation (we can still invoke something that will perform the transformation for us). However, what else can be added?

We assume that all responses (callbacks) will be handled by the adapter framework, and ABCS will make sure that the response is based according to XSD. Assumption could be a bit bold, and some internal services could be called directly. A service broker in this case is obliged to validate the output before calling the next service. Therefore, adding a new task type, Validate, in addition to Invoke and Transform is inevitable. It's not a problem yet, as it can be easily supported, but we better stop here, otherwise the SB will not be better than the initial monstrous BPEL.

> Select this approach with extra caution. It has all rights to exist because BPEL is one of the possible platforms for ABCS and Service Broker with routing capabilities; it is an essential pattern for northbound and southbound adapters. In a non-agnostic controller, the best approach would be to validate the XML property for validating inbound and outbound XML messages; just set it to `True` (in the SOA EM console, **SOA Administration | BPEL Properties**). Of course, it impacts the performance. For our agnostic controller with a containerized payload as the `<any>` element, we have limited options, but anyway, it would be more prudent to delegate this function to every individual adapter that is isolating the endpoint.

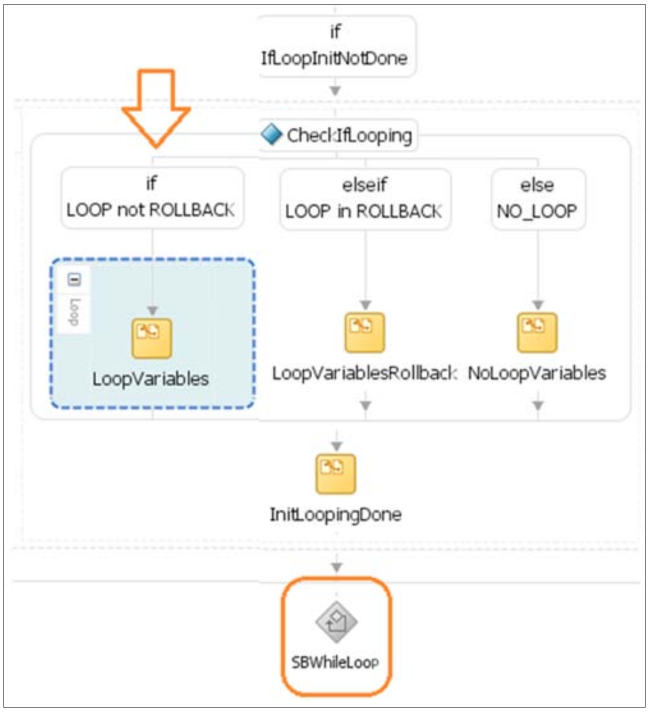
One more thing. We have already departed from the telecom-orders-specific Composition Controller and strive to achieve a truly agnostic realization, hence the routing slip could potentially contain more than ten invocations (endpoints). In the SCM domain, for instance, we deal with objects such as CargoManifest, BillOfLading, VesselSchedule, and quite often, the same task will be repeated a considerable number of times (even dozens). How can we do that?

Extending the asynchronous agnostic Composition Controller

The following table explains the method to accomplish the functions in the first column:

Task	Realization
Include the same task *n* number of times in EP	This option allows us to keep the task parser within the Business Delegate simplified. Only one loop is maintained. Alternatively, if we need to perform nested looping, we will call another service broker's instance with the new EP (extracted first), or another composite with the `While` loop based on the payload, extracted EP, and object's context. This option is very attractive for the implementation of `parallel` or `foreach` flows. The drawback here could be that the size of the EP can grow considerably depending on the number of task interactions.
Delegate the looping to the Adapter framework	At first glance, this would be the best approach, as it looks similar to debatching. With debatching, we send message chunks to the designated endpoint. Unfortunately, debatching here is not pure as we want to send the entire message several times (or perform several invocations with the same payload); most importantly, batching and debatching functionalities are supported for a limited number of adapters: DB, FTP, and File for inbound adapters. Anyway, an adapter is the most logical option from the separation of concern standpoint. We have to supply an adapter with the number of interactions we want to execute and that could be done using EP (see the following option with the looping attribute). The previous tip is also valid here.

Set an extra attribute for individual tasks, describing the number of iterations for this task:



Here, setting the attribute is not a problem:

```
$currentTask/ns15:taskList/ns15:task/@loopOver
```

However, we should be very careful about further requirements that might follow. Technically, we have implemented a nested loop for a single task, but this task can be part of the compensative actions (the Rollback EP mode) for carrying out the primary operation, and it can compensate the single (last) interaction for the task group or the entire group. In other words, without the support of native adapters, we have to handle all the errors ourselves with this approach, and nested looping must be applied discriminately for positive and negative execution scenarios.

You can also follow the second option (adapter); in this case, establishing the attribute will be enough for fulfilling these requirements, and the specific adapter will do it non-agnostically.

Using this five-step approach, we have implemented the agnostic Composition Controller using SCA components. This is the first iteration after functional decomposition, and we still have some deficiencies in the design:

- We still have synchronous and asynchronous parts in one composite.
- Focusing mostly on the component's design, we didn't cover the structure of the message container and all its parts. We will do this later in relation to the metadata storage.

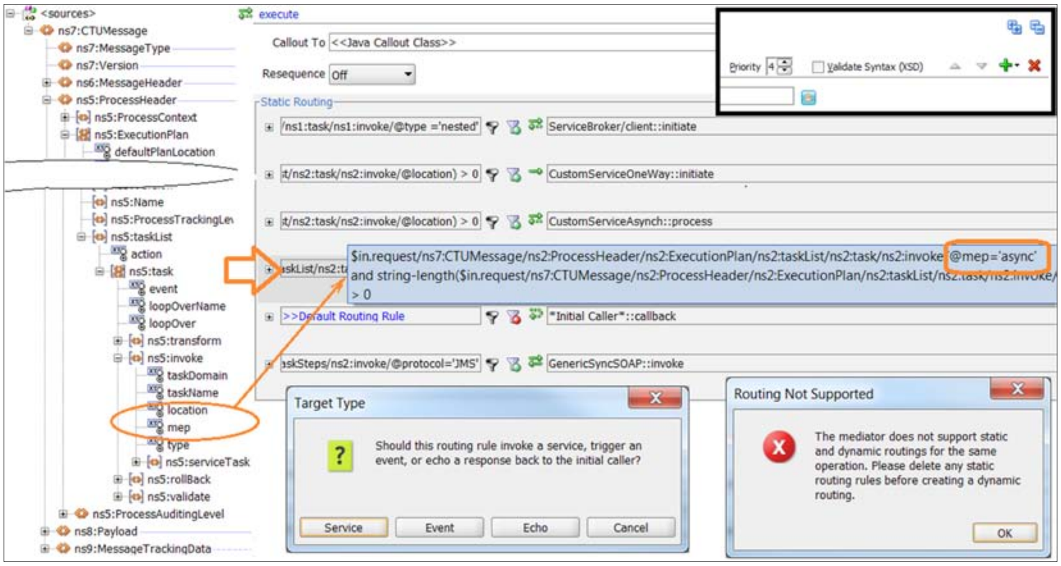
Now we will discuss the role of Mediators in our design and in general.

Usage and limitations of a Mediator as a dynamic router

The primary role of Mediators is to implement the Intermediate Routing SOA pattern in SOA Suite SCAs, which are the building blocks of either Orchestration (EBF) or Adapter frameworks. From the SOA Patterns catalog, we know that Intermediate Routing together with the Service Broker belongs to the ESB compound pattern. So, at first glance, it might be a bit outlandish to have them both actively discussed in the chapter that is dedicated to Orchestration. This is the obvious question that arises when someone tries to apply (or avoid applying) a seemingly fitting pattern, instead of understanding the actual problem and the ways of its mitigation (not exactly by the pattern with a fancy name from the catalog).

The message path in complex compositions could be equally complex, sometimes even unpredictable at design time, requiring dynamic (or rule-based) dispatching. Oracle SOA Suite composites (as deployment units) could be really complex, aggregating different components (primarily presented, but not limited, by four predefined types); the negative impact of excessive usage of the heavy `if-else` logic that is implanted into any of primary components (mainly BPEL) has been clearly explained at the beginning of this chapter.

The BPEL's dynamic endpoints' invocation approach discussed in the *DynamicPartnerLink* example is ingenious but has certain limitations that are mentioned in this paragraph and the first bulleted list in this chapter. Also, this approach is apparently obsolete with the presence of SOA Suite SCA 11*g*. We need more than just simple WS invocations in our use case, and Mediator covers them all gracefully.



Callout To <<Java Callout Class>>

Resequence Off

Priority 4 ☐ Validate Syntax (XSD)

/ns1:task/ns1:invoke/@type ='nested' ServiceBroker/client::initiate

:t/ns2:task/ns2:invoke/@location) > 0 CustomServiceOneWay::initiate

:t/ns2:task/ns2:invoke/@location) > 0 CustomServiceAsynch::process

askList/ns2:ta $in.request/ns7:CTUMessage/ns2:ProcessHeader/ns2:ExecutionPlan/ns2:taskList/ns2:task/ns2:invoke @mep='async'
and string-length($in.request/ns7:CTUMessage/ns2:ProcessHeader/ns2:ExecutionPlan/ns2:taskList/ns2:task/ns2:invoke/
> 0

>>Default Routing Rule *Initial Caller*::callback

askSteps/ns2:invoke/@protocol='JMS' GenericSyncSOAP::invoke

Target Type

Should this routing rule invoke a service, trigger an event, or echo a response back to the initial caller?

Routing Not Supported

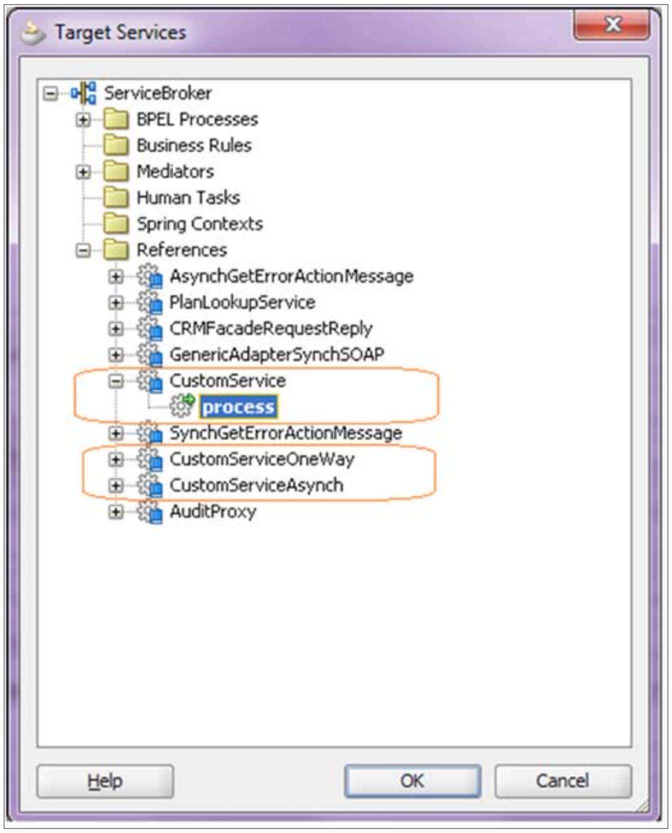
The mediator does not support static and dynamic routings for the same operation. Please delete any static routing rules before creating a dynamic routing.

Configuring Mediator in Agnostic Composition Controller

Do you remember that in the first release of our Service Broker, we have two mediators for synchronous and asynchronous processes? The logic we will cover next is applicable for each of them.

Firstly, by decoupling (isolation) our business delegate BPEL process from actual service-workers, we have to anticipate different MEPS involved in service activities. Yes, we have already demonstrated that Mediator can initially be created with either synchronous or asynchronous interfaces (we will discuss one-way and event-based later).

This fact, though, is significant for SCA dehydration routines (Mediator *is* a stateful component whereas OSB is *not*) and for the definition of the Port Type; naturally, the sync interface has one Port Type as the response that is returned to the caller, whereas, for async, you can define Callback Port Type for response messages.



For now, it's important to provide all combinations of one- and two-way MEPs.

Mediators' MEPs (green one- and two-way arrows to the left of the service address fields) are based on the service WSDL we are selecting for target service operations. We have already decomposed a considerable number of processes, and we are quite aware of their WSDL. Therefore, any speculation about the possible MEPs and operations will be easy, and it's pretty similar to defining the "generic LoanService" from the *DynamicPartnerLink* example in *SOA Best Practices: The BPEL Cookbook, Oracle*.

Here, we just go a little further and define the abstract and generic "Sync CustomService", "Async CustomService", and "OneWay CustomService" for services we would like to invoke directly from our service broker. Also, in the earlier screenshot, you can see some statically defined services such as Audit, EP Lookup Service, and Error Handler, which we will be using anyway. The CRM facade here is just an example of the static business non-agnostic service, which is still not covered by a dynamic composition. Feel free to amend it in your actual implementation.

The generic Adapter endpoint in the list is probably the most important endpoint for cross-framework communication as it represents an OSB adapter, fulfilling the same routing/transformation functions in the EBS framework. That's how we communicate with eternal standalone services and applications that are not directly invoked by a single SCA.

The sequence of establishing this static routing is quite standard (see the Oracle docs, `http://docs.oracle.com/cd/E17904_01/integration.1111/e10224/med_createrr.htm`); we will shortly highlight the important step. After clicking on the green plus sign in the upper-right corner, you will be invited to select the target service. It is obvious that static and dynamic routes cannot be mixed in one mediator instance (a warning is shown in the screenshot); you can consider it as a limitation, but there are several simple ways to mitigate it. Nothing prevents you from calling another mediator of the desired type sequentially.

The next step is to learn how to set content-based routing using an expression builder. The content and structure of our execution plan are completely related to our metadata taxonomy in ESR; we will have to dedicate much more time to it later, although we have already displayed several key fields in the earlier screenshot:

- **mep**: This is the flag that is used for routing to the appropriate abstract service. In our case, it could be generic sync, async, and one-way flags. You can add your own MEP type according to WSDL 2.0 MEP if you want.
- **location**: This is the concrete service URI and will be used further for substituting in a generic endpoint. This is necessary for distinguishing endpoint types.
- **taskDomain**: This is an optional element for routing to the specific business domain (another Mediator or Service Broker in a separate business domain). It could be the CRM, ERP, SCM, and so on. Use your own business landscape, but this element in general contradicts the agnostic nature of our Service Broker.

- **serviceEngine type**: This was not shown earlier, but it is quite a useful element that you can declare as a service engine that will be responsible for executing tasks such as BPEL, Transformation, and DB. It's especially relevant for transformations, as they can display different behavior in complex compositions. You can route to the specific engine using this element.

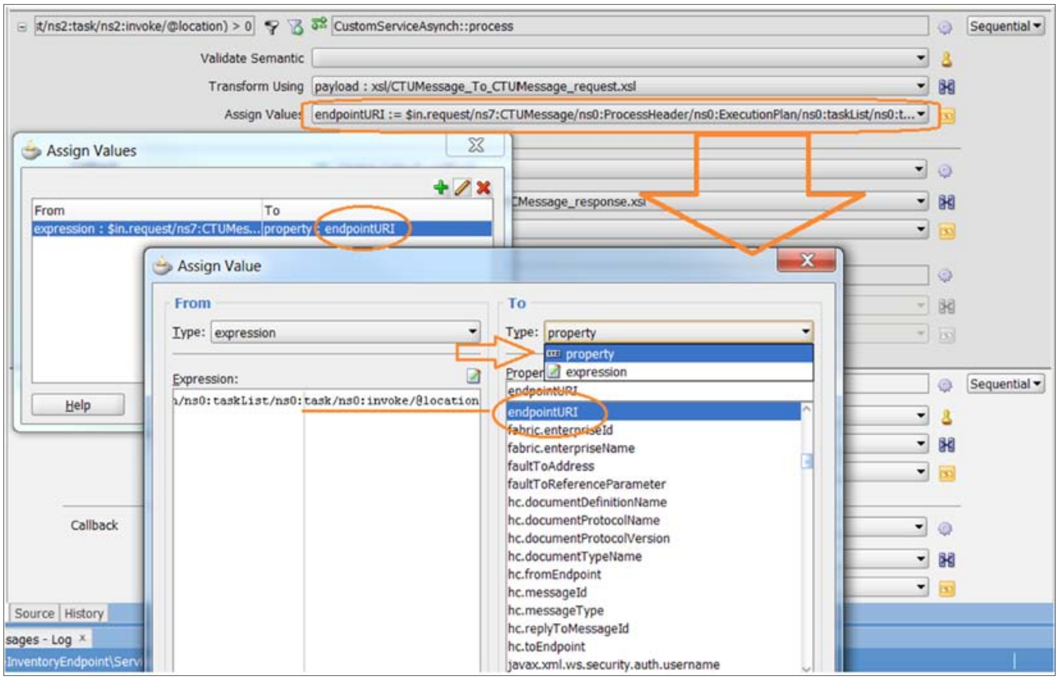
  We are still in Mediator's request path. Here, there are three things that we should consider (following the logic of our solution).

- **Assign value**: This is the field that does all the magic. Using mediators' properties (we have plenty of useful properties; please read about these in the documentation, as we have no place to discuss them here), `endpointURI` in particular, we can assign our service URI from the execution plan dynamically, resolving any address issues according to the selected MEP.

> Please remember that the Mediator has inbound and outbound properties, and it could happen that you will not find the right property in the drop-down list. Just type `property` in the dialog box field and make sure that you spell it correctly. It's not a bug; it's a long lasting feature now.

- **Content transformation**: This is another highly important feature of Mediator. Yes, we are trying to minimize transformations within a single service domain; however, with the huge legacy application burden, it's not always possible. For an agnostic Composition Controller, it is even more inevitable, as we need to extract (by means of XSLT) the payload from our agnostic message container (CTUMessage) because the endpoint application cannot handle the entire container. Apparently, we would prefer to do this in the ABCS layer as transformations are usually associated with adapters.
- **The Validate Semantic operation**: This is provided by the Schematron tool or XSD validation, and the most interesting part of this feature is that we could apply the Partial Validation SOA pattern, checking only part of the message, thereby considerably reducing the time to perform quite an expensive XML operation. As you can see in the following screenshot, we do not use this operation in an outbound call. Why is that? It is because of the following reasons:
    - It is our outbound flow. We believe that here we can trust ourselves. Still, we have the possibility to employ southbound ABCSes to perform the validation.

- ◦ We have already decided that the validation would be part of the execution plan performed on SB. It gives us a lot of flexibility, and we can maintain this operation in a really agnostic way.



t/ns2:task/ns2:invoke/@location) > 0

CustomServiceAsynch::process

Sequential

Validate Semantic

Transform Using: payload : xsl/CTUMessage_To_CTUMessage_request.xsl

Assign Values: endpointURI := $in.request/ns7:CTUMessage/ns0:ProcessHeader/ns0:ExecutionPlan/ns0:taskList/ns0:t...

From	To
expression : $in.request/ns7:CTUMes...	property : endpointURI

From Type: expression

Expression: ı/ns0:taskList/ns0:task/ns0:invoke/@location

To Type: property

Property: endpointURI

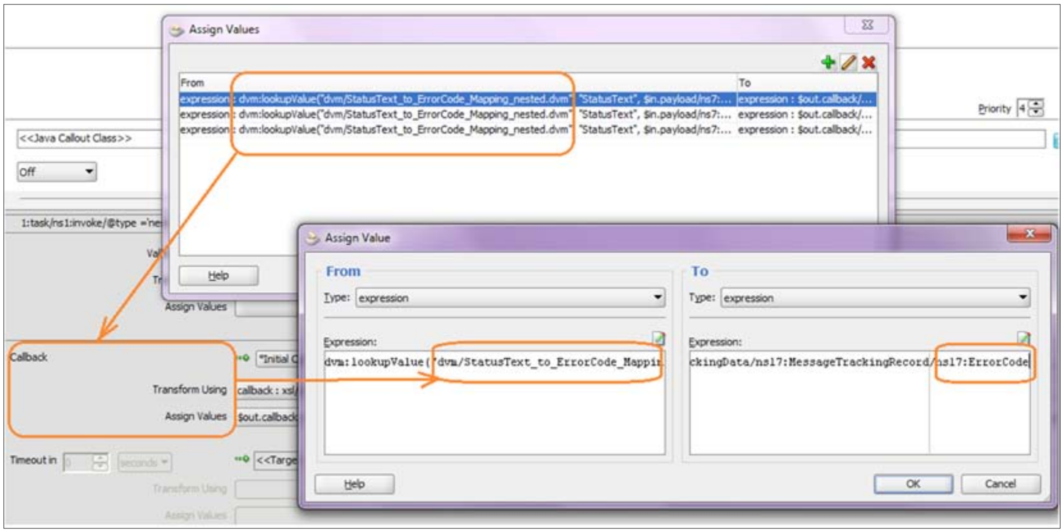
Configuring Mediator's endpoint property

You have to consider this option wisely. With an agnostic payload, we can generally validate `MessageHeader`, `ProcessHeader` (with EP), and `MessageTrackingData`. Logically, it should be done on OSB and/or the adapter layer. Although the Schematrons functionality is really quick, we can achieve the same kind of performance by other means. We suggest that you perform JIT verifications yourself. For this use case, we will not perform it on Mediator.

Inbound flows are easier (again, for this use case only), but practically, all of the outbound functionalities are applicable for inbound flows as well. You can also perform partial/complete validation (we don't). The transformation of SCA's responses is also inevitable, as we will put the response into a container. We can also assign an endpoint for a callback, even for synchronous calls; however, we don't want to do that, as we will handle all the responses in an agnostic controller. Although there is some other Mediator functionality available, we would like to demonstrate it with regard to inbound flows here.

Dynamic compensations in a simple agnostic controller

We want to demonstrate another interesting Mediator feature we used in some inbound flows, although it's not related to any SOA patterns directly at first glance. For some operations within a single service invocation group, we will need to translate the remote response code (errors, messages, and statuses) into generic agnostic controller statuses. Using these statuses, the controller will decide whether to proceed with the next task or execute the compensative action. You can see it in a flows fragment as demonstrated in the earlier screenshot. The task is fairly obvious as external applications, which are not in our Service Inventory, naturally have their own code, statuses, and resolution logic. At best, they will provide you with certain code, reflecting their operational status; however, those situations when some error dumps will be returned are also quite common. Codes and related resolutions should be discussed during the service encapsulation and functional decomposition stage, and we should have at least the initial table with values and code in the form of **Domain-Value Map** (**DVM**). You can create DVM for your SCA application/active project (*Ctrl* + *N*, then navigate to **All Technologies** | **Domain-Value Map**) and store it in your MDS for common use. You can use it in any of your XSLTs as well (you can find plenty of examples on that), but the usage in Mediator is most interesting to us.



From		To
expression : dvm:lookupValue("dvm/StatusText_to_ErrorCode_Mapping_nested.dvm"	"StatusText", $in.payload/ns7:...	expression : $out.callback/...
expression : dvm:lookupValue("dvm/StatusText_to_ErrorCode_Mapping_nested.dvm"	"StatusText", $in.payload/ns7:...	expression : $out.callback/...
expression : dvm:lookupValue("dvm/StatusText_to_ErrorCode_Mapping_nested.dvm"	"StatusText", $in.payload/ns7:...	expression : $out.callback/...

From Expression: `dvm:lookupValue('dvm/StatusText_to_ErrorCode_Mappi`

To Expression: `ckingData/ns17:MessageTrackingRecord/ns17:ErrorCode`

Using DVM in Mediator for error code mappings

The expression present in the **Type** field in the **From** section, realizing the lookup of the error code via status text using DVM in our MDS, is as follows:

```
dvm:lookupValue("dvm/StatusText_to_ErrorCode_Mapping_nested.dvm",
"StatusText",
$in.payload/ns7:CTUMessage/ns17:MessageTrackingData/
ns17:MessageTrackingRecord/ns17:StatusText,
"ErrorCode", $in.payload/ns7:CTUMessage/ns17:MessageTrackingData/
ns17:MessageTrackingRecord/ns17:ErrorCode)
```

As demonstrated in the earlier screenshot, we perform three lookups for `ErrorCode`, `ErrorMessage`, and `ErrorDetails`, which we store in the `MessageTracking` section of the message container, and our agnostic controller will use this value for dynamic compensation. As mentioned earlier, every task has a compensation pair, registered as a sibling in the EP. In cases where there is a recognized error (recognized as a Rollback flag even if it's not an actual rollback), this compensative task will be executed instead of the next one in the sequence; the task counter will not be increased and further execution will depend on the results of this compensation (again, provided via Mediator). The final error resolution will be covered by a dedicated Error Handler, which is part of the conventional BPEL compensation flow, and we will discuss it in detail in *Chapter 8*, *Taking Care – Error Handling*.

> This simple but effective solution does not eliminate the necessity of full-grown error handlers and compensation flows. We are just demonstrating how DVM can help with basic resolution actions and execution status tracking.

Of course, compensative branching logic in an agnostic controller must be strictly conducted as per the values you put in DVM. DVM can be modified at runtime, changing the business logic brokered by an agnostic controller; however, if you misspell a word, the situation can be unpredictable. (Another question is who would do that directly in production? Sadly, we all know the answer.) Another issue that must be taken into consideration is what if the business task does not (or could not) have a dynamic compensation routine? For simplicity, we didn't show this scenario, but this situation can be anticipated. In most common cases, we can ignore this condition and continue with the next task or park it in Human Workflow to come up with a manual resolution.

So, now you can see that Mediator can do a lot as a static router:

- Dynamically substitute the endpoint URI, which is most interesting for us here.
- Substitute the values using DVM, stored in MDS. Here, we can partially implement the Exception Shielding SOA pattern, improving our security landscape (do not rely on that much though).
- Perform all kinds of transformations for invocations and callbacks.
- Perform partial validation, implementing the Partial Validation SOA pattern, thereby improving performance.

Other things we didn't discuss here (yet) are as follows:

- Event propagation, implementing Event Driven Messaging (the Event Aggregation Programming model in AIA terms)
- Using Java callouts in message mediation and process handling
- Rule-based dynamic routing

Actually, now we are going to discuss the implementation of the decision service that is responsible for selecting the appropriate execution plan and then we will continue with Mediator.

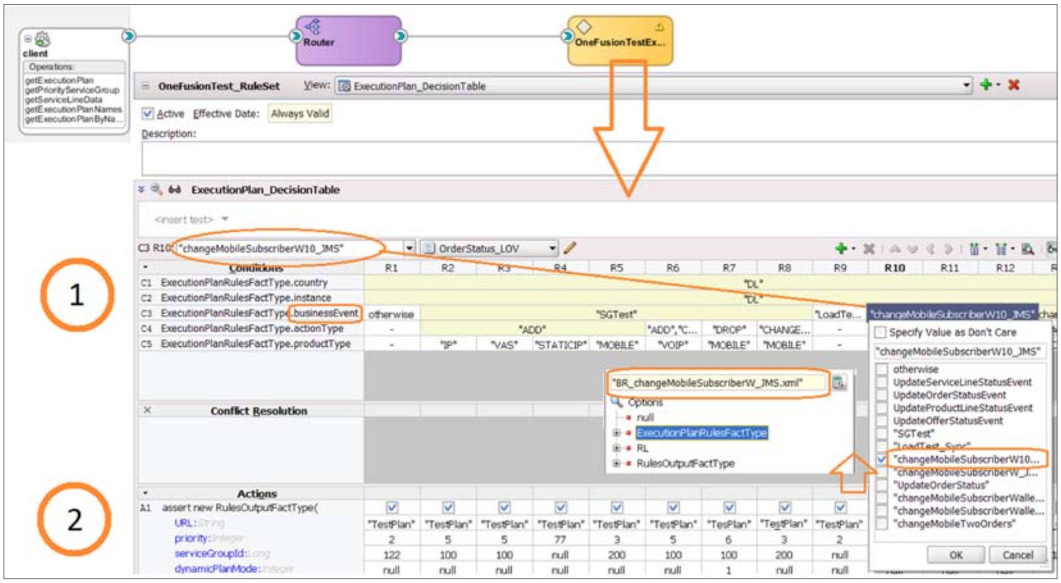
The Rule Engine endpoint and decision service

In the third step depicted on the consolidated agnostic Composition Controller SCA diagram, we called `PlanLookupService` to acquire the routing slip (EP). This service is clearly visible among others in the list of target services that are available for invocation by our Service Broker. You already know the basic structure of EP and the `MessageHeader` elements used as input parameters; therefore, establishing WSDL for this service should not be a problem. Simply put, this is all that we know: the requested product, county/affiliate, business event, and the requested business operation. This information should be sufficient for determining which business process (in the form of EP) must be fired (namely, EP's name extracted and passed for its execution in Business Delegate within a loop). Consequently, these parameters will be the elements of the Business Rule components' inbound and outbound variables, where outbound is just the name of an XML object (file) stored in MDS.

To begin with, we design this SCA Composite in an exceedingly simple way; it should only contain Mediator and the Business Rule component. We will start the configuration from business rules. The configuration steps are obvious; just follow the Oracle documentation:

- The XSD schema will be created with two types, reflecting the inbound and outbound elements (hint: see the transformation part in the following screenshot). Feel free to adapt the types according to your requirements or just simplify it, leaving no more than three elements for the start.
- Drag the rules component from the right palette, and assign the input and output variables from the defined XSD (input is based on the `MessageHeader` XSD and output just on a string).
- These input and output variables will be our XSD-based rule facts.
- It would be useful to define baskets as **List of Values** (**LOV**) for country, affiliate, product, and business operations (action types). Think of baskets as enumerators with fixed values that we will use in our decision table.

Now, we can create our decision table as shown in the following screenshot:



OneFusionTest_RuleSet View: ExecutionPlan_DecisionTable

☑ Active Effective Date: Always Valid

Description:

ExecutionPlan_DecisionTable

<insert test>

C3 R10: "changeMobileSubscriberW10_JMS" OrderStatus_LOV

Conditions	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12
C1 ExecutionPlanRulesFactType.country	"DL"											
C2 ExecutionPlanRulesFactType.instance	"DL"											
C3 ExecutionPlanRulesFactType.businessEvent	otherwise	"SGTest"							"LoadTe...	"changeMobileSubscriberW10_JMS"	cha	
C4 ExecutionPlanRulesFactType.actionType	-	"ADD"				"ADD","C...	"DROP"	"CHANGE...	-			
C5 ExecutionPlanRulesFactType.productType	-	"IP"	"VAS"	"STATICIP"	"MOBILE"	"VOIP"	"MOBILE"	"MOBILE"	-			
Conflict Resolution												
Actions												
A1 assert new RulesOutputFactType(	☑	☑	☑	☑	☑	☑	☑	☑	☑			
URL: String	"TestPlan"	"TestPlan"	"TestPlan"	"TestPlan"	"TestPlan"	"TestPlan"	"TesPlan"	"TestPlan"	"TestPlan"			
priority: Integer	2	5	5	77	3	5	6	3	2			
serviceGroupId: Long	122	100	100	null	200	100	100	200	null			
dynamicPlanMode: Integer	null	null	null	null	null	null	1	null	null			

"BR_changeMobileSubscriberW_JMS.xml"

Options
- null
- ExecutionPlanRulesFactType
- RL
- RulesOutputFactType

☐ Specify Value as Don't Care

"changeMobileSubscriberW10_JMS"

☐ otherwise
☐ UpdateServiceLineStatusEvent
☐ UpdateOrderStatusEvent
☐ UpdateProductLineStatusEvent
☐ UpdateOfferStatusEvent
☐ "SGTest"
☐ "LoadTest_Sync"
☑ "changeMobileSubscriberW10...
☐ "changeMobileSubscriberW_J...
☐ "UpdateOrderStatus"
☐ "changeMobileSubscriberWalle...
☐ "changeMobileSubscriberWalle...
☐ "changeMobileTwoOrders"

OK Cancel

Building the decision table for Agnostic Composition Controller

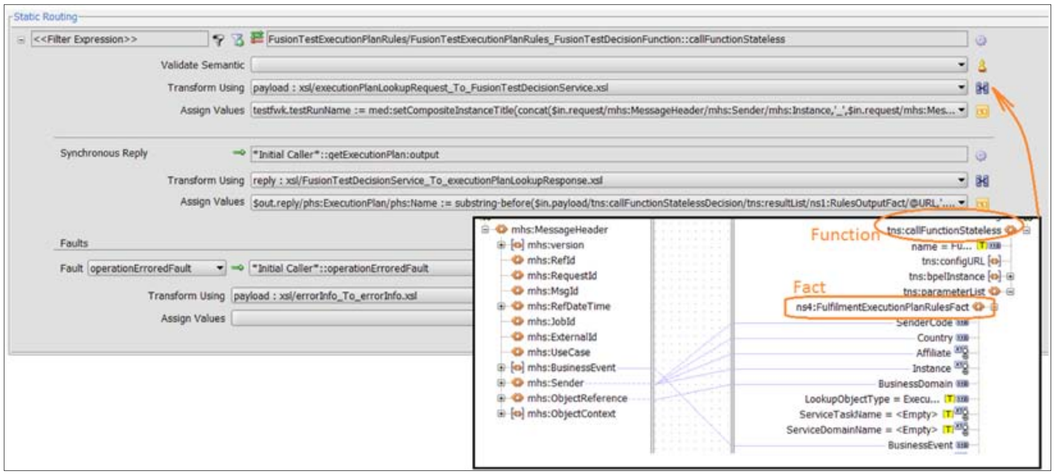
The earlier table is similar to any table you can create for this purpose (in Excel, for instance). There are two parts. In the first part, using basket LOVs, we define the conditions provided by inbound parameters such as productType: **VOIP**; businessEvent: **UpdateOrderStatus**; country: **BR** (for Brazil); and affiliate: **BraTello** (for Brazil Telephonic). In the second part, we define actions, which in our case is the filename of the Execution Plan. Needless to say that this file is stored in MDS, and we know the actual path; it's part of the deployment profile and configuration plans. You do not have to describe the whole complexity of business rules and conditions from the start; taking one small step at a time is important, and we would really like you to repeat it. Obviously, the execution plans and their names will emerge gradually (and not exactly slowly) along the way of the functional decomposition process together with logic centralization and service refactoring (all of them are SOA patterns, representing the common sense we mentioned many times).

Apparently, the ultimate outcome of this practical exercise would be the establishment of a concrete Service Inventory, which is currently maintained on MDS. We are not going to discuss the pros and cons of this realization now; we're just jumping ahead, but we can assure you that we will rebuild it for better association with SOA Patterns and as a result provide better performance, maintainability, and scalability. Anyway, you can be rest assured that the presented solution is already solid enough and it works!

We could only have the Business Rule component in our EP lookup composite, but again, we will add the Mediator. Why is that? For its transformation and routing support, of course. Again, following the SOA principles, we want to maintain `PlanLookupService` in an atomic state, decoupled from the `MessageHeaders` implementation. The MH XSD can evolve, following our understanding of business requirements and the SBDH standard; also, we definitely do not want to affect our Rule Facts, which are already implanted into the decision service. Another important thing is the flexibility of the Rule Engine itself. We can have more than one ruleset, supporting multiple decision tables or rule functions. Thus, we should not only transform message headers' values from an inbound request into rule function parameters (see the `callFunctionStateless` transformation in the following screenshot, the Transform Using field), but also route individual requests to the specific decision table (or rule function). Static routing will do perfectly fine here.

You can see how many operations we can support from the client's perspective with our service in the previous screenshot from the previous paragraph; all operations are synchronous, which is logical because async MEPs simply will not work here.

Talking about the Rule Centralization SOA pattern, we have to stress the fact that concentrating on all the rules in the form of heavy decision tables in one decision service is not a good idea at all. The rule of the thumb here would be to centralize all the business delegates' decision tables in one decision service and separate it from other business domains (see the corresponding `BusinessDomain` element in the following mapping):


Static Routing

<<Filter Expression>> FusionTestExecutionPlanRules/FusionTestExecutionPlanRules_FusionTestDecisionFunction::callFunctionStateless

Validate Semantic

Transform Using payload : xsl/executionPlanLookupRequest_To_FusionTestDecisionService.xsl

Assign Values testfwk.testRunName := med:setCompositeInstanceTitle(concat($in.request/mhs:MessageHeader/mhs:Sender/mhs:Instance,'_',$in.request/mhs:Mes...

Synchronous Reply *Initial Caller*::getExecutionPlan:output

Transform Using reply : xsl/FusionTestDecisionService_To_executionPlanLookupResponse.xsl

Assign Values $out.reply/phs:ExecutionPlan/phs:Name := substring-before($in.payload/tns:callFunctionStatelessDecision/tns:resultList/ns1:RulesOutputFact/@URL,'....

Faults

Fault operationErroredFault *Initial Caller*::operationErroredFault

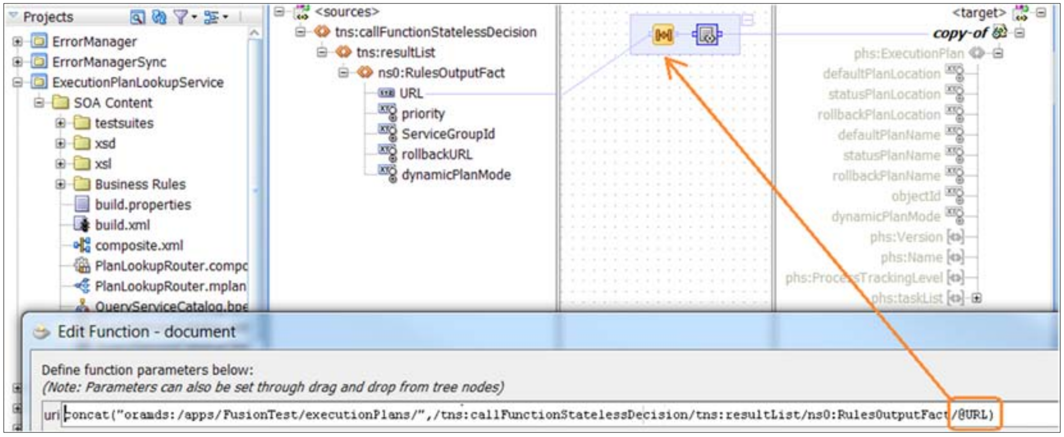
Transform Using payload : xsl/errorInfo_To_errorInfo.xsl

Assign Values

Acquiring an XML object using Mediator

Handling a synchronous reply is easier as we have only one element to deal with, that is, the URL of the actual execution plan. Here as well, we have to use transformation with one simple `concatenation` function to combine our `oramds:` path with the `<EP>.xml` filename; you can see it in the following screenshot. Now, back to the Mediator routing rules and transformations presented earlier in the chapter, you can see more clearly how we transform our Execution Plan as an XML object.

The next operation in the row `Assign: Retrieve_ExecutionPlan` is just to assign the whole of the reply to the EP type in the Messages Process header. (Yes, "retrieve" probably is not the best term here as it is in fact an inject, but you get the idea.)



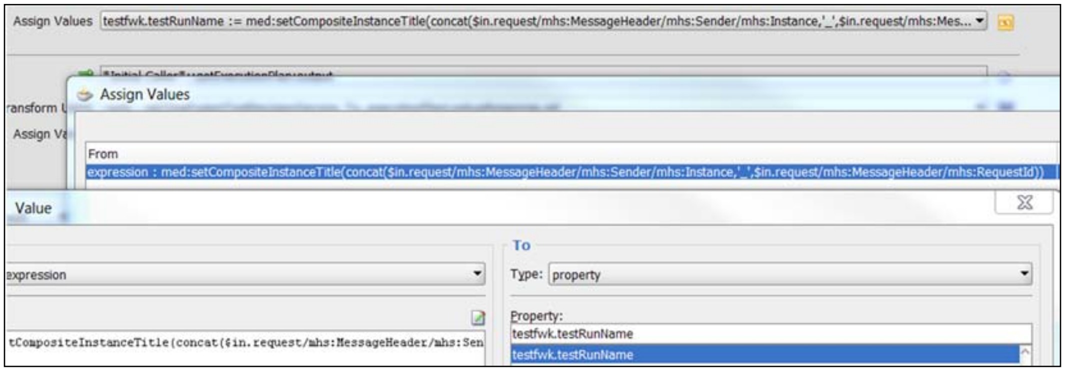
Define function parameters below:
(Note: Parameters can also be set through drag and drop from tree nodes)

uri `concat("oramds:/apps/FusionTest/executionPlans/",/tns:callFunctionStatelessDecision/tns:resultList/ns0:RulesOutputFact/@URL)`

Concatenating XML objects in Mediator's transformation

Using Mediator for process discoverability

That's it! We're almost done here. One more very convenient thing related to Mediator should be mentioned. Although it's not directly related to any SOA message mediation patterns, this feature directly supports Runtime Discoverability, and by enforcing this SOA principle, we can greatly improve the lives of our operational personnel. We all know (as diligent architects who support and control the health of deployed applications) that with a considerable number of business processes, tracing them in the SOA EM dashboard could be hard. The native SCA ConversationID could be ugly, and leaving the Instance name empty is not a decent courtesy towards those who will deal with our products. Regretfully, our dynamic approach, as you already guessed, does not improve the situation much; quite contrary to this, it will complicate it. We do not have the actual business process name until we call the EP lookup process. We should assign the instance title somehow in the request or preferably in the response to make it more searchable and traceable.



Assign Values testfwk.testRunName := med:setCompositeInstanceTitle(concat($in.request/mhs:MessageHeader/mhs:Sender/mhs:Instance,'_',$in.request/mhs:Mes...

Assign Values

From

expression : med:setCompositeInstanceTitle(concat($in.request/mhs:MessageHeader/mhs:Sender/mhs:Instance,'_',$in.request/mhs:MessageHeader/mhs:RequestId))

Value

expression

tCompositeInstanceTitle(concat($in.request/mhs:MessageHeader/mhs:Sen

To

Type: property

Property: testfwk.testRunName

testfwk.testRunName

Setting the Composite Instance title using Mediator

Here again we can use Mediator properties to complete the task; see the previous screenshot. Feel free to select any meaningful fields from your message in the XPath function call, as presented in the next code snippet (here, `med:` stands for Mediator's namespace). Now we can assign the concatenated value to the mediator property. In the Oracle documentation, `tracking.compositeInstanceTitle` is mentioned, which is obviously invisible. In our example, we use `testfwk.testRunName`, which is pretty good as well:

```
med:setCompositeInstanceTitle(concat($in.request/mhs:MessageHeader/
mhs:Sender/mhs:Instance,
'_', $in.request/mhs:MessageHeader/mhs:RequestId));
```

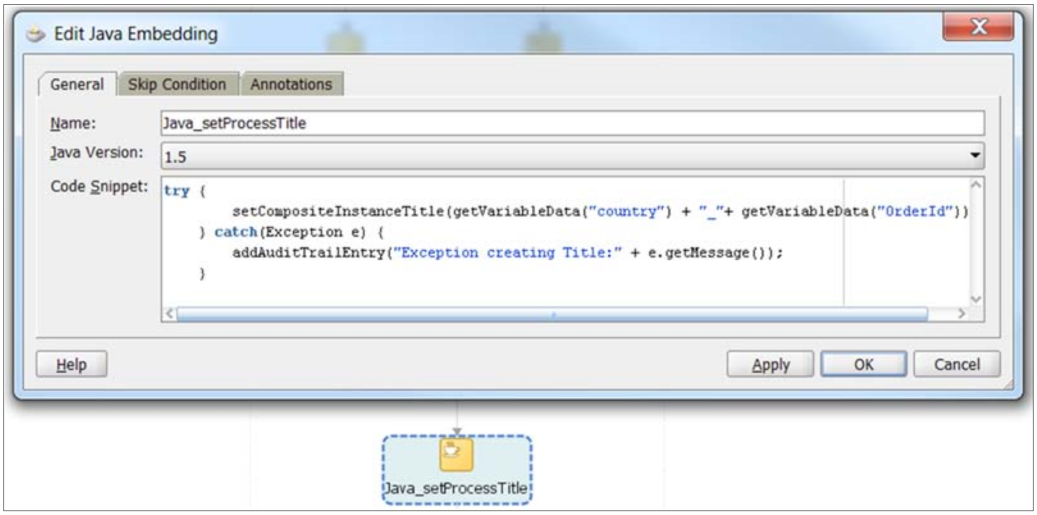
You can imagine that this is not the only way to set the Instance title.

The Orchestration pattern and embedded Java

In a book that is dedicated to practical aspects of SOA patterns, we cannot devote much space to Java, as we are supposed to stay reasonably language-neutral (one of the SOA benefits, you remember). Sure, it's impossible when we talk about concrete implementations on the Oracle platform where Java is the blood and bones of the entire ecosystem. Thus, some words about it are in order here.

We deliberately omitted some steps at the beginning of our Business Delegate (or Service Broker) process. They are quite typical and are as follows:

- We have to set appropriate audit and tracing levels, identifying them from the message context (the Message Header is involved).
- We want to set the instance title from the very beginning, again using Message Header values. This can be done using embedded Java; see the following screenshot:



Name: Java_setProcessTitle

Java Version: 1.5

Code Snippet:

```
try {
        setCompositeInstanceTitle(getVariableData("country") + "_"+ getVariableData("OrderId"))
    } catch(Exception e) {
        addAuditTrailEntry("Exception creating Title:" + e.getMessage());
    }
```

You can do a lot using Java callouts in BPEL, but be reasonable and do not reinvent existing functions. Other areas of direct Java application would be Mediator's Java Callouts and BPEL custom sensor actions; you can register custom classes with the SOA Suite.

Summary

Using an interactive approach, we gradually constructed our first SOA framework that is responsible for holding, running, and maintaining our Enterprise Business Flows (in Oracle's AIA notation). Although AIA denotes this framework as optional, in reality it's the most common and heaviest element of the integration infrastructure in almost any enterprise. Consequently, the source of practically all the problems in this framework is the understanding of business flows as part of a solid integration, sometimes point-to-point and not service collaboration.

Orchestration, according to the SOA pattern catalog, is the biggest compound pattern, comprising many atomic patterns that are responsible for turning the integration approach into a more agile service-oriented computing. As demonstrated in this chapter, the first step would be the proper categorization and decoupling of our bulky services according to their models and levels of granularity. Reassembling these services into managed compositions should not always be done in a static way, especially for services of a similar business nature with slight composition variations. Despite the nature of a static BPEL (rather imaginable), it's quite possible to implement agnostic Composition Controllers that are capable of assembling business processes dynamically. At first, we must stay focused on business process parts, which can be automated without manual interventions; Oracle SCA components such as BPEL, Mediator, and Business Rules are the main building blocks, and they are powerful enough to deliver any Orchestrated solution.

We also demonstrated a problem-focused approach, where we do not lock on to particular patterns, but rather strive to alleviate the negative sides of the existing solution by applying the most fitting method (based on pattern, if possible). By practicing this approach, we demonstrated a flexible way of understanding patterns' boundaries using Service Broker and Intermediate Routing patterns, which are originally bound to OSB in the Orchestration framework. We kindly advise you to follow the same "pattern", so to say, not focusing on the patterns' catalog, but analyzing the problem according to SOA principles, decomposing it into manageable pieces, and refactoring components according to service-orientation if possible. Also remember that ultimate reusability can come at a price you simply could not afford, so balance your efforts wisely, again using SOA principles.

The presented part of the solution (Orchestration) in the beginning is completely functional and is taken from a real-life project, although the company is fictitious, of course. Now you have enough samples to build your own controller, MDS, and service repository; however, please bear in mind that this is just a beginning, and we will show you the ways to improve it.

This chapter just begins the functional decomposition process. It is evident that we have synchronous and asynchronous service brokers, acting as agnostic Composition Controllers that are maintained on the Orchestration platform. In the next chapter, we will separate them using the appropriate patterns.

4

From Traditional Integration to Composition – Enterprise Business Services

In the previous chapter, we used Foundational Service Patterns and Composition Implementation Patterns along with elements of a Compound Orchestration Pattern to start the gradual refactoring of a complex telecom service landscape (refer to the *Design Patterns (by category)* section at `http://soapatterns.org/`). After the initial analysis and redesign, we came up with the first working model of a service orchestration layer, and this layer is equipped with Universal Agnostic Controller; it serves both long- and short-running compositions. Although completely operational, this solution is still far from optimal. Thus, we dedicate this chapter to further optimization of solutions through continuous service-oriented analysis, identification of common problems, and application of solutions in the form of commonly approved SOA patterns.

As an immediate result, we expect the following benefits:

- The segregation of synchronous and asynchronous layers
- Establishment of the service abstraction layer with low cohesion and improved APIs
- Improved loose coupling between the controller's components
- Refined platform reliability and composability

We will continue our practice of chaining appropriate SOA patterns together in a logical fabric, as demonstrated in the previous chapter, starting from the analysis phase. You will get an understanding of **Pattern-Oriented Software Architecture** (**POSA**) in relation to the SOA domain at `http://www.cs.wustl.edu/~schmidt/POSA/`, although this is not our primary goal here.

A practical outcome will be establishing the synchronous Agnostic Composition Controller based on the OSB functionality within the **Enterprise Business Services** (**EBS**) framework boundaries and the demonstration of the OSB Adapter Factory capability, the service proxy, and the Service Facade. You will also see how to (re)use the Rule Engine (RE) decision service in OSB. As this dynamic controller solution can be a bit too complex to dive in to directly, we will start with the demonstration of a simple Java-based Message Broker in order to explain some concepts.

The Dynamic Service Collaboration platform

According to the Oracle methodology formalized in the AIA approach, Enterprise Service Bus is the only mandatory pattern in the enterprise SOA infrastructure, which is maintained as an **EBS** framework. That's what we have learned in the first two chapters. At the same time, in our practical exercise in the previous chapter, we started with Enterprise Business Flows, discussing the orchestration patterns and not ESB. How is it possible to have an optional core? Is it really a core? Yes, it is. First of all, **Enterprise Business Flows** (**EBF**), which are offered as task-orchestrated services, are the closest entities that represent our business and how we understand it. No wonder most of the IT initiatives related to SOA are carried out by the BPEL implementations (since Oracle 10*g*), despite the service's nature, their MEPs (synchronous or asynchronous), and the necessity of orchestration in general. The telecommunication primer is pretty common as seven out of 10 companys' architects associate the SOA patterns with orchestration alone, and this legacy has to be refactored sooner or later, and so have we done it.

As we have already demonstrated, the SCA dynamic Composition Controller is able to carry out reliable broker services for all the MEP types within an existing business domain (order provisioning) and between other domains in OSS/BSS. The actual numbers achieved with this approach are presented in the following table:

Typical OSS/BSS NFRs	**Metrics**	**Operational conditions and settings**
Average number of transactions	30,000 orders daily (60,000 peak)	This workload has been handled with one fault per 50,000 orders. The most common reason is that the input is not according to XSD.
Asynchronous transactions	2 minutes to 14 days	This depends on the type of order and requested product. No message loss takes place.

Typical OSS/BSS NFRs	Metrics	Operational conditions and settings
Synchronous transactions (Stress test)	800 ms for an execution plan with nine consecutive tasks	• **soa-infra**: Select **SOA logs | Log configuration**: **Level**: **ERRORS**; • **soa-infra**: Select **SOA Administration | Common Properties**. Set **Audit Level**: **Off** • **Capture Composite Instance State**: **Off**
Synchronous transactions (Production)	1500 ms for an execution plan with nine consecutive tasks	• **soa-infra**: Select **SOA Logs | Log Configuration**; **Level**: **WARNING** • **soa-infra**: Select **SOA Administration | Common Properties**; **Audit Level**: **Production** • **Capture Composite Instance State**: **Off**

In the preceding table, you can see that we can maintain less than 100 ms per operation (invocation) with audit / logging set to `OFF` in SCA. Of course, in production, the required numbers are almost doubled. With some more fine-tuning, we could gain 10 to 15 ms more, but apparently these numbers are rather inadequate for fast-running synchronous transactions. This is just one obvious thing that we can spot immediately. We should continue improving the solution using a service-oriented approach.

> The presented numbers were achieved on a double CPU 2Core VM with 8 GB memory and single node SOA Suite installation. All endpoints were mocked using SoapUI.

Improving the Agnostic Composition Controller

The implementation of a redesigned service collaboration platform with the orchestration pattern in the OSS/BSS business domain was recognized as a major success. Other departments decided to implement the existing functionality on their premises or, even better, they decided to reuse the existing frameworks directly as a Service Broker / Mediator to develop the service inventory. However, some concerns were expressed. It is clear that to improve performance, we have to separate the synchronous and asynchronous service activities. Together with the orchestration platform's performance fine-tuning, we will probably gain some milliseconds.

However, this is not an ideal approach. Thus, a synchronous part that does not require the Partial State Deferral pattern implementation shall be moved to the Enterprise Service Bus we are about to build. This means that the Dispatcher and Decision Service (another instance) elements of the functional decomposition figure in the previous chapter shall be redesigned in a new framework. Not only that, the Business Delegate (`http://www.corej2eepatterns.com/Patterns2ndEd/BusinessDelegate.htm`) that decouples the service consumer / composition imitator will continue to play its role in the SCA realization of the service broker by performing the following tasks:

- Minimize the number of calls from the client to the service provider, acting as a transaction coordinator for logically aggregated business operations
- Hide the networking issues associated with the distributed nature of services
- Shield the clients from possible volatility in the implementation of the business service API
- The Business Delegate should be implemented on ESB, somewhere close to the repositioned dispatcher (for performance reasons)

These roles associated with the maintenance of the Loose Coupling principle are traditionally linked to implementation of the Proxy pattern of services/components. This design pattern was one of the most popular patterns (before SOA) to wrap the physical component in an additional layer to control access, maintain the distributed nature of an asset by delegating interoperability functions, and hide the complexity of the implementation. With emerging enterprise services, especially on the WS technology, this pattern becomes even more demanding, as the detachable service contract fits really well into the Proxy concept and makes consumer-provider decoupling easier and more native. This design pattern is intrinsically connected to some other generic (non-SOA) design styles, and it's the basis for some fundamental SOA patterns. In Oracle ESB realization, you basically put all actual development behind a proxy. Thus, by following the POSA promise, we will start with this pattern and explore its relations with other ESB patterns.

The Proxy design pattern and its relatives

We have already mentioned some tasks that are explicitly related to traditional Proxy realization. We will definitely employ a Proxy pattern when we have certain needs, such as:

- Establishing the local representation of a remote object that may exist in another network segment or domain. This primary and classical role of Proxy has a long history with COM/COM+/DCOM and CORBA.

- Protecting our object; a protective Proxy can be established where access permissions can be verified. Reverse Proxy is one of the forms of protective Proxy.
- Establishing the placeholder for the object that we will create only upon the client's request. This virtual Proxy can help us save a lot of physical resources.
- Implementing a smart, type-based Proxy that can check for the presence and state of the object before passing the request, or that can act as a master objects reference table, counting number of object instances and destroying unused ones.

So, Proxy is a decoupling layer between the client and provider, safely plugging the requestor to the subject, allowing them to live or evolve independently at the same time. Can it be described as an Adapter pattern then (another generic pattern)? Yes? No? Yes? Perhaps more like maybe. Talking seriously, in classic design, adapters must (according to their name and nature) adapt inbound signals to the receiver's capabilities expressed in a contract. Thus, the Transformation and Bridging traditional patterns are native companions of an Adapter, while Proxy is supposed to provide the same interface. Of course, nothing is set in stone here, and we are just outlining the patterns relation according to generic POSA, with some common sense for spice.

What if we want to extend the existing service contract, add a new capability, or change the granularity for one or several contracts elements (functions and not data types)? In OOP, it's similar to extending a class (subclassing), but we would still like to maintain decoupling. As we know, OOP inheritance is a useful thing, but it certainly introduces tight coupling in the inheritance hierarchy. What is good at the component-level architecture is not exactly positive and desirable at the next higher level (service layer) within the composite architecture, and further in the service inventory. Also, sometimes, this kind of enhancement must be performed at runtime; therefore, OOP is not really applicable to its inheritance.

In real-life situations, with so many variations added to the superclass through inheritance, we can easily introduce a nightmare of the so-called "class explosion". These complications can be resolved by the application of the Decorator pattern. The purpose of a Decorator is to "mash up" the capabilities of a concrete class with new capabilities to present a new compound contract. Usually, this is done by creating an abstract decorator, which is a proxy to the main object (in fact, it's an exact replica of the abstracting class), and concrete Decorators that enhance the actual object's methods. So, here, the role of Proxy is clearly visible and the link between Proxy and Decorator is apparent (compare this with the Adapter-Proxy relations). With regards to the SOA design, the Decorator pattern is implemented as a composite at the component level, and this pattern can be applied in the composite application layer as well.

The Decorator pattern is also linked to the adapter's pattern implementation. This relation is not straightforward though. It is better to see it from the Decorator's downside. Although we have replaced the inheritance by composition (except Abstract Decorator-Proxy), we can still end up with tons of concrete Decorators. Yes, we have achieved a reasonable level of decoupling, but at what cost? We still have a lot of static modules (that is, enhanced interfaces) to maintain; wrap them into Decorators and explicitly instantiate them. It would be good if we could instantiate them blindly through the interface without knowing about concrete implementation. Actually, this is possible by using the Factory Method pattern, which provides an interface to create (access) objects without knowing their realization. Of course, all objects in a collection must be of the same type. We can go one step further in our abstraction and implement the Abstract Factory pattern on the top of concrete factories in order to access/instantiate different object collections. Now, decorators along with Abstract Factory will not only decouple a caller from the object (provider) with the enhanced interface, but it will also allow us to access this interface without knowing its explicit specification dynamically.

So, a Decorator as a wrapper could work well together with Abstract Factory. Adapters can be seen as a wrapper as well. In real life, we don't always deal with only one adapter (unfortunately), as we have many interfaces to adapt. Thus, our collection of adapters must be abstracted first and then accessed in a dynamic way, without the knowledge of an actual consumer/provider implementation and depending on the invocation directions (remember, we have northbound and southbound sides of the same ESB). In this case, Abstract Factory can be evolved into Adapter Factory.

This pattern is not exactly from the traditional pattern catalogues (for SOA, visit `http://soapatterns.org/`, or for OOP, check out `http://www.oodesign.com/`), but its necessity is obvious. Although every physical adapter is an individual wrapper to the concrete API (not always legacy), transformation, validation, filtration, and enrichment are common functions associated with the traditional adapter pattern. It is also obvious that JCA technology/protocol adapters (for file/FTP, JMS MQ, AQ, and in most cases, DB) also have a lot of common properties. This allows us to group them under the factory methods, which will do the actual dispatching to specific adapters.

Abstract Factory, which acts as a dispatcher, should be isolated from the client by the Facade with a simplified interface that accepts Message Container. In fact, in JEE terms, it's a Transfer Object (`http://www.oracle.com/technetwork/java/transferobject-139757.html`) with the Business Object inside and all elements in a header, necessary for obtaining the instance of the required protocol adapter from Adapter Factory.

In this case, the Factory Method pattern acts as a dispatcher between adapters of the same technology/protocols, Abstract Factory mediates between groups of transport adapters, and Facade isolates client calls from Adapter Factory and performs all necessary tasks on Transfer Object (extract headers elements, for instance). Validation and transformation of business payload should ideally be done on Adapter in order to keep Adapter Factory's Facade completely generic, but this task can be generalized as well by using OSB; we will be discussing this soon.

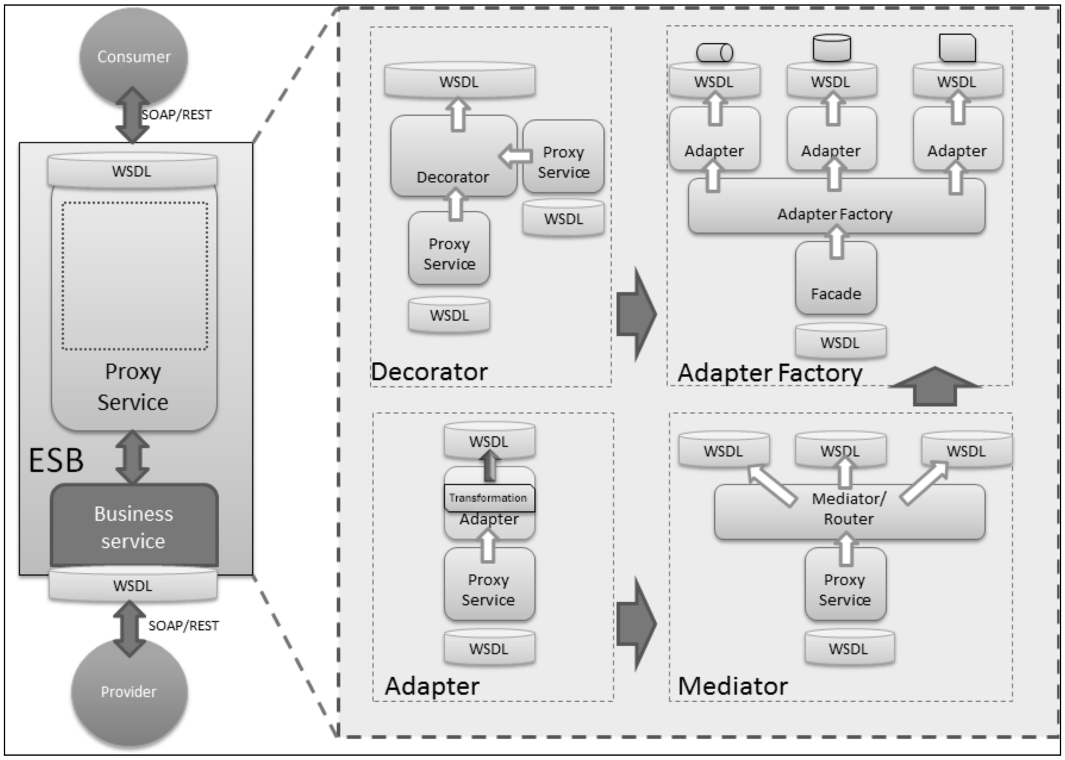
We just mentioned another pattern related to Proxy, that is, Facade. If we want to simplify the interface, make it more generic and universal (in other words, abstract), we should put the Facade upfront. It's similar to the Decorator pattern, but we are not adding a new method to the abstracting (wrapping) interface; we are simplifying it. Actually, it's quite the opposite. The whole point here is the unification of access to the object/API. We are hiding the complexity of the existing implementation and moving the delegation to the underlying methods upon receiving the call. Thus, it's pretty close to Abstract Factory and not Decorator. Is it some kind of Adapter? Not really. The Facade pattern implements a new, simplified interface, while Adapter strives to preserve the existing one. This key difference is crucial, as it defines the physical location of Facade in SOA topology and we will get back to it when we will discuss the SOA reincarnation of these traditional patterns. You can spot one relation already: our SCA Mediator is in fact the Facade!

> We left Mediator in the SCA orchestration layer, and now we have mentioned it while talking about traditional patterns in ESB. Confusing? Yes, if you believe that the SOA patterns catalogue is something fixed and predefined. Here, we are showing the application of patterns in a working platform, in relation to other patterns and physical boundaries, are limited by the logic of the SOA principles and characteristics we want to achieve.

Mediator is one of the core patterns that establishes intermediate routing between sender(s) and receivers (for OOP, visit `http://www.oodesign.com/mediator-pattern.html`, and for GoF, `http://www.eaipatterns.com/Introduction.html`). Mediator makes the senders and receivers reference each other indirectly. There are some other traditional patterns that implement routing and referencing, statically or dynamically, but we will stay with the more traditional definition of Mediators. Someone can describe Mediator's core functionality as **Receive-Transform-Deliver** (**RTD**), which is exactly opposite to ETL), but Transformation is not a traditional design pattern and it is mostly associated with the Adapter implementation. By combining Mediator and Transformation (from Adapter, both for Data Format and Data Models), we essentially implement the Message Broker pattern, which is the predecessor of traditional ESBs. Practically, all hub-and-spoke integration patterns are based on a compound Message Broker.

Due to its simplicity, it is one of the most popular lightweight implementations of traditional EAI patterns and it's not that easy to draw a line between Message Broker and Service Broker in the SOA ESB. In fact, for the services that have strictly predefined contracts (REST Services, which are based on HTTP POST/GET operations) and are exposed as lightweight endpoints, Message Broker could be the most optimal solution from performance and maintenance standpoints.

Basic relations between the discussed traditional patterns are illustrated in the following figure:



Our short walk through Proxy and other related patterns can be summarized as follows:

- Most of the patterns discussed in the preceding sections are from OOP and traditional components design practices. Their area of application lies mostly within the component/service architectural layer, and they are governed by the principles that are not always in accordance with the SOA design principles. Inheritance is quite opposite to Loose Coupling.

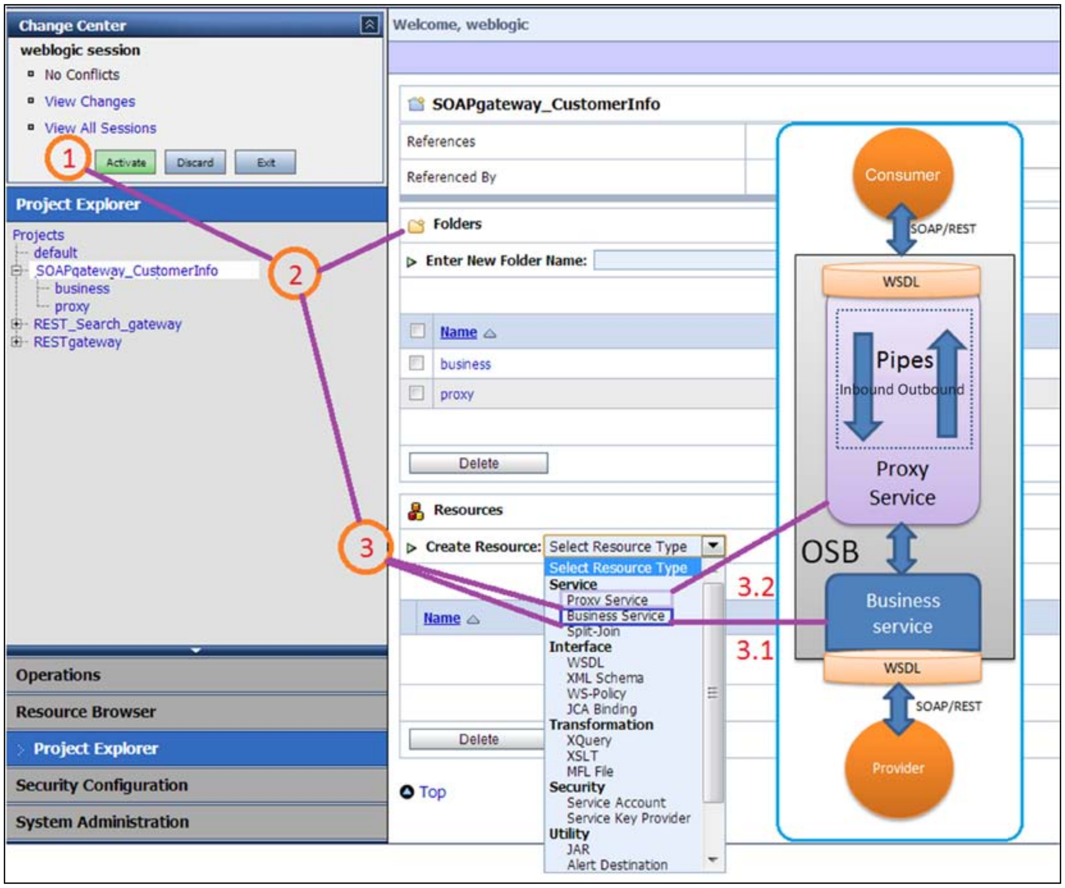
- The preceding fact must not discourage us from learning and adapting these patterns to work in an ESB framework. Contradiction of principles is a regular occurrence, even the SOA principles are not always friendly to each other. Besides, OOP and traditional integration are major contributors to SOA.
- Proxy is a service governance pattern that is actively used in more complex design patterns. Its primary role is to decouple service consumers and service providers. Generally, it is either an object wrapper or a remote placeholder for the object that presents the same interface to its subject.
- Adapter is a wrapper as well, but it offers a different subject's interface (adapted to consumer). This allows dissimilar objects to communicate. Transformation (model or format) is one of the functionalities of an Adapter. Protocol bridging for disparate protocols is another functionality.
- The Decorator pattern enhances an existing interface by adding new method(s).
- Facade implements a simplified and more abstract interface suitable for invocation by clients with different requirements, as it introduces a consolidated view of enterprise objects. Abstract Factory has a similar purpose.
- In order to guarantee smooth and reasonably painless service evolution, you must plan two placeholders for the Façade pattern implementation in a new service design: one between the contract and service logic and another between the service logic and the underlying resources.
- Abstract Factory can also abstract and wrap dissimilar interfaces together with Adapter patterns, effectively introducing Adapter Factory with the main purpose of dispatching a request to the particular interface.

Now, it will be interesting to see how these EAI patterns can be implemented on an Oracle realization of Service Bus.

Implementing a basic Proxy on OSB

Although we have no intention (and primarily, bookspace) to supply you with detailed step-by-step OSB tutorials, some guidelines to create the basic OSB resources and Proxy in particular will be in order. We are certain that you will find plenty of good guidelines on the Internet or in the Packt Publishing bookstore. However, since OSB is the only mandatory SOA framework and the Proxy pattern is the heart of the OSB, we will walk through the creation process in just three and a half quick steps.

Have a look at the following screenshot:

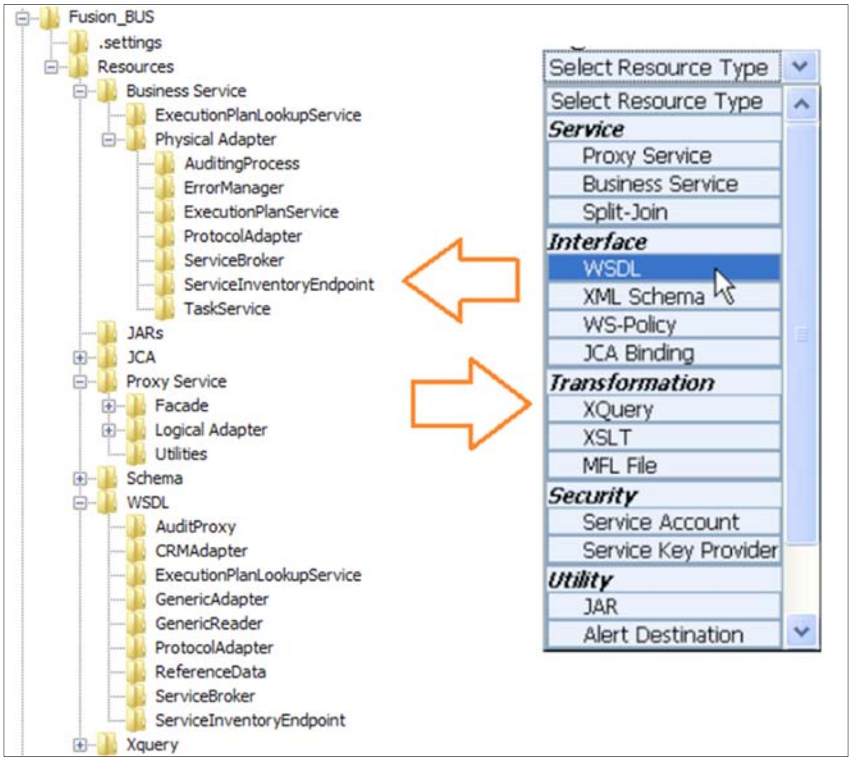


In order to demonstrate the different Oracle tools to handle OSB, we will create OSB artefacts (Business and Proxy services in particular) by using the OSB console (`http://<your-host>:7001/sbconsole`) and not Eclipse. According to WLS practice, any changes in the resource configuration shall be done during an alteration session:

1. In **Change Center** of the **Oracle Service Bus**, click on **Create** to create a new session. If something goes wrong, you can always click on **Discard** (and lose all of your changes). However, we believe that we will be happy with OSB. So, after completing all the required steps, we will activate our changes.

    1. In the **Oracle Service Bus Console** navigation pane, select **Project Explorer**.

2. In the **Enter New Project Name** field, in the **Projects** section, type the name of your project. In the preceding screenshot, we used `SOAPgateway_CustomerInfo` from one of our test cases. You can use any other name as per your convenience.
3. It will be a good practice to dedicate individual folders in our project space for every resource that we will deploy or abstract in OSB. For example, **Proxy Service** (Proxy), **Business Service** (Business), and **WSDL**.



Some common XML-based artifacts are as follows:

- WSDL resources for agnostic web services, the Entity and Utility service models
- XML schemas for EBO/EBM, which are widely used in other projects
- The WS-policy files that are used in more than one project/Proxy
- XSLT transformations are related to EBO/EBM, and are especially used in facades for core business processes
- Cross-project XQueries (yes, it's not XML, but anyway)

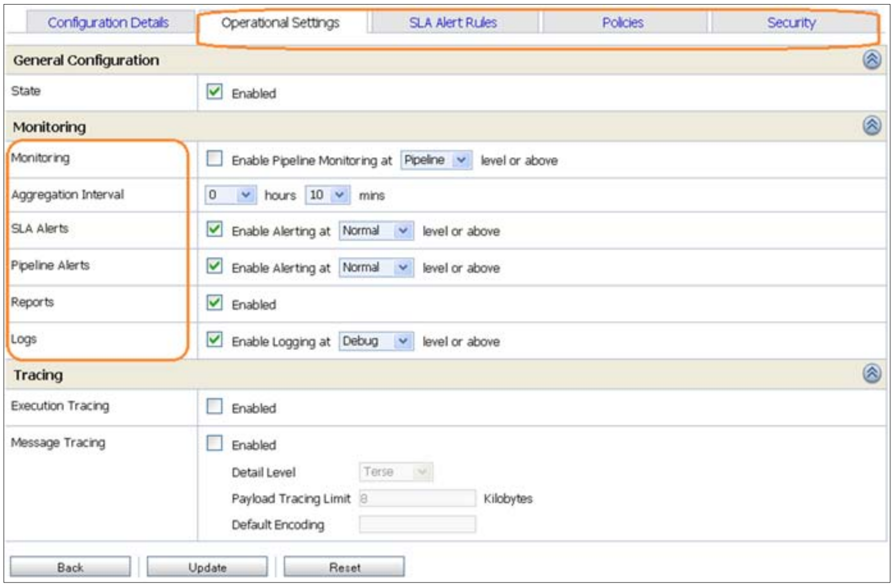
It is better to put them into dedicated common folders that are not related to individual projects. That's what we have done here by placing the entity WSDL in the dedicated folder. Therefore, it is not visible in the preceding screenshot. You will see a more elaborate folder structure in the following example of SB. Why do we put your attention on that? It is not your real artifacts repository yet. Implementing proper resource structuring will save you from lots of grief even for small projects, as resource structuring can be painful. The additional steps are as follows:

   1. On the **Project** page, in **Folders**, enter the folder name in the field provided. Initially, enter `WSDL` and click on **Add Folder** (create a folder that is more suitable for common recourses if you deem it prudent).
   2. Repeat the previous step for Proxy and Business, but now in the concrete project, which you will use for Proxy-based operations.
   3. Activate your changes in WLS **Change Center**.

2. We assume that you have plenty of web services in your service inventory, as we don't want to waste your time in creating some "Hello world":
   1. Create a new OSB session in **Change Center** and select **Project Explorer** in the navigation pane.
   2. Go to the **WSDL** folder you created earlier.
   3. In the **Resources** pane, from the **Select Resource Type** list, select WSDL.
   4. Enter the following information in the **Create a New WSDL Resource** page:
      - Enter the meaningful name for your service as the resource name
      - Browse and select the WSDL associated with your service
      - Click on **Save** to create the WSDL resource and activate the changes
3. Finally, we create the Proxy Service by performing the following steps:
   1. Again, create a new session and go to the proxy subfolder of your project.
   2. In the **Select Resource Type** list, select **Proxy Service**.
   3. Give the Proxy Service a service name according to your preferences, but follow the Canonical Expression SOA pattern.

4. In **Service Type**, select **WSDL Web Service** and then click on **Browse**.
5. The **Select a WSDL** page will appear. The Proxy Service pattern is based on the WSDL resource that you originally created; hence, you must reference the resource here. Select the initially imported WSDL.
6. Select the WSDL port.
7. Submit the changes on this page.
8. Go to the next page and accept the default protocol as HTTP.
9. It is a good time to set the endpoint URI for our Proxy, as from now on all consumers will access your service through a newly created `/order/getCustomerInfo` Proxy.
10. Accept the default for the **Get All Headers** option, or set it according to your operational requirements. This option is especially important for the REST services.
11. After reviewing the Proxy Service configuration settings, click on **Save** to register the service.
12. Do not forget to activate the changes.

We are done. We have created a middle tier between the requestor and the provider, and all abstract patterns visualized in the first figure of this chapter will be squeezed into one purple box behind Proxy, thanks to the **Message Flow** tab that is associated to the Proxy in the OSB realization. Thus, Proxy is our ESB Patterns enabler in Oracle apprehension, of course. Still, it's perfectly aligned with the vendor-neutral SOA patterns catalogue.



General Configuration

State	☑ Enabled

Monitoring

Monitoring	□ Enable Pipeline Monitoring at Pipeline level or above
Aggregation Interval	0 hours 10 mins
SLA Alerts	☑ Enable Alerting at Normal level or above
Pipeline Alerts	☑ Enable Alerting at Normal level or above
Reports	☑ Enabled
Logs	☑ Enable Logging at Debug level or above

Tracing

Execution Tracing	□ Enabled
Message Tracing	□ Enabled Detail Level Terse Payload Tracing Limit 8 Kilobytes Default Encoding

We not only separated the business service (provider) and client, who will now access the resource using `/order/getCustomerInfo`, but we also established a layer of framework. This will allow us to control all the aspects of the service's lifecycle—SLA, policies, security, and monitoring (refer to selected areas in the preceding figure). Now, we can intercept all traffic to/from our service and do all that is necessary. Interestingly, Interceptor is the exact word that describes this behavior in other realizations of the ESB SOA pattern, for instance, ServiceMix/CXF. Naturally, we will not be able to touch all these aspects in one chapter, so please refer to the OSB documentation for each tab in the preceding screenshot.

> Our short exercise was based on the WS/WSDL service. The REST Proxy implementation will be even easier, as we do not need to import any resources. Yes, REST services were designed with simplicity in mind, but think of detachable contract—how much flexibility it gives us! The time of holy wars between SOAP and REST is over, and thanks to OSB, we have unified service management for all types of services.

The Message Broker pattern mentioned previously is equally capable to dispatch messages and could be a good choice for REST or just HTTP-based services. Message Broker, as a commercial product, is commonly associated with IBM (`http://www-01.ibm.com/software/integration/ibm-integration-bus/library/message-broker/`), and we hope that you understand that we are focusing only on the pattern. It will be interesting to see what we gain and what we lose if we try to implement a custom lightweight Message Broker for a predefined canonical protocol.

From Message Broker to Service Broker

Again, we will look at the CTU business case started in the previous chapter. While the Brazilian CIO was busy re-engineering and implementing order fulfillment in a Pan-American way, urgent needs for a lightweight service broker for mobile OSS/BSS operations were expressed in the Chile regional office.

A new project named **Extended Data Interchange** (**XDI**) was started independently. Initial requirements were to cover message brokering between ERP (Oracle EBS R11) and mobile/network equipment providers, routing purchase orders to different suppliers. Direct communication was impossible for the following two reasons:

- OeBS R11 was able to post messages as SOAP 1.1, where most of the suppliers required SOAP 1.2.

- Supplier endpoint maintenance was considered impractical in the core ERP and the need for a middleware layer was expressed. The solution had to be compact enough to be moved out from the production to the communication zone or even to the DMZ. Basic security features were implemented in the new broker.

From the very beginning, this solution was considered as temporary, as more mature enterprise products were expected. However, nothing is more permanent than temporary. Migration from R11 to R12 with the SOAP 1.2 support has been considerably delayed; full-fledged ESBs were prohibited by the headquarter until completion of the pilot SOA project in Brazil. We are sure that you must be familiar with such situations.

What have we got as an architect here?

- We have local developers familiar with JEE.
- We managed to negotiate the Canonical Protocol for all parties as HTTP. That greatly simplifies our life as we will deal with only one synchronous protocol, with simple operations and no SOAP conversions. If SOAP 1.2 is required, we will wrap our message in the required envelope.
- We realize that this broker will be temporary anyway, but all EBO/EBM structures, transformations, routing slips are related to the core business, and will stay. Therefore, the solution must be highly modular, ensuring easy migration or coexistence with solutions to come.
- Obviously, MB must be a very reliable and a good performer.

Naturally, new requirements emerged quite soon. Now, we have to transport more business messages (not only the purchase order as it was planned initially). Also, we have more application to integrate in addition to a single instance of OEBS. Still, all of this doesn't sound that bad. We can easily deduce that we have the classic RDT interchange pattern with transformation, possibly with translation and content-based routing. Synchronous APIs do not require complex processing; all brokering must be performed from a central location. The broker will be some kind of single point of contact, acting in the same manner as a Front Controller. The Front Controller is one of the common JEE patterns, although it is commonly related to UI and presentation-tier integration (`http://www.oracle.com/technetwork/java/frontcontroller-135648.html`). Physical realization of a Front Controller is our main interest as it's a classic servlet, and after consultations with developers we agreed that this will be the heart of our Message Broker.

Talking about Message Broker's body parts, the next vital organ will be Business Delegate, which can be accessed through the servlet's helper. As we discussed in the previous chapter, it has the responsibility to transform and dispatch messages to the actual workers (service providers). This will be the solution's brain. The last D (deliver) from RTD will be closely associated with Business Delegate, which presents the Adapter Factory pattern. Despite the initial intention to serve only HTTP communications, file/FTP operations requirements were added into the design.

A simplified Message Broker implementation

Now, we will walk through the RTD parts focusing on pattern implementation in a simple hub-and-spoke solution.

Receive

The receiving part must contain functions for message parsing, message identification, and the extraction of an execution plan based on message ID. These are typical parts of any servlet, except probably the execution plan, which we add for dynamic task invocation. A servlet's lifecycle starts with initialization, which occurs only once when the servlet is started, as presented in the following code:

```
String dispatchSufix = "";
String mappingURL = null;
...
public void init(ServletConfig config) throws ServletException {
        super.init(config);
        ...
        tmp = config.getInitParameter(MAPPING_URL);
        if ( tmp != null ) mappingURL = tmp;

        tmp = config.getInitParameter(OK_PAGE);
        if ( tmp != null ) okPage = tmp;
        ...
        tmp = config.getInitParameter(XMLRULESPROVIDER_URL);
        if ( tmp != null ) XDIRuleListWSURL = tmp;
```

This fact gives us the possibility to put all heavy, but one-time, operations into the initialization routine, for instance, obtaining and caching the data sources. For simplicity, we will not use the database directly in this example, and in general the performance is not our primary concern. Initially, we just need some pages to display the positive and negative responses, resource mapping, default logging level, and type of rule engine we will use for the process type resolution (name of routing slip), and default transformation. All these parameters will be registered in the `web.xml` deployment descriptor according to the servlet specification and reassigned during the initiation step as shown in the preceding code.

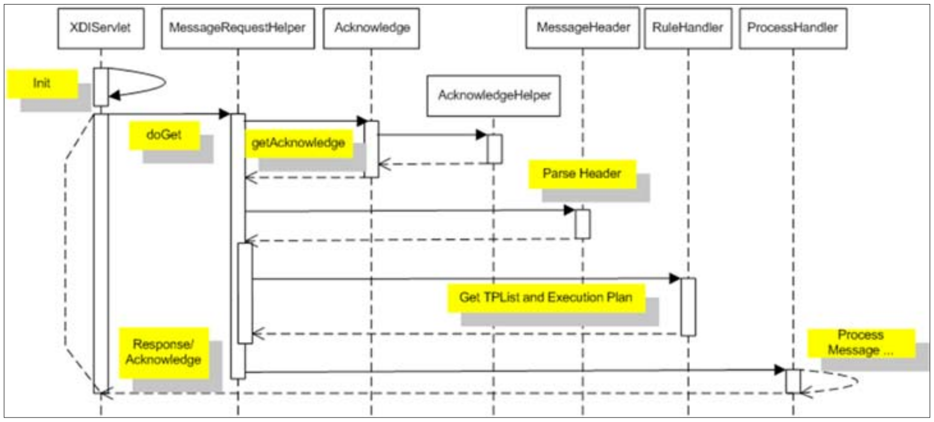
We will simplify two default servlet operations, `doGet` and `doPost`, by calling the same `processRequest` procedure that will be our entry point to the processing logic:

```
public void doGet(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException  {
         processRequest(request, response);
  }
public void doPost(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
    doGet(request, response);
  }
```

First, we will make an instance of a request helper, a log handler, and two supporting objects, namely `MessageHeader` and `Acknowledge`:

```
protected void processRequest(HttpServletRequest request,
  HttpServletResponse response)
throws ServletException, IOException    {
  ...
    MessageRequestHelper helper =
      getMessageRequestHelper(request, response);
    LogHandler logger = new LogHandler();
   ...
   MessageHeader mheader = new MessageHeader();
   Acknowledge ack = new Acknowledge();
```

The `MessageHeader` object will contain all message-context elements that we will extract from the inbound message. It is ideal that we will have a unified standard header (which is not always true) within a single organization (CTU), but external partners/mobile equipment suppliers use their own standards, which are not always compatible with SDBH. Thus, we will have to modify the message header after identifying the trading partner(s). Have a look at the following figure:



This modification can be done on the **MessageHeader** Java object if the receiver is one, or after combining the message header and message body together in an individual transformation step within an execution plan if we have multiple TP receivers.

We can say the same about the `Acknowledge` object. Initially, the sender was only one, Oracle e-Business Suite, and the `Acknowledge` was predefined for this application. However, as it was described in the business case, quite soon, other internal applications discovered the benefits of this Message Broker and the transformation of the `Acknowledge` message becomes necessary. Luckily, standardization of the `Acknowledge` message is easier than Message Header, as all applications are internal.

You can understand the structure of the `Acknowledge` object from the overridden `toString()` function, which we recommend that you should have for every entity class. The following code is for object structure only, as you can be better off with Apache commons for the implementation of the `toString()` function, by using `ToStringBuilder.reflectionToString(this)` for instance:

```
/*
    * @webmethod
    */
    public String toString(){
        return getClass().getName()
        +" ackReceiptID:"+ackReceiptID
        +" ackDateReceived:"+ackDateReceived
        +" ackDateProcessed:"+ackDateProcessed
        +" ackTradingPartnerID:"+ackTradingPartnerID
        +" ackFileName:"+ackFileName
        +" ackReferenceNumber:"+ackReferenceNumber
        +" ackMessageID:"+ackMessageId
        +" ackMessageStatusCode:"+ackMessageStatusCode
        +" ackEventCode:"+ackEventCode
        +" ackEventDescription:"+ackEventDescription
        +" ackEventSource:"+ackEventSource
        +" ackAction:"+ackAction;
    }
```

When we talk about Message Header, we assume that it will be based (if not compatible) on the SBDH standard. For `Acknowledge`, there are no publicly accepted standards, but the structure of an existing, common Logging service (and its storage) could be a good start. In this design, a traditional `log4j` library was used for technical errors and a `log DB` structure was standardized long before the creation of this Message Broker. In addition to this, the function-related `LogHandler` is responsible for choosing an appropriate way to register business events (type of information, warning, and error).

Naturally, WS realization will be the most atomic and modular, therefore `LogHandler` has the capability to select the logging procedure. However, there were some concerns regarding possible performance deprivation. Logging level and realization are the parameters settled during servlet's initialization, and the default values are in the `web.xml` descriptor. We must remember that parameters acquired during the `init` phase will stay active during the servlet's life. Therefore, the logging level of the process must be configurable from the execution plan's header:

```
// Implementation of the task logger in LogHandler
public void log(String ipMsgtype, Task currtask) throws java.
io.IOException  {
        try{
            log(ipMsgtype,
                getLogEventDescription(currtask), currtask.getMsgId(),
currtask.getSenderTPId(),
                currtask.getTaskEngine(), currtask.getXDIInstanceID(),
currtask.getEdiProcessReportLevel()
           );
       }
       catch(Exception ex)  {  ex.printStackTrace(); }
    }

//Implementation of the basic logger in LogHandler
    public void log(String ipMsgtype, String ipMsgtext, String
      ipMsgcode, String ipUsermsg, String ipMsgsrc,
      String ipMsgjobid, String ipLoglvl) throws
        java.io.IOException
    {
        EventLogHandler logwriter = new EventLogHandler();
        logwriter.log(ipMsgtype, ipMsgtext, ipMsgcode, ipUsermsg,
ipMsgsrc, ipMsgjobid, ipLoglvl);
    }
....
//Typical use
  logger.log( "INFO", "Rule engine in use : " + RuleEngineType ,
    mheader.getXDIMsgId(), mheader.getXDIUser(),
    "XDI.RuleEngine", mheader.getXDIJobRefId(), "3" );
```

Although the realization of the Logger WS is pretty straightforward, it also has two distinctive paths: either you write data directly into the DB or enqueue the message into Advanced Queue. Anyway, it is a one-way service and must not disrupt message processing. It is important to mention that `Log4j` (or a newer `SLF4J` library if you choose to use it) also has its own configuration for level and layout, which we set in a LogHandler.

Now, after the helper's initiation, our first task is to recognize an inbound message. There are two possible strategies here, based on the XML parsing: `DOM` and `SAX`. Obviously, each strategy has its own pros and cons, but if we require really high performance, we should rely on `MsgID` recognition by the SAX parser. We instantiate the SAX `cmdHelper` in the servlet's `processRequest` and execute the `SAX` parsing function.

The `SAX` parsing sequence is presented by the following code snippet:

```
//In XDIServlet, processRequest
MessageCommand cmdHelper = helper.getCommand();
    XdiMsgID = cmdHelper.execute(helper);
    logger.log( "INFO", "New message received. Message ID: "+
        XdiMsgID", "N/A", "N/A", XDIServlet.servletInstance + "
        Front Controller", "N/A", "3");

//In MessageRequestHelper
.....
    private static final String elementName = "XDIMSG_ID";
....
    public MessageCommand getCommand(){
        java.util.ArrayList elementList = new
            java.util.ArrayList();
        elementList.add(elementName);
        return new MessageGetElementCommand(elementList);
    }

// In MessageGetElementCommand
public String execute(MessageRequestHelper helper) throws javax.
servlet.ServletException, java.io.IOException {
    String elementContent = null;
    //
    // Start reading content
    //
    try{
        java.io.Reader bodyReader = helper.getReader();
        elementContent = getElementContent(bodyReader);
        bodyReader.close();
    }
    catch ( javax.xml.parsers.ParserConfigurationException e){
        throw new javax.servlet.ServletException(e);
    }
```

```
        catch ( org.xml.sax.SAXException e){
            throw new javax.servlet.ServletException(e);
        }
        return elementContent;
        }
    ......
    // where getElementContent is SAX Parser implementation

    private String getElementContent(java.io.InputStream inputStream)
    throws
        javax.xml.parsers.ParserConfigurationException,
        org.xml.sax.SAXException,
        java.io.IOException {
            String elementContent = null;
            inElement = false;
            SAXParserFactory factory = SAXParserFactory.newInstance();
            factory.setNamespaceAware(true);
            SAXParser parser = factory.newSAXParser();

            parser.parse(inputStream,this);
            if ( stringBuffer.length() > 0 ) elementContent =
                stringBuffer.toString();
            return elementContent;
        }
```

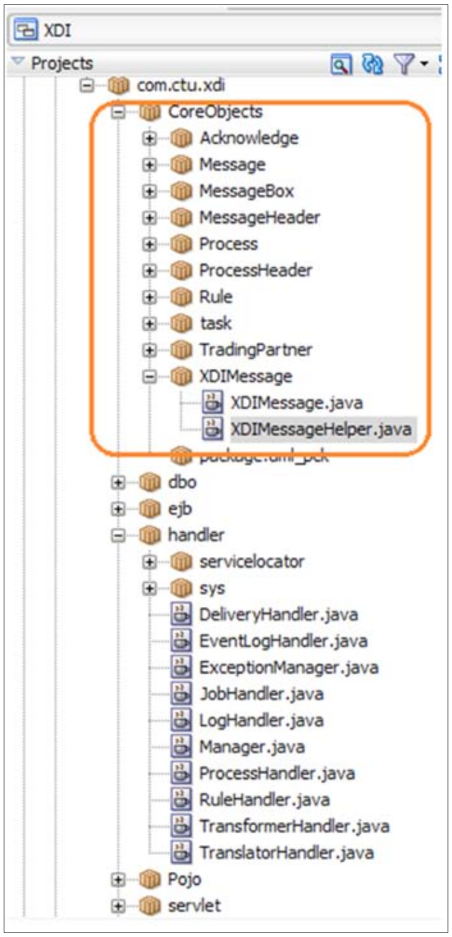
This SAX parsing routine is pretty standard for all the XML servlets as it's the first step to map message type/content for further actions. For simple routing actions, this implementation will be sufficient. As a matter of fact, if we want to have a full-fledged Message Broker with content-based routing, an implementation of the DOM parsing is inevitable, as shown in the following code:

```
    //In XDIServlet,   processRequest
    //new: DOM parser.
    //Get message DOM using Oracle XDK
        XMLDocument msgXdiDOM = helper.getMessageDOM(ack);

    //In MessageRequestHelper
        public XMLDocument getMessageDOM(Acknowledge ack) throws
            ServletException, IOException  {
        String bodyroot = null;
        //declaration for different DOM specs
        XMLDocument msgXdiDOM = null;
        // Document msgXdiDOM;
        ...
        XDIMessageHelper msghelper = new XDIMessageHelper();
        try
        {
            //Oracle parser
            msgXdiDOM = msghelper.getmsgXdiasDOM(getReader());
        }
        catch(Exception ex) {
            logger.log( "ERR", "Unable to parse incoming message",
                "N/A", "N/A", XDIServlet.servletInstance + " Front
                Controller", "N/A",  "3" );
            ack.setackMessageStatusCode(ack.STATUS_CRITICAL_ERROR);
            ack.setackEventCode(ack.STATUSCODE_CRITICAL_ERROR);
            ack.setackEventDescription(ex.toString());
            ex.printStackTrace();
          }
            return msgXdiDOM;
        }

    ......
    // In XDIMessageHelper. Actual parsing
    //Technically you can use any DOM parser. We use classic oracle.xml.
    parser.v2.* It's also configurable through Servlets initiation
        public XMLDocument getmsgXdiasDOM(java.io.Reader reader)
            throws IOException, SAXParseException, SAXException
        {
            XMLDocument msgXdiDOM = OraXMLHelper.parse(reader, null);
            return msgXdiDOM;
        }
```

As you have noticed, we are populating the `Acknowledge` object (passing the `ack` parameter) every time when it's necessary, and definitely in case of errors. We will do exactly the same with `MessageHeader`, right after obtaining the `DOM` message:

```
    // set Message header values
        helper.setMessageHeader(msgXdiDOM, mheader, ack);
```

Why do we need to do this? For the same reason why the SAX parser with one element (MsgID) recognition is not enough. For the guaranteed identification of the business process, we will need SenderID and EventName at least (and some more, but we will skip the details for brevity):



```
XDI
Projects
    com.ctu.xdi
        CoreObjects
            Acknowledge
            Message
            MessageBox
            MessageHeader
            Process
            ProcessHeader
            Rule
            task
            TradingPartner
            XDIMessage
                XDIMessage.java
                XDIMessageHelper.java
            package.uml_pck
        dbo
        ejb
        handler
            servicelocator
            sys
            DeliveryHandler.java
            EventLogHandler.java
            ExceptionManager.java
            JobHandler.java
            LogHandler.java
            Manager.java
            ProcessHandler.java
            RuleHandler.java
            TransformerHandler.java
            TranslatorHandler.java
        Pojo
        servlet
```

Our MessageHeader elements will be used in the next step for the recognition of business process and extraction of the execution plan for this process. It will be used further for the construction of the XML Message Header in the delivery phase. Here, we will discuss how we are going to implement marshalling/unmarshalling for the core objects that we have in our broker. You can see a list of objects in the preceding screenshot.

Naturally, not all of them have to be converted into XML and back, but **Process**, **ProcessHeader**, **Acknowledge**, and **MessageHeader** are the primary candidates. For instance, in *Chapter 2*, *An Introduction to Oracle Fusion – a Solid Foundation for Service Inventory*, dedicated to modern SOA technology, we mentioned several common O/X mappers such as JAXB and JiBX frameworks. The key factors that are naturally defining our choice of marshaller are performance and ease of configuration. Spring O/X can be a very good alternative as we do not need to construct the JAXB context, JiBX binding factories, and so on. We have other options such as using Castor XML mappings, XMLBeans marshallers, and XStreams. The choice is yours, but you can also implement the conversion of an object to XML without any libraries. Here, our task is really simple. Therefore, good, simple Java constructs will work quite well and surprisingly fast. Every object has its own helper, where we have a primitive section for XML construction. First, we get an instance of the element writer utility, based on `System.out.println()`, as shown in the following code:

```
ElementWriter ewriter = new ElementWriter();
    java.io.StringWriter xdiDocWriter = new
        java.io.StringWriter();
```

We will then write our elements as shown in the following code:

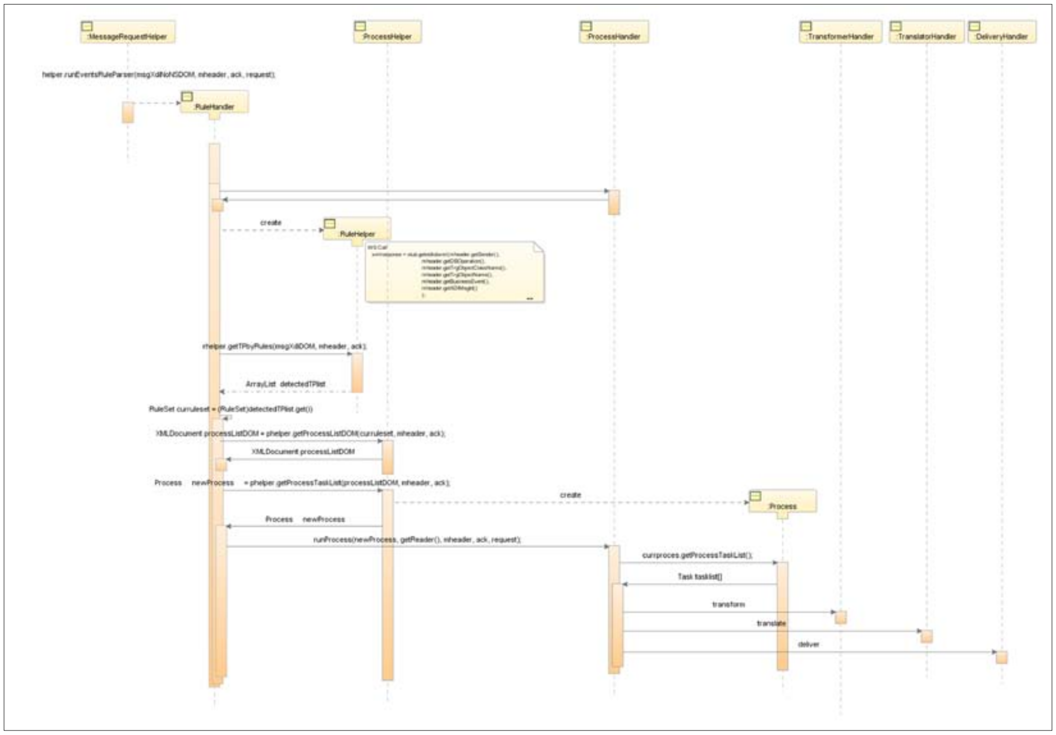
```
xdiDocWriter.write("<mhs:MessageHeader>");
ewriter.element(xdiDocWriter, "mhs:XDIMsgId", mhs.getXDIMsgId());
ewriter.element(xdiDocWriter, "mhs:BusinessEvent",
    mhs.getBusinessEvent());
ewriter.element(xdiDocWriter, "mhs:Sender", mhs.getSender());
```

For more complex objects and messages, we will advise you to use the Spring framework, or any other that suits you.

Now, we have all the necessary functionalities to extract an execution plan. Similar to the realization discussed in the previous chapter, the execution plans are the XML objects stored as a file object, and they will be extracted from XML mapping file, linking the sender and the message IDs with process. The name and location of this file is configured using the `web.xml` deployment descriptor and extracted during the servlet's `init` phase. The FileIO realization of this lookup makes this broker extremely lightweight and suitable for autonomous installation on DMZ or in an integration zone without connecting to any database. If your requirements are not that strict, you can implement `ExecutionPlanLookupService` as a WS (using SCA from the previous chapter) and invoke it using `MessageHeader` as an input parameter. This was not the case when we decided to build this broker. After the extraction of the execution plan XML, we are ready to process it.

Transform

Transformation is not the only task we will perform; therefore, this name is a bit misleading. In general, we can invoke any EJB or another HTTP endpoint registered in the execution plan. Initially, we agreed that the scope of our tasks will be transformation, translation, and delivery (as FileIO or HTTP post), which are presented as individual helpers (dispatchers) that are controlled by the `ProcessHandler` factory. Please refer to the following figure:


:MessageRequestHelper
:ProcessHelper
:ProcessHandler
:TransformerHandler
:TranslatorHandler
:DeliveryHandler
helper.runEventsRuleParser(msgXdiNoNSDOM, mheader, ack, request);
:RuleHandler
create
:RuleHelper
rhelper.getTPbyRules(msgXdiDOM, mheader, ack);
ArrayList detectedTPlist
RuleSet curruleset = (RuleSet)detectedTPlist.get(i)
XMLDocument processListDOM = phelper.getProcessListDOM(curruleset, mheader, ack);
XMLDocument processListDOM
Process newProcess = phelper.getProcessTaskList(processListDOM, mheader, ack);
create
:Process
Process newProcess
runProcess(newProcess, getReader(), mheader, ack, request);
currproces.getProcessTaskList();
Task tasklist[]
transform
translate
deliver

VETRO sequence on custom Service Broker

It's a classic VETRO pattern, where validation (for V) is initially done by the receiver (servlet) when we parse it into DOM and implicitly done during the individual transformation (enrichment) steps. A message incompliant to the declared XSD will result in failure in transformation.

Technically, we are looping though the task list nodes and invoking a related helper to execute the task as follows:

```
TaskHelper tskhelper = new TaskHelper();
TransformerHandler transformhandler = new
    TransformerHandler();
    try {
        setBody(reader, true);
        XMLDocument taskListDOM =
            tskhelper.getTaskListDOM(mheader,ack);
        // new call for task ArrayList
        ArrayList  tasklist =
            tskhelper.getProcessTaskList(taskListDOM,
            mheader,ack);
...
         for (int i = 0; i < tasklist.size(); i++) {
             Task currtask = (Task)tasklist.get(i);
..
             if(currtask.getTaskAction().equals("Transform")){
                 log.info( "Executing:" +
                     currtask.getTaskAction());
                 msgbodyReader =
                     transformhandler.transformdispatcher
                     (getReader(), currtask, request);
             if(currtask.getTaskAction().equals("Deliver")){
                 log.info( "Executing:" +
                     currtask.getTaskAction());
                 DeliveryHandler deliverer = new
                     DeliveryHandler();
                 deliverer.deliver(getReader(), currtask, request,
                     ack, mheader);
```

The transformation engine type is defined as a parameter for the transformation task, and the dispatcher will send it to an appropriate engine where transformation is finally done. This is demonstrated in the following code:

```
try {
    stylesheetfile = task.getStylesheetLocation();
    stylesheet = new XSLTInputSource(stylesheetfile);
    xmlsource = new XSLTInputSource(reader);
}
catch (Exception e) { ... }
try {
...
    XSLTResultTarget xmlresult = new XSLTResultTarget(out);
    XSLTProcessor transformer =
        XSLTProcessorFactory.getProcessor();
    transformer.process(xmlsource, stylesheet, xmlresult );
    ...
    java.io.Reader msgbodyReader  = new
        java.io.StringReader(xmlresult.toString());
    return  msgbodyReader;
try {
    stylesheetfile = task.getStylesheetLocation();
    stylesheet = new XSLTInputSource(stylesheetfile);
    xmlsource = new XSLTInputSource(reader);
}
catch (Exception e) { ... }
try {
...
    XSLTResultTarget xmlresult = new XSLTResultTarget(out);
    XSLTProcessor transformer =
        XSLTProcessorFactory.getProcessor();
    transformer.process(xmlsource, stylesheet, xmlresult );
    ...
    java.io.Reader msgbodyReader = new
        java.io.StringReader(xmlresult.toString());
    return msgbodyReader;
```

Deliver

The last broker's responsibility is to deliver the message to the ultimate recipient(s):

```
//Dispatcher uses the task engine to dispatch to the certain task
...
if(task.getTaskEngine().equals("XDIMB.apache.httpcomponents.httpcl
    ient")) {
        log.info("Dispatching as HTTP, TaskCommType: "+
            task.getTaskCommType() + "; Engine: "+
            task.getTaskEngine());
            httpdeliverer.deliverCommonHTTP(outbodyReader, task,
                request,  ack, mheader);
    }
...
//here is the standard Apache HTTP Component library
....
    tskurl = task.getReceiverEndpoint();
    userName = task.getReceiverEndpointUserName();
    port = task.getReceiverEndpointPort();
    tskhost = task.getReceiverEndpointHost();
    ...
    HttpParams params = new SyncBasicHttpParams();
    HttpProcessor httpproc = new ImmutableHttpProcessor(new
        HttpRequestInterceptor[] {...}

    ...
```

```
        HttpRequestExecutor httpexecutor = new HttpRequestExecutor();
        HttpContext context = new BasicHttpContext(null);
        HttpHost host = new HttpHost (tskhost, port );
        DefaultHttpClientConnection conn = new
            DefaultHttpClientConnection();
        ConnectionReuseStrategy connStrategy = new
            DefaultConnectionReuseStrategy();
        ...
        BasicHttpEntityEnclosingRequest request = new
            BasicHttpEntityEnclosingRequest("POST", tskurl);
        request.setEntity(requestBodies[i]);
        ...
        request.setParams(params);
        httpexecutor.preProcess(request, httpproc, context);
        HttpResponse response = httpexecutor.execute(request, conn,
            context);
        response.setParams(params);
        httpexecutor.postProcess(response, httpproc, context);
    ....

    //If you want to dispatch to an other HTTP poster, add engine type to
    Execution Plan, new IF branch to dispatcher and new Java  deliverer
```

The pros and cons of a simplified Message Broker

The solution based on the presented architecture has been delivered quickly and served its purposes really well for a limited number of trading partners (message recipients). Most importantly, performance was more than acceptable and it was really reliable, so the tactical goals were achieved. We can even see this servlet-based approach as a good investment in the REST service infrastructure. Commonly, REST is implemented by Jersey-servlets, and we encourage you to look at this technology as it's not in the scope of this book. You will find a lot of similarities with the quick example we discussed previously. Oracle has many good examples that cover the servlet pattern and its utilization in JAX-RS/Jersey (`http://docs.oracle.com/cd/E19776-01/820-4867/ghqxq/index.html`). This means that you really do not have to do everything from ground zero for message brokering and service implementation. The actual purpose of this example was to demonstrate the physical implementation of some patterns such as Mediation and Adapter Factory, as discussed before, and mainly the EAI-SOA path of evolution: **point2point | hub-and-spoke | message broker | service broker | full-fledge ESB**.

The example also demonstrated the complexity of the task. We didn't cover a lot of features that are compulsory for a full-scale solution, for instance:

- Basic security
- MTOM / messages with attachments
- Throttling / Load balancing

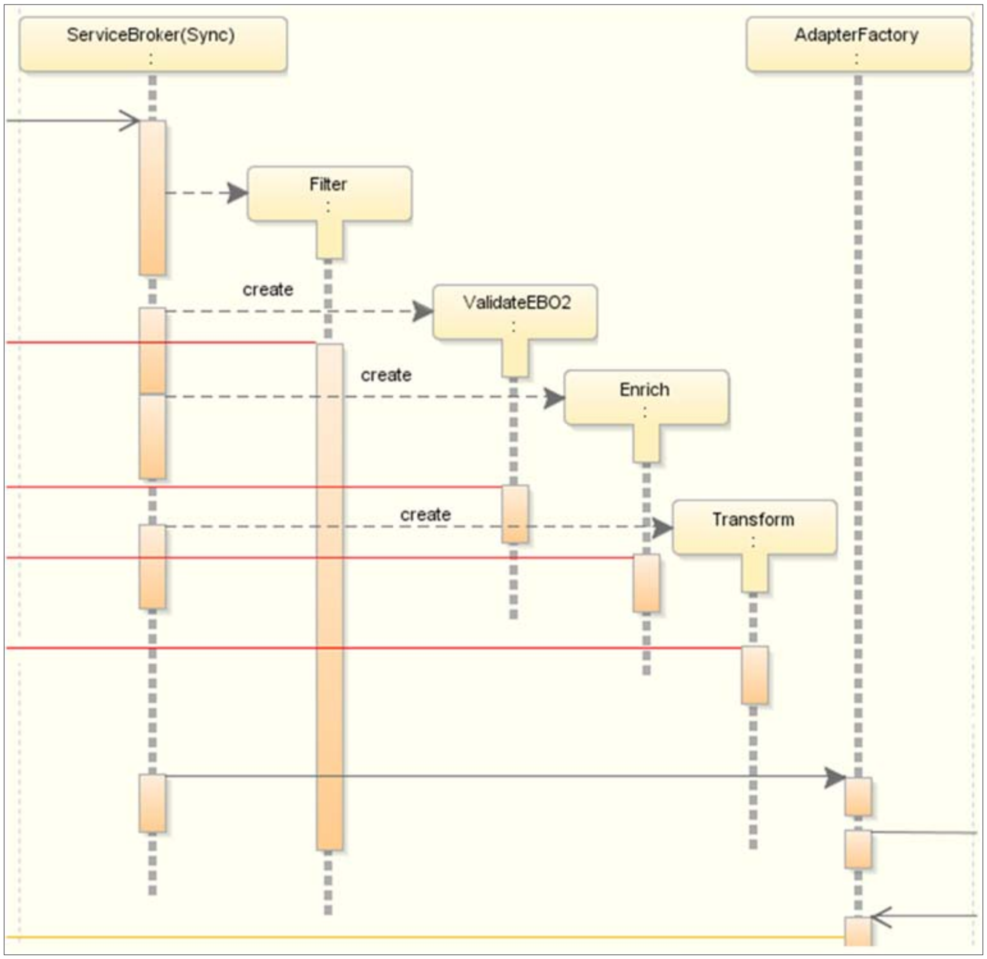
Also, many more solutions will be covered later. However, most importantly, we didn't implement true service decoupling, as no Proxy concepts were provided. It is a good time to return to the CTU service broker now and see how we can improve this solution using the discussed and tested patterns.

Oracle Enterprise Business Service's SOA patterns

After completing the previous task, we have a Message Broker that is capable of implementing the RTD interchange pattern in the form of a hub-and-poke controller. Although it is perfectly operational, you have to add a little to the demonstrated code snippets to create a production version. Its practical application is limited by assumptions we made at the beginning of this exercise. Let's repeat the assumptions again:

- Limited number of protocols.
- Limited number of message validation techniques.
- Limited ways of message transformation (XSLT is preferable).
- Relatively elaborate ways of implementing new delivery options and any pluggable modules in general.
- When it comes to policy-based management, you will have to manage everything on your own. You will have to implement policy enforcement points (PEP) on your own too.
- To make the situation more dramatic, as you remember, we even implemented a custom rule engine (although this argument is weak as nothing prevents us from using any RE with Java or WS API; almost nothing, as performance should be considered seriously).

Simply put, this is a servlet with some customization and with all downsides related to a custom solution. Apparently, CTU cannot accept it as a generic Pan-Latin-American Service Broker and Adapter Factory. Equipped with knowledge gained during this exercise and understanding what we need from the SOA patterns catalogue to mitigate the discovered problems, we are now ready to refactor our SCA solution from the previous chapter and move the synchronous parts to the OSB. There will be more protocols to adapt, more formats to translate, more models to transform, more tasks to invoke, and more operations to execute. Refer to the following figure:



The preceding figure demonstrates a sequence diagram for the refactored part of the asynchronous Service Broker. As we have mentioned before, following the AIA methodology, the EBS layer will be presented by northbound and southbound parts (as sequence diagrams go from left to right, this geographic notation is naturally shifted anticlockwise). Service Bus, as a natural part of EBS, should be universal enough to execute VETRO patterns regardless of the location. For a better understanding of the challenges, we have summarized SB's functional capabilities for each area in a table that follows in the next section. We need it for a better understanding of the feasibility of implementation of an all-in-one Composition Controller by using the OSB.

Detailed analysis – functional decomposition

The Synchronous Composition Controller can receive service consumer calls from two directions—from Composition Initiator (Application Sender), optionally through the Adapter framework, or from the EBF/SCA master Composition Controller. In the case of a synchronous interchange pattern, we could perform all operations in OSB without involving the SCA business flows.

We really *should* do that, because even synchronous logic can be quite complex. Do not let your enterprise service collaboration layer (or EBS framework) turn into a full-scale orchestration! We have started with a demonstration of the Proxy capability of ESB, and we would like to repeat that decoupling again. It is probably the most important role of ESB and we would perform transformation, validation, and protocol bridging (which we can relatively easily implement on the Adapter level, which is not always ESB) only after that. Synchronous Composition Controller should operate only with simple and linear routing slips (execution plans), with a number of steps not more than 10 (this is quite an empirical figure). Of course, the limit for the number of invocations should be set according to your operational environment / computing power.

> Let's do the simplest math. You certainly have three fully equipped network zones, namely DB, APP, and integration, in your technical infrastructure. Ping integration from the DB. Most probably, you will get no less than 2 ms. Even the simplest pass-through Proxy will have the same ping interpretation period of 2 ms. We can expect double or more from utility services implementing VETRO patterns in OSB. With full logging, 10 consecutive invocations can have up to 300 ms of operational time in OSB alone. Be careful about promising something faster than that.

In the case of a linear sequential synchronous process, inbound and outbound functional parts of OSB will be reduced to the same technical layer that acts as a Service Broker with intermediate routing and asynchronous queuing, if necessary. Here, the main difference with the Message Broker will be the complete reliance on the Proxy Service pattern implementation for true decoupling.

Now, we will summarize the challenges of ESB's functional decomposition.

Task	Northbound	Southbound
Receive	We must bear in mind that a message could come in the form of a container with a Message Header object already in place. Although it simplifies the message recognition, we should be careful when putting this message into a universal container for Agnostic Composition Controller. Service Facade should be implemented to remove nested headers.	In general, a southbound controller will never act as an ABO message receiver. From the SCA business flow, we will receive EBM with all elements in place, but an execution plan will most probably contain only one task, as SCA will implement its own master Composition Controller.
Message identification	If Message Header is in place, the universal `MsgID` will be used for identification. Otherwise, we can use the name of the message root element. In this case, the trading partner agreement will guarantee that every ABO will have its own unique root element. If none of that is possible, the Adapter framework must be employed for message header construction. **Application Business Connector Service** (**ABCS**) will be entirely responsible for conversion of ABO into EBM.	Only the `MessageHeader` element `MsgID` is needed.

Task	Northbound	Southbound
Business Process Recognition	This is the next logical step after message identification. It can be done by `MsgID`, `senderID`, and events name; all of them are the parts of **Standard Business Document Header** (**SBDH**). The result of this operation could be the extraction of an execution plan, but that will be necessary if EP is not provided in the message container by the ABCS framework. Alternatively, we can delegate the extraction of an execution plan to the asynchronous Service Broker in SCA. However, if we are really looking for good performance (and reliability too) of synchronous services, we should avoid calling the SCA level excessively. `MessageHeader` elements will clearly indicate the process **Message Exchange Pattern** (**MEP**).	In this part of the framework, we only consume an execution plan provided in the previous steps.
Validate	Without ABCS, this function shall be performed on the northbound ESB part. This functionality is standard in the Message Processing palette.	We should perform message validation on a response action pipe.
Filter	Filtering is based on identification of a certain XPath in the message body and has a purpose to suppress unwanted requests. This must be configurable in the Request Action pipe.	Similar to northbound, the edge should be configurable for the response action pipe.
Enrich	This is not applicable for northbound. Enrichment is usually done on the adapter level. Inbound messages could be transformed, though.	Enrichment means service callout, which requests additional data. Thus, its regular invocation of dynamic endpoint, when the endpoint URL is extracted from the EP in Agnostic Composition Controller, shall be configurable with a bypass option.

Task	**Northbound**	**Southbound**
Transformation	This operation is not really common on the northbound side, but could be possible if the URL to XSLT or XQuery resources will be provided in the inbound EP.	The most common task on the southbound side of SB. The URL to XSLT or XQuery resources shall be provided in an inbound EP. For better isolation, it could be realized by the implementation of the transformation Proxy where we will perform dynamic transformation.
Deliver message	This passes well-formed messages to the SCA or southbound SB part.	The realization of Adapter Factory was discussed earlier. It will dynamically dispatch a message to a concrete protocol Adapter Proxy, similar to what we had in Message Broker.
Adapters	At this moment, we assume that all our OSB operations will be based on SOAP. The technology of creating a JCA adapter in Oracle 11*g* is practically the same as in the previous 10*g* release. Traditionally, we have to create an empty SCA in JDeveloper, drag the required adapter (DB, File, FTP, and so on), and set all the necessary properties related to the adapter's type (pay special attention to JNDI name). We will use the previously created `.jca`, `.wsdl`, and `.xsd` files in Eclipse to generate the OSB services, representing JCA transport. Thus, we will call these adapters *transport adapters*. They will be actively used by both Northbound and Southbound parts of our agnostic controller. However, we assume that the Southbound layer will employ them directly, when Northbound will require a generic reader to dispatch to the different kind of controllers. It is also important to bear in mind that the JCA-compliant resources are hosted not on OSB, but one level down—on WLS as a public resource. On OSB, we create only Proxy/Business services that interact with a supported adapter. That's the beauty of decoupling! We will discuss the OSB Adapter framework in more detail in the following chapters.	

Short summary

Let's summarize what we learned. The preceding table does not reveal any contradicting functionality in the southbound and northbound OSB realization of the synchronous composition controller; therefore, we can build unified versions that are suitable to handle all In and Out service collaboration scenarios. We will just have to add conditional branches in our inbound and outbound pipelines to enable/disable certain operations, depending on the process context that is provided by a recognized execution plan. Let's not forget that not every service collaboration scenario should be handled by an Agnostic Composition Controller, only those that really require implementation of the SOA ESB patterns in a business agnostic way (and dynamic invocation is a prerequisite). Please refer to the business requirements formalized in the previous chapter.

Apart from the Proxy-based service governance patterns discussed previously, we are constantly referring to the service messaging and transformation patterns in the SOA patterns catalog that is assembled under the ESB compound pattern as three compulsory patterns: Asynchronous Queuing, Intermediate Routing, and Service Broker suitable for Agnostic Composition Controller scenarios will fall into these patterns, but we need more detailed descriptions for each of them, linking the generic SOA patterns with the VETRO tasks identified in the preceding table. Really, Intermediate Routing sounds too generic.

To understand the nuances, we are going to link SOA patterns with the underlying EA integration patterns, and then we are going to see the operations we will have to implement on OSB to make them generic.

Problem	Components required	EAI patterns	SOA patterns
Intermediate routing			
We need to redirect messages to the different service providers depending on the message content.	Static routing table, Business Delegate	Content-based routing	Intermediate Routing
This is similar to the previous task, but we need to establish a high level of maintainability and business agility. Therefore, we need to avoid dependency on the router on all possible (current and future) destinations.	Rule storage, rule engine, Business Delegate	Dynamic Router	Intermediate Routing plus rule centralization; Inventory Endpoint;

Problem	**Components required**	**EAI patterns**	**SOA patterns**
We want to process messages with different parts, each of which have different processing requirements; that is, different service providers.	Transformation engine (Java Bean, POJO), message intermediate storage	Splitter	Intermediate Routing plus Composition Controller
After processing the different parts of the message on different endpoints, we want to assemble the results into one message.	Transformation engine (Java Bean, POJO), message intermediate storage	Aggregator	Intermediate Routing plus Composition Controller, and Service Instance Routing
We want to process a message sequentially, passing it through a series of processing steps.	Execution plan (XML), Business Delegate	Routing slip	Intermediate Routing plus state messaging, messaging metadata, service agent
This is similar to routers, but we want to dispatch messages to several recipients *n* out of total list *m*.	Static routing table with parallel rules Business Delegate	Recipient list	Intermediate Routing
We want to suppress the unwanted message.	XPath processor (XMLBean)	Message filter	Messaging metadata
That's the ultimate processing block, presenting the RTD processing pattern in a compound way. We split messages upon receiving, route it to the destinations, and aggregate the responses. The pattern can include routing slip or recipient list as well instead of a basic router.	All components from the preceding column	Composed Message Processor	Composition controller and subcontroller; Service Broker plus Intermediate Routing

Problem	Components required	EAI patterns	SOA patterns
Transformation			
We need to enhance the messages data by the elements not available on the receiver.	Decorator: Callout to external service and transformation engine	Content Enricher	Data Model Transformation plus Composition subcontroller
We are not suppressing a message. We are filtering out unwanted elements.	Transformation engine (Java Bean, POJO)	Content Filter	Data Model Transformation
The same information we want to receive in different formats from the same service. For instance, the same service should provide the same content in JSON and XML format for SOAP and REST implementations.	Different O\X Mappers implemented in the same service		Content negotiation
This is similar to the previous one; service provides can be requested using the URL query string (empty body) or by an HTTP payload for SOAP and REST implementations.	Intermediary router in front of several transformation service agents	Normalizer	Dual Protocol (Messaging, not Transport); multichannel endpoint

Message Transformation also covers classic Sort and Validate, and it is realized on XML Transformation engine(s). A Partial Validation pattern is supported for performance improvement as one of the validation forms. Throttler, Load Balancer, and Multicast are the forms of the traditional intermediate routing. A Generic (and therefore pretty undisruptive) Service Agent SOA Pattern is the most common way of addressing the mentioned tasks, but we are not sure that such a vague definition could help anyone. The components required for each realization are listed in the previous table, but what we really want is an agnostic way of fulfilling the tasks.

Establishing a Service Inventory

For a synchronous Agnostic Composition Controller, we will reuse most of the supporting service building blocks created for the asynchronous Service Broker. The most obvious ones are listed as follows:

- Inventory endpoint is a service that will provide us execution plans together with Oracle Rule Engine. The input parameter will be our Message Header.
- Logging and auditing services.
- The CTU message container. Naturally, all XML entities will be the same.

In addition to this, we will have to build two types of endpoints:

- Generic synchronous and asynchronous endpoints (generic), which are required for dynamic routing to the actual services. Endpoint addresses will be obtained from the execution plan during runtime.
- The Transport Adapters that we discussed earlier.

Asynchronous Agnostic Composition Controller

All prerequisites are identified, required patterns are discussed, and building blocks and components are defined. Now, we can take a step by step walk through the implementation of a dynamic service broker with agnostic capabilities. Again, we assume that you are familiar with OSB; therefore, we will skip obvious operations. The Oracle pack for Eclipse will be our main tool. First, we will create an ESB project and folder for Service Broker (you can use any name you want, but we decided to call it this way as it reflects the project's functional purpose). Then, we will create several proxies for Service Broker and Generic Adapter, which will be used to route to the custom service. In addition to this, we will need separate folders for protocol adapters, Facades, XQueries, XSDs, and XSLTs. To execute the following steps, we advise you to keep the SB sequence diagram close together with the overall solutions block diagram, presented on the functional decomposition figure (refer to the previous chapter).

Business Delegate (main dispatcher)

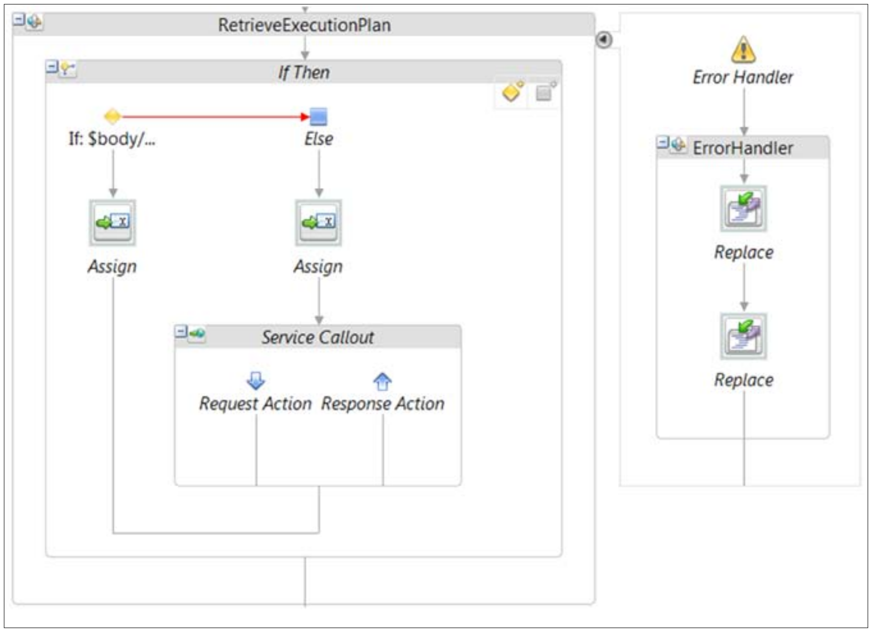
The Business Delegate is responsible for the following tasks:

- Initial verification of inbound message
- Extraction of rooting slip using detected message header
- Looping variables initiation
- Traversing through tasks in an EP's task list

Execution plan extraction

The first operation in a main dispatcher as Agnostic Composition Controller will be the extraction of a routing slip with a task list.

This is the Service Callout of the execution plan business service. Before extracting EP, we must check for the presence of an execution plan in the inbound message. If it's there, we assume that EP was provided by an adapter or a master Composition Controller and accept it. In the case of an error, **ErrorHandler** will record a detailed description into the log in most of the common cases: the **Lookup service** is not available, EP is not found, or the CTU message container already exists in the message body. Take a look at the following screenshot:

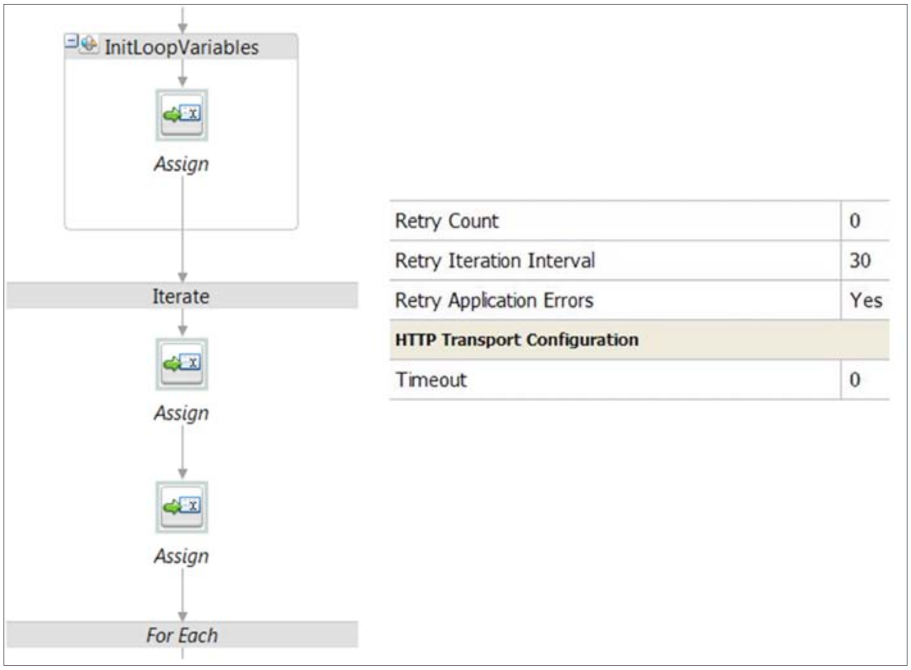


Just a small reminder, the first step in the message flow is actually saving the message body (`$body`) in the `v_originalBody` variable (for obvious purposes).

Parameter initiation

We have to initiate looping variables and current values for all the Message Container parts, so please refer to the following script.

One important thing must be realized clearly: for Agnostic Composition Controller, we cannot apply Configuration Details (in the simple Proxy example in the preceding screenshot) for `main` for parameters in the following screenshot:



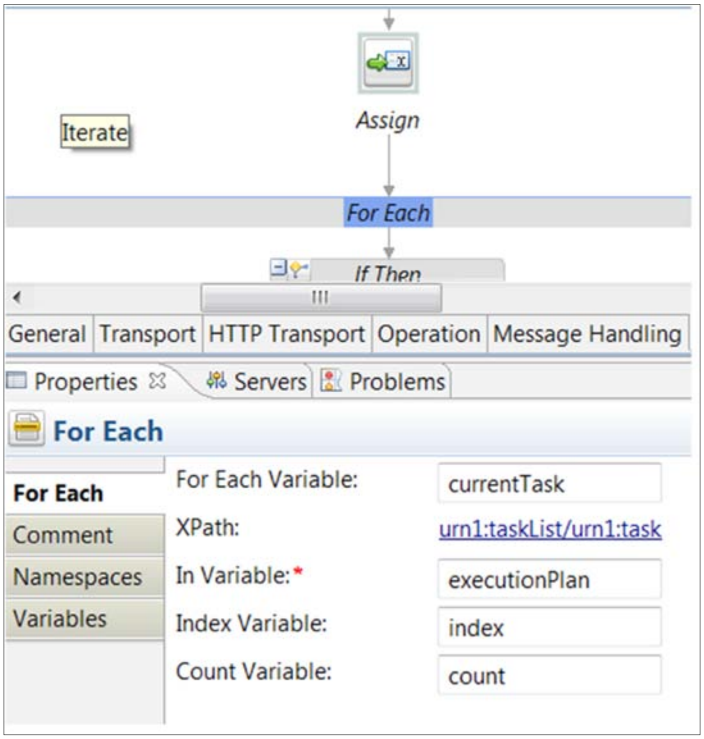
Retry Count	0
Retry Iteration Interval	30
Retry Application Errors	Yes
HTTP Transport Configuration	
Timeout	0

The reason is clear—if a controller is agnostic, then these values will depend on the nature of the business process and the nature of the services we will dynamically invoke. We shall pass these parameters in EP, as shown in the following code, and create an additional loop for retries within the task loop, with the counter set in the **Assign** operation. This is demonstrated in the preceding screenshot.

```
<con:stage name="Iterate">
    <con:context>
    <con1:varNsDecl
        namespace="urn:com:telco:ctu:la:ctumessage:v01"
        prefix="urn"/>
    <con1:varNsDecl
        namespace="urn:com:telco:ctu:la:processheader:v01"
        prefix="urn1"/>
    <con1:varNsDecl
        namespace="urn:com:telco:ctu:la:messagetrackingdata:v01"
        prefix="urn2"/>
    <con1:varNsDecl namespace="urn:com:telco:ctu:la:payload:v01"
        prefix="urn3"/>
    <con1:varNsDecl
        namespace="urn:com:telco:ctu:la:messageheader:v01"
        prefix="urn4"/>
    </con:context>
    <con:actions>
        <con2:assign varName="taskInput">
        <con1:id>_ActionId-3019043452746376819-
            7d5319e3.13a7dc61937.-76d7</con1:id>
        <con2:expr>
        <con1:xqueryTransform>
        <con1:resource
            ref="CTUFusion_BUS/Resources/Xquery/ServiceBroker/
            XQ_CTUMessage"/>
        <con1:param name="messageTrackingData">
        <con1:path>$body/urn:CTUMessage[1]/
urn2:MessageTrackingData[1]</con1:path>
            </con1:param>
            <con1:param name="messageHeader">
            <con1:path>$body/urn:CTUMessage[1]/
                urn4:MessageHeader[1]</con1:path>
            </con1:param>
            <con1:param name="Version">
            <con1:path>$body/urn:CTUMessage[1]/urn:Version[1]/
                text()</con1:path>
            </con1:param>
            <con1:param name="processHeader">
        <con1:path>$body/urn:CTUMessage[1]/
            *:ProcessHeader[1]</con1:path>
        </con1:param>
        <con1:param name="MessageType">
        <con1:path>$body/urn:CTUMessage[1]/urn:MessageType[1]/
            text()</con1:path>
        </con1:param>
        <con1:param name="payload1">
        <con1:path>$body/urn:CTUMessage[1]/
            urn3:Payload[1]</con1:path>
        </con1:param>
        </con1:xqueryTransform>
        </con2:expr>
        </con2:assign>
```

Main tasks loop

Now, we can start looping over every task in a task list by using the **For Each OSB** flow control. You can see a set of variables in the following screenshot:



For Each Variable: currentTask

XPath: urn1:taskList/urn1:task

In Variable:* executionPlan

Index Variable: index

Count Variable: count

You must be thinking, "What type of tasks did we decide to support in our execution plan (VETRO)?" Naturally, for transformation, we will need to extract and assign working variables into data for the following operations:

- Transformation input
- Transformation output
- Current payload

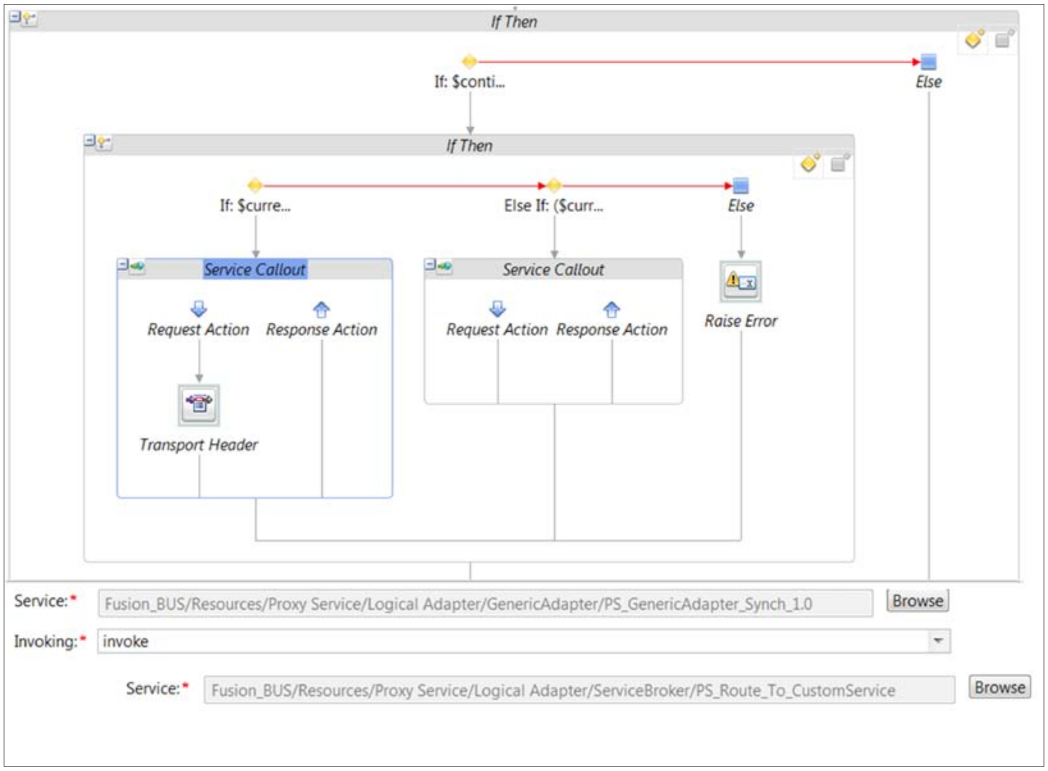
We will use them in the transformation Proxy of this type of task.

For the retry looping option, we will set "continue" as a flag to continue with the next operations.

Service invocation

This is the essence of this Business Delegate and the core of the task loop. If the continue flag is on, that is, there are no errors in the previous operation and we can continue with the current task, `$continue='true'`, generally we have the following two scenarios:

- We could perform any combination of VETRO operations, which are in the left path of the **If** branch. We will call Adapter Factory (or Generic Adapter, as it's displayed), where we will delegate all generic operations for transformation, enrichment, validation, and filtering, before moving on to Transport Adapter.
- We could invoke any service directly, as a synchronous request-response or as fire-and-forget; this is the path to the right (**Else If**). In this case, we also perform the service callout to the Custom Service Proxy, which is acting as a placeholder for actual service address substitution.



Service:* Fusion_BUS/Resources/Proxy Service/Logical Adapter/GenericAdapter/PS_GenericAdapter_Synch_1.0

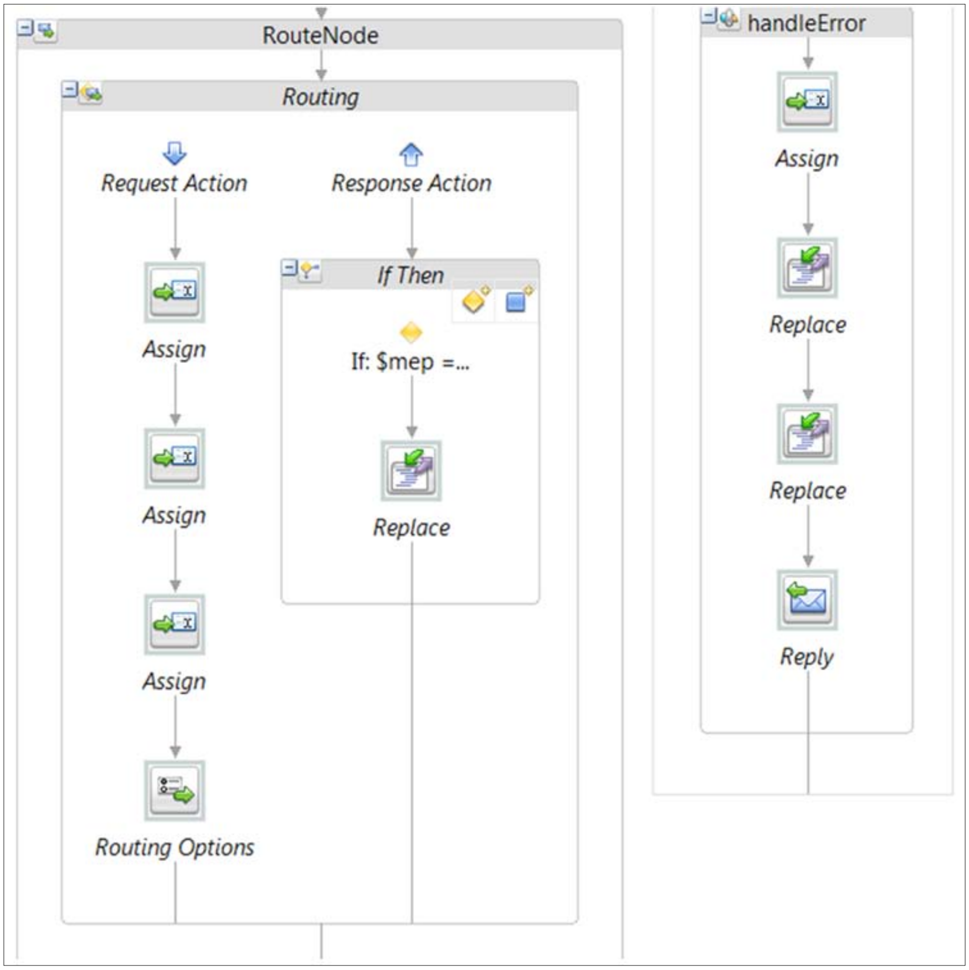
Invoking:* invoke

Service:* Fusion_BUS/Resources/Proxy Service/Logical Adapter/ServiceBroker/PS_Route_To_CustomService

Generally, that's all about the core path in Main Dispatcher. We will skip error handling for now as it's not that important to understanding the logic, and will only make us deviate from Service Broker realization. We will definitely discuss it later on. Now, we will discuss the paths where we routed our container to, starting from the right branch for custom service invocation as it's relatively easier.

Invoking custom services

This Proxy Service (refer to the next screenshot) pattern is acting as a Facade for the actual business service.

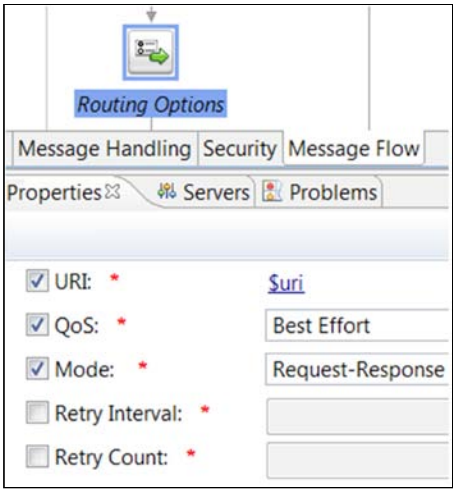


Again, due to the nature of our composition controller, invocation could be either synchronous or one-way synchronous. In the request pipe, we configure three assignment settings that are required for the last operation; that is, routing:

- **Message Exchange Pattern** (**MEP**)
- Message payload
- **Uniform Resource Identifier** (**URI**)

In a routing option, we will use `$uri` from the previous assignment to dispatch a message to the actual service. As you can clearly see here, we have left the retry interval and retry count empty, as they are not applicable and we handle them using EP parameters. You will have also noticed that the mode is **Request-Response**. Therefore, for one-way MEP, we should suppress the response payload by providing a proper error code in message tracking data:

```
<CTUMessage xmlns="urn:com:telco:ctu:la:ctumessage:v01">
    <Payload xmlns="urn:com:telco:ctu:la:payload:v01"/>
    <MessageTrackingData xmlns="urn:com:telco:ctu:la:messagetrackingd
ata:v01">
        <MessageTrackingRecord>
            <ErrorCode>0</ErrorCode>
        </MessageTrackingRecord>
    </MessageTrackingData>
</CTUMessage>
```



☑ URI: * $uri
☑ QoS: * Best Effort
☑ Mode: * Request-Response
□ Retry Interval: *
□ Retry Count: *

Implementation of the business-agnostic VETRO pattern on OSB

Now, we will look at the left branch, starting with **Generic Adapter**. It's more complex than just dynamic service invocation with our standard message container (remember, we extracted the payload from the container before invoking the service).

Invoking Generic Adapter

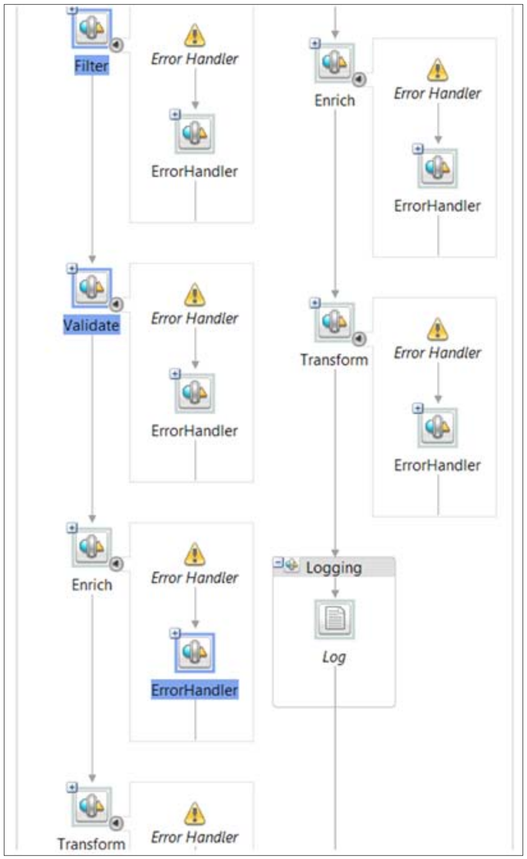
The Generic Adapter is the Adapter Factory pattern's realization, performing common-for-all adapters VETRO operations on both pipelines.

As always, initially we assign working variables for each part of the message container and do validation for the container's consistency. After that, we go through the VETRO steps.

Technically, it's a sequence of flow stages, and in each step we check for the task step name, for instance, `filter`:

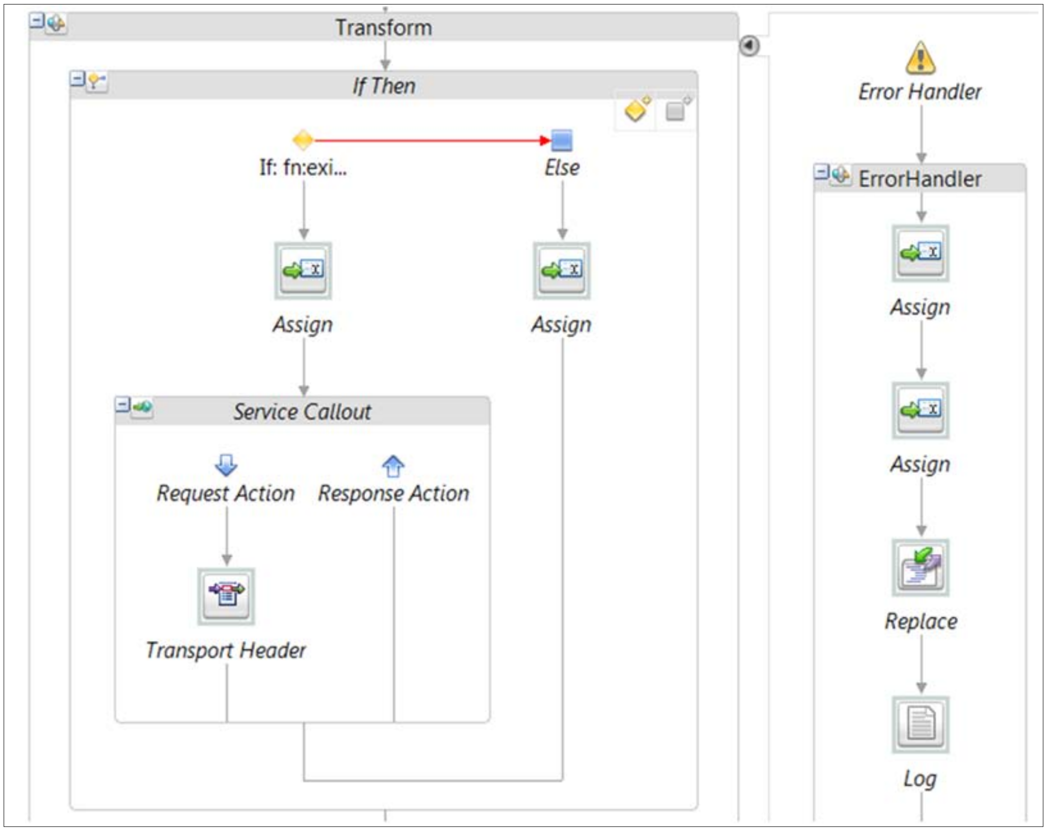
```
fn:exists($body/*:CTUMessage/*:ProcessHeader/*:ExecutionPlan/*:taskLis
t/*:task/*:invoke/*:serviceTask/*:taskSteps[@flow='request']/*:filter)
```

In each stage (where it's necessary), we can use the **Publish** operation (from **Stage Actions**) to invoke the Audit Proxy service. It's also done by using **Routing Options**. The auditing level for every process is maintained by the EP element:



Transformation

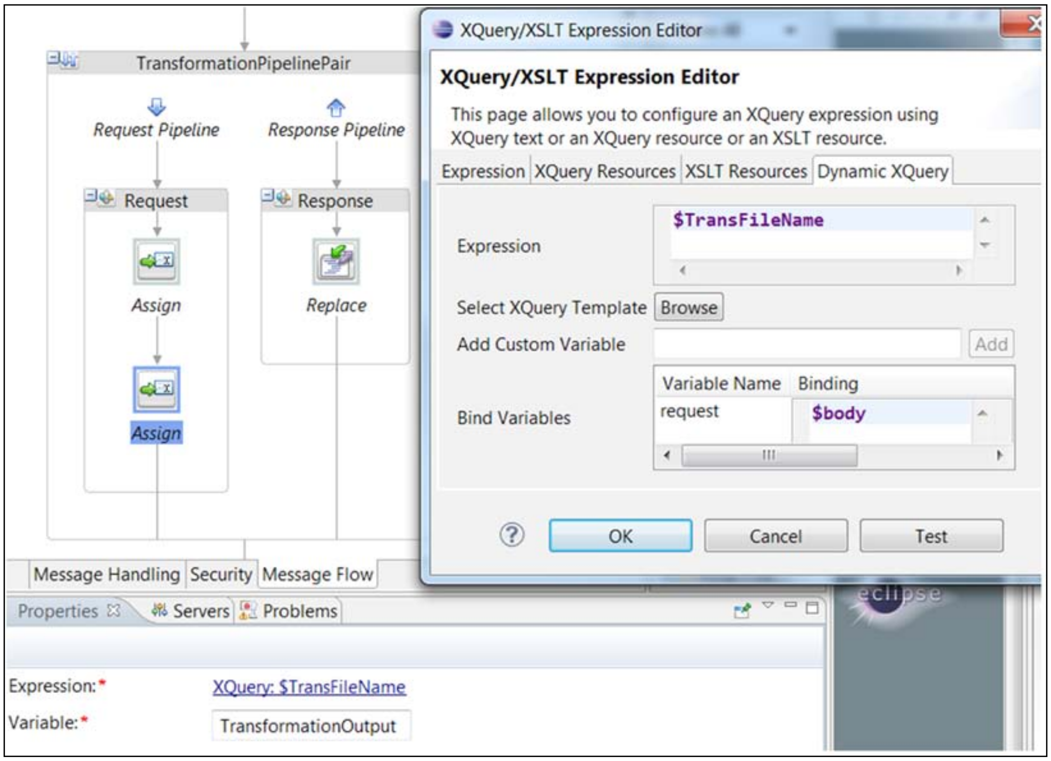
Naturally, the most common operation before the invocation of Transport Adapter is Transformation, as described in the following screenshot:



We decided to take this operation from Generic Adapter for better modularity and separation of concerns. Just like we did for Message Broker, we want to have a possibility to use different transformation engines/techniques that are interfaced by transformation proxies, as presented in the following code. Currently, it's a very simple Proxy Service pattern with two assign operations, where at first we assign the transformation query:

```
data($body/*:CTUMessage/*:ProcessHeader/*:ProcessContext/*:
ParameterValue[@name = 'TaskTransformation']) to TransFileName
```

Then, apply the operation to the message body (through dynamic XQuery) with the following reassignment to the transformation output:



XQuery/XSLT Expression Editor

This page allows you to configure an XQuery expression using XQuery text or an XQuery resource or an XSLT resource.

Expression: $TransFileName

Bind Variables:

Variable Name	Binding
request	$body

Expression:* XQuery: $TransFileName

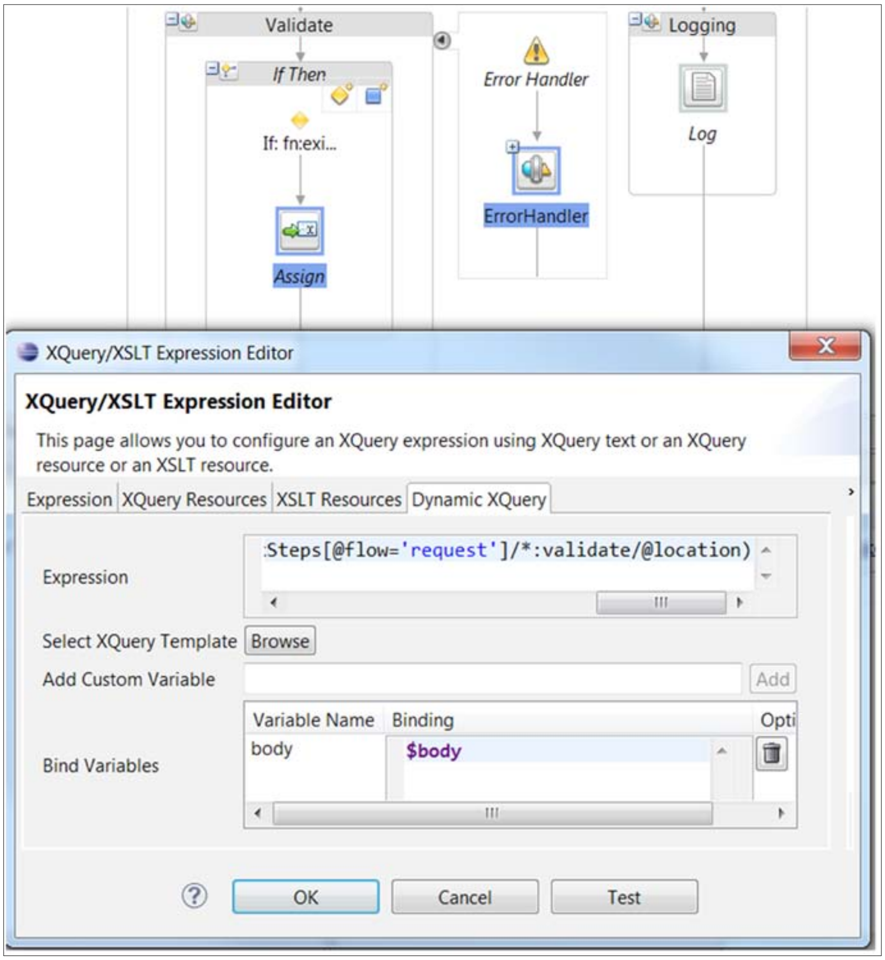
Variable:* TransformationOutput

Implementation of message validation on OSB

In a response pipe, we'll replace a message body with the transformation output.

Validation

Validation is similar to transformation as it uses dynamic queries for the XSD validation of the message body. The major difference is that we will not create a separate Proxy for this operation, as Oracle Service Bus can perfectly implement effective complete or partial validation by using the standard XQuery mechanism, as shown in the following screenshot:



XQuery/XSLT Expression Editor

This page allows you to configure an XQuery expression using XQuery text or an XQuery resource or an XSLT resource.

Expression: `:Steps[@flow='request']/*:validate/@location)`

Select XQuery Template

Add Custom Variable

Bind Variables:

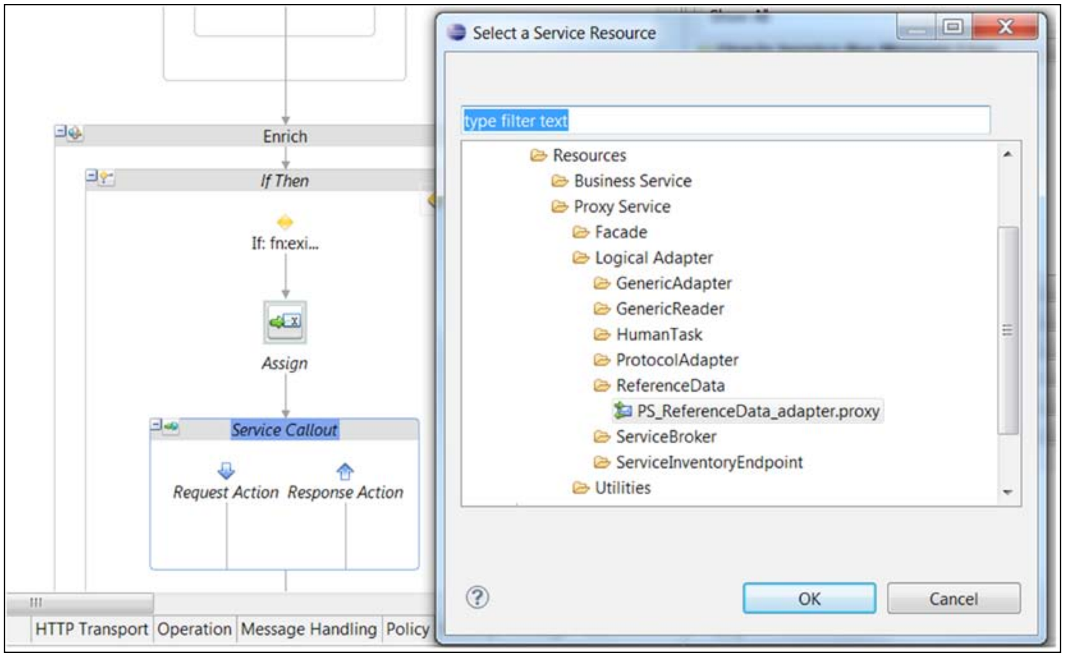
Variable Name	Binding	Opti
body	$body	

Of course, we do not use the standard operation (**Validate from OSB Flow Control**). We'll just extract XQuery from the task's parameter by using the following expression:

```
data($body/*:CTUMessage/*:ProcessHeader/*:ExecutionPlan/*:taskList/*:
task/*:invoke/*:serviceTask/*:taskSteps[@flow='request']/*:validate/@
location)
```

Enrich

This operation denotes a callout to the external Entity service to extract additional data. This is also a dynamic invocation of the URI provided in the EP's task parameter.



Implementation of Enrich functionality on OSB

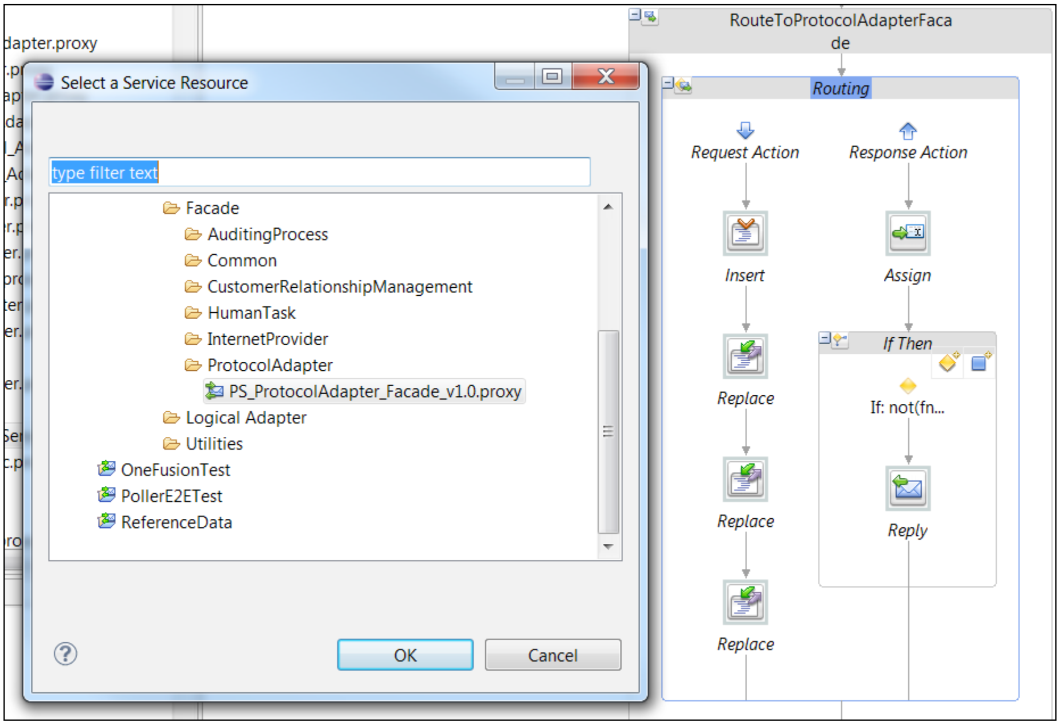
Of course, the proxy for the reference data adapter must be created. We will look at it in *Chapter 6*, *Finding the Compromise – the Adapter Framework*. Right now, we will just repeat what we have said in the functional decomposition table. Practically, there are no dedicated adapters here, compared to what we see in SOA Suite. So, we will use the JCA adapter approach, especially for DB, following the sequence described earlier.

We will skip the filtering operation for brevity, but you can easily figure it out by yourself. Naturally, that will be another dynamic query in the `Assign` operation, checking the message body for the presence of XPath, which is provided as a parameter in the EP **Filter** task. If the XPath pattern is recognized, the **Discard** flag is returned and further processing is terminated by the **Raise Error** operation with the following message: **The message has been filtered**.

Now, we have come to the last part of the VETRO sequence: operate. We will pass the message to the Adapter Facade Proxy Service, which will do the actual dispatching.

Operate

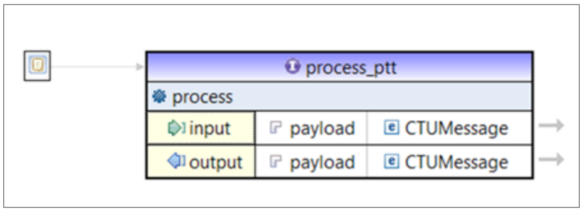
The operate task is actually the routing implementation. In the request action pipe, we set ABO and Transport Header (which is responsible for transporting the JCA parameters that are related to the Transport Adapter properties):



Implementation of the Operate functionality on OSB

Replace the message body with the results of the transformation (or the original body if the transformation step was omitted). The Adapter Facade Proxy is (traditionally) based on `CustomService.wsdl`, our dummy service which we use for dynamic binding. Please refer to the simple WSDL in the following screenshot. The **CTUMessage** payload is our generic message container.

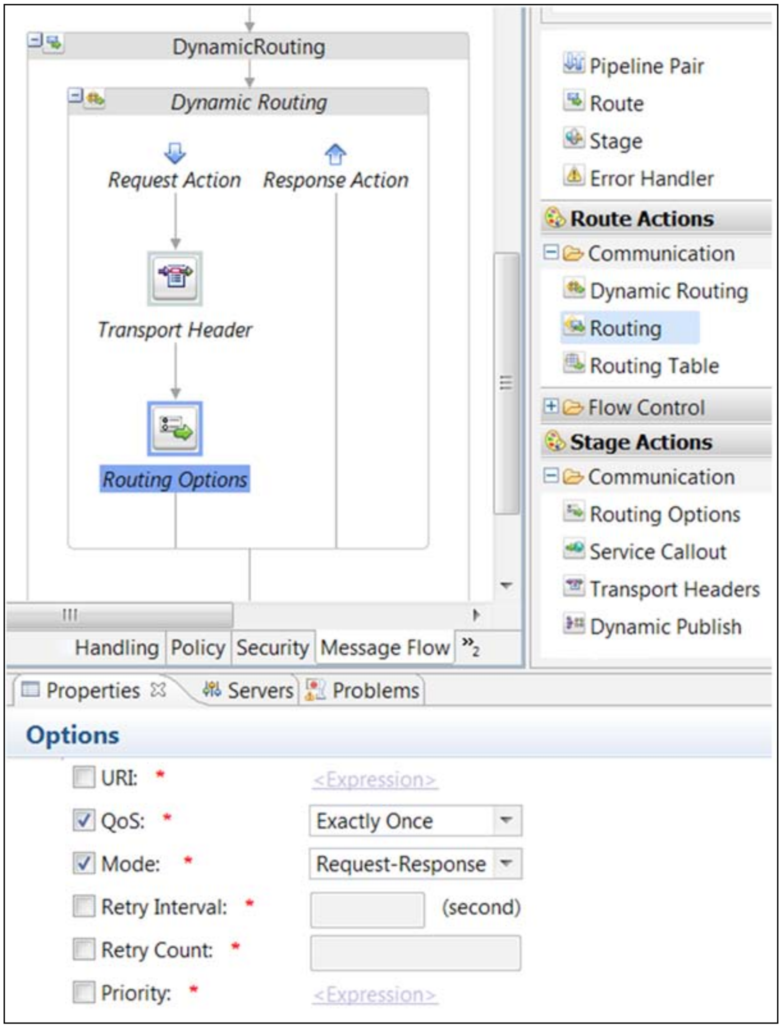
That's the beauty of the "contract first" approach, isn't it?



process_ptt		
process		
input	payload	CTUMessage
output	payload	CTUMessage

Routing to the protocol adapters (delivery)

Proxy is relatively simple with one major action: dynamic routing with XQuery as a service. XQuery's function checks for the type of protocol adapter we want to invoke and route to the appropriate URI.



Options

☐ URI: *	<Expression>	
☑ QoS: *	Exactly Once	
☑ Mode: *	Request-Response	
☐ Retry Interval: *		(second)
☐ Retry Count: *		
☐ Priority: *	<Expression>	

OSB dynamic routing

The XQuery function checks for the type of protocol adapter we want to invoke and route to the appropriate URI, as shown in the following code:

```
xquery version "1.0" encoding "Cp1252";
(:: pragma  parameter="$request" type="xs:anyType" ::)
```

```
declare namespace xf =
    "http://tempuri.org/CTUFusion_BUS/Resources/
    XQuery/routing/XQ_Request_ProtocolAdapter_Route/";
declare namespace ctx = "http://www.bea.com/wli/sb/context";
declare function xf:XQ_Request_GAdapter_Route($request as
    element(*))
    as element(*)  {

    if (data($request//*:AdapterMessage/*:protocol)= 'DB')
        then
            ( <ctx:route>
             <ctx:service
                 isProxy='true'>CTUFusion_BUS/Resources/Proxy
                 Service/Logical
                 Adapter/ProtocolAdapter/PS_Database_
                 Protocol_Adapter</ctx:service>
             </ctx:route>)
        else if (data($request//*:AdapterMessage/*:protocol) =
            'HTTP')  then
            (  <ctx:route>
             <ctx:service
                 isProxy='true'>CTUFusion_BUS/Resources/Proxy
                 Service/Logical
                 Adapter/ProtocolAdapter/PS_Soap_
                 Protocol_Adapter</ctx:service>
             </ctx:route> )
        else if (data($request//*:AdapterMessage/*:protocol) =
            'REST')  then
            (  <ctx:route>
             <ctx:service
                 isProxy='true'>CTUFusion_BUS/Resources/Proxy
                 Service/Logical
                 Adapter/ProtocolAdapter/PS_REST_
                 Protocol_Adapter</ctx:service>
             </ctx:route> )
        else if (data($request//*:AdapterMessage/*:protocol) =
            'JMS' or data($request//*:AdapterMessage/*:protocol) =
            'SOAPJMS') then
            (  <ctx:route>
             <ctx:service
                 isProxy='true'>CTUFusion_BUS/Resources/Proxy
                 Service/Logical
                 Adapter/ProtocolAdapter/PS_JMS_
                 Protocol_Adapter</ctx:service>
             </ctx:route> )
```

```
else if (data($request//*:AdapterMessage/*:protocol) =
    'FTP')  then
    ( <ctx:route>
     <ctx:service
         isProxy='true'>CTUFusion_BUS/Resources/Proxy
         Service/Logical
         Adapter/ProtocolAdapter/PS_FTP_
         Protocol_Adapter</ctx:service>
     </ctx:route> )
else if (data($request//*:AdapterMessage/*:protocol) =
    'SFTP')  then
    (<ctx:route>
     <ctx:service
         isProxy='true'>CTUFusion_BUS/Resources/Proxy
         Service/Logical
         Adapter/ProtocolAdapter/PS_FTP_
         Protocol_Adapter</ctx:service>
     </ctx:route> )
   else if (data($request//*:AdapterMessage/*:protocol) =
       'HumanTask')  then
     ( <ctx:route>
      <ctx:service
          isProxy='true'>CTUFusion_BUS/Resources/Proxy
          Service/Logical
          Adapter/ProtocolAdapter/PS_HumanTask_
          Protocol_Adapter</ctx:service>
      </ctx:route> )
   else if (data($request//*:AdapterMessage/*:protocol) =
       'SMTP')  then
     (<ctx:route>
      <ctx:service
          isProxy='true'>CTUFusion_BUS/Resources/Proxy
          Service/Logical
          Adapter/ProtocolAdapter/PS_SMTP_
          Protocol_Adapter</ctx:service>
      </ctx:route>)
   else if (data($request//*:AdapterMessage/*:protocol) =
       'OSB')  then
      ( <ctx:route>
       <ctx:service
           isProxy='true'>CTUFusion_BUS/Resources/Proxy
           Service/Logical
           Adapter/ProtocolAdapter/PS_OSB_
           Protocol_Adapter</ctx:service>
      </ctx:route> )
```

```
        else (
            <ctx:route>
            <ctx:service isProxy='true'>DEFAULT</ctx:service>
            </ctx:route>)
        };
    declare variable $request as element(*) external;
    xf:XQ_Request_GAdapter_Route($request)
```

That's it. You can extend this list as much as you want and build your own adapters by importing the JCA and WSDL files from the SCA adapter composite. Respect the contract-first principle. Bear in mind that lots of parameters can be passed to the adapter itself; even the JNDI name that you give in SCA can be declared as a variable (view it as a property in `composite.xml` for related BPEL) and managed dynamically or via System MBean Browser in `soa_domain` (WLS console).
The sequence of actions for the DB adapter, for instance, could be as follows:

- Transport Data Adapter receives CTUMessage as an input with JCA properties
- Adapter extracts data source name and SQL statement
- Adapter calls the XQuery executing `fn-bea:execute-sql()`

For your Service Broker, we recommend that you build this list gradually, starting from the most common and generic adapters: HTTP/SOAP first. Some adapters are not simple like for JSON payload, where some Java coding will be necessary to remap XML to the JSON format.

Conclusion – the pros and cons of OSB Agnostic Composition Controller

We separated the synchronous part of our Agnostic Composition Controller and moved it away from SCA. Now, our asynchronous Service Broker in SOA Suite looks much more concise and modular, and all synchronous service interactions are delegated to the EBS Framework where they belong. You can clearly see the resemblance in implementation steps for Message Broker and Service Broker, but the difference is quite distinguishable:

- Just look at the list of transport protocols we can support (yes, they are all JCA adapters, but they are pretty standard).
- Although some coding efforts are still required (XQuery for instance), it is comparably less than the custom servlet-based approach.
- Dynamic Routing, Invocation, and Transformation is really handy.

- Most importantly, by following the contract-first approach, we delivered a modular solution! The concept of Proxy Service allows us to gracefully assemble truly loosely coupled components. We have highly manageable and completely decoupled layers that are based on patterns (SOA and EAI), namely Business Delegate, Service Broker, Adapter Factory, and Adapters.
- EAI and JEE patterns are not dead (as you probably heard on some SOA symposiums). Appropriately applied on the service composition level, they can improve modularity even more. Thanks to them, we can Aggregate, Split, Route, Transform, Enrich, Filter, and Validate. Combined with the power of the agnostic controller, we can do it dynamically and process-agnostically.

Summary

Oracle realization of the Service Bus is based on the Business Service / Proxy Service concept, where every service is treated individually. That's what you will learn in every book dedicated to OSB (such as *Oracle Service Bus 11g Development Cookbook, Guido Schmutz and Mischa Kölliker, Packt Publishing*). This kind of personal-touch approach for every service has a strong reason as we would like to maintain the highest performance possible and desirable level of modularity. Only one more or less complex pattern (refer to the second figure in the *Implementing a basic proxy on OSB* section) is really provided out of the box in OSB: Split-Join. Using this pattern, we can split a really big message that has clearly distinctive separate parts (such as order line in a big order), do the parallel calls to order the line processor, gather responses in a join phase, and deliver the processed payload.

You have to implement all other SOA patterns yourself, just like we have covered in this chapter:

- Message Enrichment (Callout action); it's not a decorator though as we are not adding new features to our contract.
- Static Routing (Static Routing table).
- Dynamic Routing (Dynamic routing action, XQuery as a possible destination).
- Transformation (partial or complete) using XQuery or XSLT. Remember, XQuery is more complex than XSLT in general and not an XML. You do not have nice graphical features for elements mapping, but you have much more flexibility there.

All of these implementations are not a problem as OSB has a lot of features to support your development. The challenge will become apparent when you have hundreds (and most probably thousands) of synchronous services you will have to decouple, interconnect, delegate, and route.

For the services with similar operations or similar compositions, Agnostic Controller will help you minimize the number of atomic proxies that will turn your state-of-art ESB into spaghetti-mess if developed in an uncontrollable manner. We believe that the way of establishing agnostic Service Broker with Dynamic Intermediate Routing presented here will save you from lots of headache. Not all features and components of the actual solution were presented. We omitted inbound and outbound Service Facades, holding features that are specific for Northbound and Southbound parts of OSB, but it was our intention (refer to the table in the *Detailed analysis – functional decomposition* section) to concentrate on generic tasks. Partly missing components will be covered in the Adapter framework discussion.

We stepped beyond the traditional OSB features and implemented some of the mediation patterns mentioned in the functional decomposition table. We did it with the intention to demystify the most commonly used SOA terms—Service Agent and Service Facade (patterns). If you are using this book as an additional resource to prepare for your SOASchool SOA Architect (`http://www.soaschool.com/certifications/architect`) exams, we can give you a simple rule regarding this terminology:

- Service Agent is an event-driven program (module) without a publicly exposed contract. It's most commonly located in the SOA technical infrastructure between the atomic services, intercepting the interactions and reacting to certain events. Sometimes, it is called an interceptor or a listener. We just delegate routing, transformation, and other features to this module and use OSB as a container for agents. At the same time, SCA Mediator can have a contract. Is it an agent?
- Service Facade is a module that resides within an individual service, decoupling logic and underlying resources and contracts, allowing them to evolve independently. We used them in our preceding design. It's probably the closest thing to the decorator pattern. Again, SCA Mediator, as an intra-composite element, exists inside the task service. Is it a Facade?

Speaking further on realization SOA and EAI patterns from the functional decomposition table, we can draw some more conclusions. As you probably noticed, the EAI patterns look suspiciously similar to those decaled in Camel/ServiceMix ESB (Fuse is a RedHat commercial version `http://www.redhat.com/fusesource/downloads/`), they were systematized by Gregor Hope and Bobby Woolfe in *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solution*. That's absolutely true. Even the concept of a Routing Slip (we call it an execution plan) is similar, although this is not an exclusive property of Camel:

- As Camel is one of the best open source Oracle rivals, we just reconfirmed OSB's capability to keep up with the strongest performers.
- OSB can do it by means of a simple Messaging Metadata pattern (EP XML in Message Header), without inventing Java-like DSL.
- When it comes to truly lightweight but comprehensive solutions without any compromises in features, think twice before building your own Message Broker as we did at the beginning of this chapter for learning purposes. You really will have it all in Camel/Fuse (mind you, we are not working for Oracle, at least for the moment). We are just happy to confirm that the Oracle, Apache Camel and JBoss Fuse teams share the same SOA vision.

Finally, what we haven't done (neither should you) with your OSB agnostic controller:

- The OSB Service Broker is much closer to the Message Broker we discussed than to Orchestration (that's why we discussed the Message Broker, actually). It's still a good old RTD interchange pattern. Do not implement any business logic here! Do not orchestrate in ESB, only decouple, transform, and route.
- The logical continuation of the preceding point is that if you think that you can do the callout to a Java function from the OSB performing a business logic—don't! Count how many SOA principles you will break.
- Do not use your OSB Proxy Service for the patterns different from what we discussed. If it looks like implanting business logic into Proxy—stop here. Implement atomic service *and* then present it via OSB.

- It's a very thin line between the Split-Join pattern and batch processing in ESB. Although it's quite possible to do some batch processing between the two DBs (your DBA is prohibited from using DBLinks and, for some reason, believes that SQL*Loader is for dinosaurs), try to avoid that design. Use the Oracle ODI product, if you can.
- As deduced from all the preceding points—OSB can act as an adapter layer, but still Oracle recommends BPEL for this purpose. Think about it.

There are plenty of common components that are utilized by both Service Brokers, synchronous and asynchronous. In EBF and EBS frameworks, we share common Audit, Logging, and ErrorHandling services. However, most importantly, we use the same execution plan (scontroller's routing slip). This configuration entity and part of the generic message container is based on the service metadata that is maintained on Enterprise Service Repository. This core framework is the subject of our next chapter.

5
Maintaining the Core – Service Repository

In our endeavor to create the Agnostic Composition Controller using probably the most popular SOA pattern, we demonstrated the role of two out of three core SOA operational frameworks, covering Service Creation, Composition, and Governance patterns. Focusing on its most complex realization, based on dynamic service discovery and invocation, we delved into the problem of Metadata Centralization and its relation to message structure and service layering: logical and physical; we cannot disregard these any longer. Building a Service Inventory without a plan in mind will produce results far worse than point-to-point spaghetti. Unfortunately, there is no tool to guide us right after the installation. Only the SOA principles (yes, to a meaningful extent) help us build a practical SOA inventory with a working Governance Reference model.

This chapter is the focal point of this book. First, we will give you a vendor-neutral taxonomy for a Service Inventory you can use with any tool without significant investments. Second, we will see how to roll it up to the Oracle Enterprise Repository and Service Registry. The Foundational Inventory pattern, together with the Inventory Implementation and Governance patterns, is the biggest category in the SOA patterns catalogue and covers all the aspects of a service lifecycle. We will try to cover all these aspects—from requirement analysis, service design, testing, and runtime monitoring to decommissioning.

Flexible taxonomy for Service Repository

Taxonomy is a method used to classify entities, items, and categories, among other things. The ultimate goal of taxonomy is to establish a hierarchy or structure of categories (in our case, all SOA artifacts). In this sense, an **ontology** is a wider concept because it is primarily responsible for the identification, separation, and description of categories, and it is further used in our classification to establish relations. Ontology is closer to the theory of classification, originally defined by Aristotle as the *first philosophy*. Recall that that's where we started in the first chapter—the identification of frameworks, principles, and characteristics essential for our service-oriented architecture. Finally, we put them together and declared relations and dependencies.

Now, we extend this initial hierarchy with the essential SOA elements to maintain optimal composability as an immediate target and promote effective SOA Governance. The outcome will lay a strong foundation for the entire Enterprise SOA Governance from all standpoints—software architecture, business, and operations.

General objectives

Actually, we should start the book with this chapter, and only after establishing a solid foundation should we proceed with the on-top frameworks. The deceiving simplicity of WSDL-based service creation plays poor tricks on SOA-like projects, focusing only on the API aspects of service orientation. The blueprint of the Service Repository is a fundamental element of the Enterprise SOA architecture that demands commitment on all levels, primarily from architects. Surprisingly, not all architects are willing to support this architectural layer due to its complex nature and the fact that the direct benefits don't seem all that obvious. "We will think about it tomorrow. Today we stay on target—our delivery deadline." For this kind of attitude (read: project delivery mode from the initial example of *Chapter 1*, *SOA Ecosystem – Interconnected Principles, Patterns, and Frameworks*), architects are not required. If one does not know to which port one is sailing, no wind is favorable. The orchestration layer and ESB are not directions, but merely physical containers for our Service Inventory. These containers have been packed mindlessly by different teams working in parallel with a different understanding of service orientation, and will need total refurbishing every four or five years.

Think of it this way—your SOA Enterprise architecture is a megalopolis, inhabited by doctors, milkmen, truck drivers, firefighters, and so on. They are your service providers and consumers. To operate smoothly, they need to be positioned rationally (logical layering), that is, a fire brigade will serve no good if it cannot reach a certain location in the predefined time (runtime interoperability). Any good citizen should be capable of locating a (legally) required service with minimal effort, for example, by browsing the yellow pages (runtime Discoverability on the Service Registry). Most importantly, a citizen must be able to understand the nature of the service—look for a particular doctor, select the right one, and be sure that this doctor is located exactly where mentioned (interpretability of the Service Registry). These are our runtime features of the Service Inventory.

At the same time, you, as the good governor of this megalopolis, must plan the layering wisely—concentrate on one type of service, such as financial institutions (**Inventory Centralization**), and evenly spread another, such as waste removal and law enforcement (the **Cross-Domain Utility Layer**). To do so properly, you need a wide variety of information about these services (service records about your law enforcement officers), available in the Service Repository. Only a precise subset of this information will be used during runtime by other services. Some other types of service records will be constantly used to measure their (services) runtime behavior, and based on that, we will decide about service promotion or decommission. These are the design-time features of the Service Inventory.

Is the city too big for you to handle (**Enterprise Inventory**)? Don't take it personally. There could be many reasons, and most of them are out of your control—you are an architect and not a CEO after all. Start with one district (**Domain Inventory**) and constantly prepare for expansion. However, remember that neglecting even one of the patterns (presented in bold earlier in this section) will cause significant trouble in your domain. Thus, ideally, this chapter should have been at the beginning.

Nevertheless, nothing can prevent experienced architects from reading this chapter first. If you've played enough with ESB and BPEL and are now looking to get things organized, you know by now the principles to enforce and the gaps to fill.

However, the objectives we chase by implementing the Service Repository and the design rules we will implement to achieve these objectives (utilities) remain worth highlighting.

Objective	**Design rules to achieve the objectives**
Reliability	• The component/modular approach • The optimal number of components in the compositions (this is the balance between composition complexity and components size, validated during the JIT testing phase) • Avoiding long-running multiphase commit transactions • Implementing an atomic transaction coordinator • Rigorous and continuous automated assembly/testing during development • Building only when necessary • Maintaining potential single points of failure redundantly
Reusability	• Business agnostic components • Designing with reusability in mind • Having Discoverability during design time and, consequently, at runtime • The standardization of the service contract, including CDM and the implementation of SBDH, MC, Audit and Acknowledge messages, and protocol standardization • MEPs standardization/optimization • Promoting changes through configuration
Maintainability	• Unified scalable components with measurable and predictable behavior • Centralized configuration store/management • Automated assembly and deployment framework • Unified Utility patterns (Logger, ErrorHandler, and PolicyAudit) • Autonomous implementation of services • Observing service layering (do not mix Utility and Task Orchestrated services in one layer (horizontal) or combine entity services with proprietary adapters (vertical))

Objective	Design rules to achieve the objectives
Performance	• Single-purpose scalable components • Minimal-to-none state maintenance • A minimal number of transformations/translation • Minimal protocol bridging and staying with lightweight protocols, with the possibility to minimize marshalling/serializations • Optimizing message sizes • Optimizing of queue processing, parallel, balanced multi-consumer, advanced datatype queues with message filtering and queue content propagation • Preparing components for clustered, cached, or balanced deployment • Adjustable logging/audit levels • Avoiding singletons

All presented ilities and related design rules are equally important and will have a proportional impact on the Service Inventory. However, the money is in the **Discoverability** principle (presented in bold in the preceding table). If we are unable to discover the required service or artifact, or understand what we have discovered, our SOA composability is worth nothing, literally. The situation would be even more dramatic if you were running a public service on a **pay-per-use (PPU)** basis, locally or on a cloud.

Furthermore, the Service Repository design will be based on the requirements of the already redesigned cross-Latin-American realization of the TM Forum's resource provisioning component:

- A composition of three provisioning BPEL flows with Pan-American content, which are agnostic to any CTU's **geographical unit (GU)** while handling an order header and body.
- The individual handler of an order line, although it is not agnostic and caters to specific parts of the operating countries.
- Other conditions must be taken into account. Every order line could spawn several subsequent processes depending on the mentioned conditions. These conditions are met by establishing a universal agnostic composition controller in the form of synchronous (OSB) and asynchronous (SCA) Service Brokers, as explained in the previous two chapters. They are the main service consumers of the Inventory Endpoint. They decouple the Service Inventory from other services and provide the lookup capability for all sorts of service artifacts.

Generally, we can expect three kinds of compositions or individual services to call the Inventory Endpoint:

- Compositions that are completely generic and potentially reusable on every geographic unit of operation. The roles can be Controllers, Subcontrollers, Initiators, and Composition members.
- Services that have a generic structure but different Endpoints or Endpoint particulars. The roles can be Controllers, Subcontrollers, and Mediators.
- Processes/services that are non-agnostic and GU-specific.

Regarding the first two, if we follow the SOA design rules, we will also have to make them domain-agnostic so that they are not just bound to the order-provisioning domain, but are much wider than that.

Our **Assumptions** for the technical realization are obvious, as follows:

1. The SCA (BPEL) will be the main orchestration platform for long-running processes (already assumed and implemented).
2. ESB will be the main Service Broker for short-running stateless operations (done in the previous chapter).
3. Our design must be vendor-independent such that we can replace any component—stateless or common—by using custom build (Java) or other vendor components from other vendors.

The last assumption is extremely important for the Service Inventory and is the focal point in the entire service infrastructure—any inaccuracy in the implementation of the Service Metadata Centralization pattern will cause all other components to be severely affected, Discoverability to be compromised, and composability to become questionable.

Service metadata for Agnostic Composition Controller

In the previous CTU composition controllers' refactoring examples, two teams were tasked to maintain, lookup, retrieve, and inject service metadata into a service message in a form suitable for processing by at least two core players of our service composition:

- Composition Controller and Subcontroller (sync and async Service Brokers (SB))
- Adapter Factory (AF; also known as Generic Adapter (GA))

Both teams followed the same classification approach as follows:

- A single service's invocation is a task
- Each invocation has a number of particulars, expressed as parameters, bound to the task
- Composition is a sequence of the tasks that logically represents the business process or its completed part (scope is defined in BPEL terms)

Tasks can be grouped according to the business logic, but both teams must abide by the following rules:

- Do not make the routing slip (the task's execution plan) too complex
- If we have several logical groups (>3) or too many tasks in a group (>9), then it is better to denormalize some processes again into an atomic task-orchestrated service and call it dynamically from a smaller execution plan

We are reiterating this only because we want to remind you that we are not reinventing the BPEL. We are purely centralizing the service metadata and applying it to the service message when necessary. So, the general structure is obvious: metadata elements should cover all composition levels (process, individual composition, task, and task parameters), but how different can the understanding of metadata be? The composition's Execution Plan for the custom synchronous Service Broker from the previous chapter is easier as we do not have to focus on the actions on the return path. So, let's look at it now. The following is just a single task node:

```
<TPProcess>
  <TradingPartnerProcessList EdiProcessId="100" EdiProcessCommType="6"
              BusinessEventName="com.ctu.oebs.purchaseorder.insert"
              Rule="1" RuleCondition="ALL" Source="OEBS"
              ObjectClassName="PurchaseOrder"
              ObjectName="PurchaseOrderID" ObjectAction="Insert"
              EdiProcessAckID="2" EdiProcessAckComm="6"
              EdiProcessReportLevel="3">
  <TradingPartners>
  <Senders>
 <Sender>
      <TPId>1</TPId>
      <TPCode>CTU</TPCode>
      <XDIBoxId>1</XDIBoxId>
         …
      <MailBox/>
    </Sender>
  </Senders>
  <Receivers>
    <Receiver>
      <TPId>2</TPId>
      <TPCode>IBX</TPCode>
      <XDIBoxId>2</XDIBoxId>
    </Receiver>
  </Receivers>
  </TradingPartners>
<TPProcessDescription>
      <TPProcessTask TaskId="1" ProcessTaskDescription="Receive"
            ProcessTaskId="1" ProcessTaskName="Receive"
            ProcessTaskOrder="1"
            TaskActionName="Receive"
            TaskDescription="Receive and Detect" TaskCommType="NA"
            TaskActive="Y" TaskParametersCount="1"
            TaskEngine="XDIMB.Pojo" ProcDefId="100"
            ProcDefDescription="TDC IBX PO" ReceiverEndpoint=""
            ReceiverEndpointHost="//somehost"
            ReceiverEndpointPort="3201"
            ReceiverEndpointUserName="" ReceiverEndpointPrivateKey=""
            SenderEndpoint="/Box/System/Out/" MsgId="83"
            MsgFileName="PurchaseOrder" MsgFileExt="xml"
            MsgLogLevel="3" MsgDescription="PurchaseOrder"
            MsgConsolidatedFlag="N" MsgGroup="Order" MsgCode=""
            MsgHeaderVersion="" SchemaFilename=""
            StylesheetLocation=""
            FieldSeparator="" ElementSeparator="" TermSeparator=""
            MsgHeader="" MsgFooter=""/>
      <TPProcessTask TaskId="2" ProcessTaskDescription="Translate"
    …
</TPProcessDescription>
</TPProcess>
```

Horrible, isn't it? However, we will not discuss the pros and cons of putting task parameters into the `XMLNode` attributes or present them as elements to parse/traverse simplicity. Also, incompliance with the XML elements' naming standard is not a huge crime, although it should be avoided. What you clearly see from this example is that the attempt to devise an all-occasion task with all the possible parameters leads to a mess. We got everything here: a path to transformation XSLTs (transformation task), delimiters and terminators for the EDI file translation (translation task), service engine descriptors for non-WS tasks, a process-logging level flag, a task-logging level flag, and so on. Truly, when you start expanding the parameter list, it's really hard to stop.

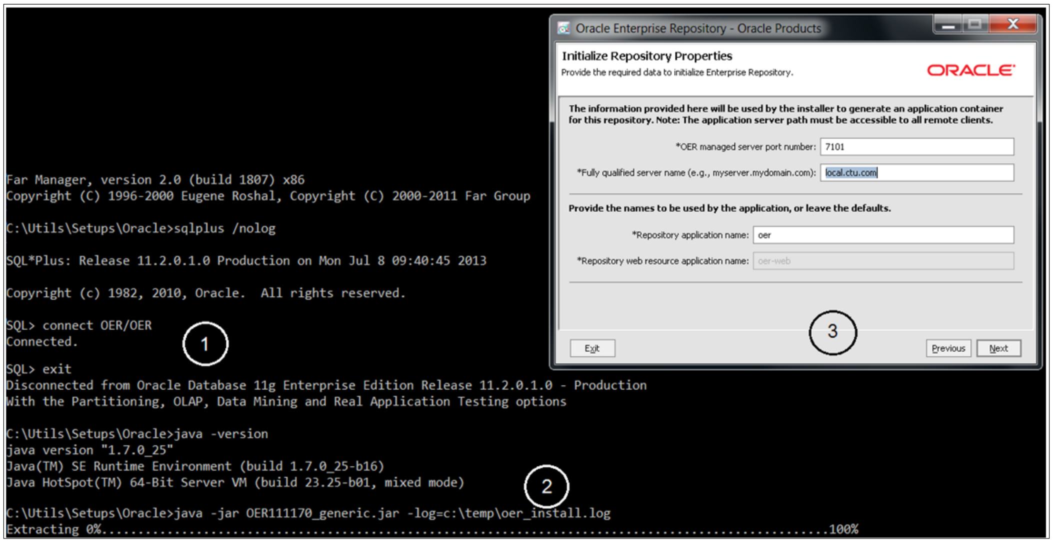
We need some guidance here to find a way to classify our major SOA artifacts and service particulars and rationalize their storage and extraction. Designing it as vendor-independent will eventually show us how this can be deployed on Oracle tools: Repository and Registry. Therefore, let's prepare the playground by installing Repository first. The taxonomy provided by Registry is UDDI based, so we will return to it a bit later in this chapter.

Exploring the Oracle Repository's taxonomy

The Repository installation is straightforward (three simple steps). The installation instructions are easily found on the Oracle site. Nevertheless, we should mention a few things here. First, it's DB based, which is a positive feature because we want to change from the initial file-based design (although MDS can also be DB-based); however, this also means that close DBA attention will be necessary for this critical component (RMAN, RAC planning, and so on). You will create three tablespaces for data, indexes, and CLOB; please place them wisely. The application is a JAR file and will be deployed on your WLS, so the app server is the second prerequisite for the actual installation. Of course, OER can be installed not just on WLS (please see the complete list in the installation guide). Actually, the same is valid for DB as well. During the installation, please give special attention to the correctness of the value you put in the **Fully qualified server name** field (in local test mode, it's just a localhost). A typical `.jar` installation has a script screen identical to that in the following figure.

By the way, if something went wrong and you had to restart the installation, please do not forget to recreate the tablespaces (the installation will not continue with the tables in place) and purge the OER instance from the `Oracle_Home` inventory. It would be logical to create a new WSL domain (`oer_domain`) separated from `soa_` and `osb_`. Do not forget to add an admin server with the console as a feature. Start it in the regular fashion (first is the Node Manager and then WLS, followed by the WLS console, `start oer_server1`) and log in to `http://<localhost>:7101/oer/` using the default credentials. We will not tell you which ones.

At the time of writing this, it was admin/admin. Yes, have fun with Oracle password consistency. You will be prompted to change the password anyway. For further references, please go to the folder `[Oracle_Home] \repository[xxx]\core\tools\solutions`; the folder has some utilities essential for further SOA Governance:



```
Far Manager, version 2.0 (build 1807) x86
Copyright (C) 1996-2000 Eugene Roshal, Copyright (C) 2000-2011 Far Group

C:\Utils\Setups\Oracle>sqlplus /nolog

SQL*Plus: Release 11.2.0.1.0 Production on Mon Jul 8 09:40:45 2013

Copyright (c) 1982, 2010, Oracle.  All rights reserved.

SQL> connect OER/OER
Connected.
SQL> exit
Disconnected from Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 - Production
With the Partitioning, OLAP, Data Mining and Real Application Testing options

C:\Utils\Setups\Oracle>java -version
java version "1.7.0_25"
Java(TM) SE Runtime Environment (build 1.7.0_25-b16)
Java HotSpot(TM) 64-Bit Server VM (build 23.25-b01, mixed mode)

C:\Utils\Setups\Oracle>java -jar OER111170_generic.jar -log=c:\temp\oer_install.log
Extracting 0%...........................................................................................100%
```

Oracle Enterprise Repository - Oracle Products

Initialize Repository Properties

Provide the required data to initialize Enterprise Repository.

The information provided here will be used by the installer to generate an application container for this repository. Note: The application server path must be accessible to all remote clients.

*OER managed server port number: 7101

*Fully qualified server name (e.g., myserver.mydomain.com): local.ctu.com

Provide the names to be used by the application, or leave the defaults.

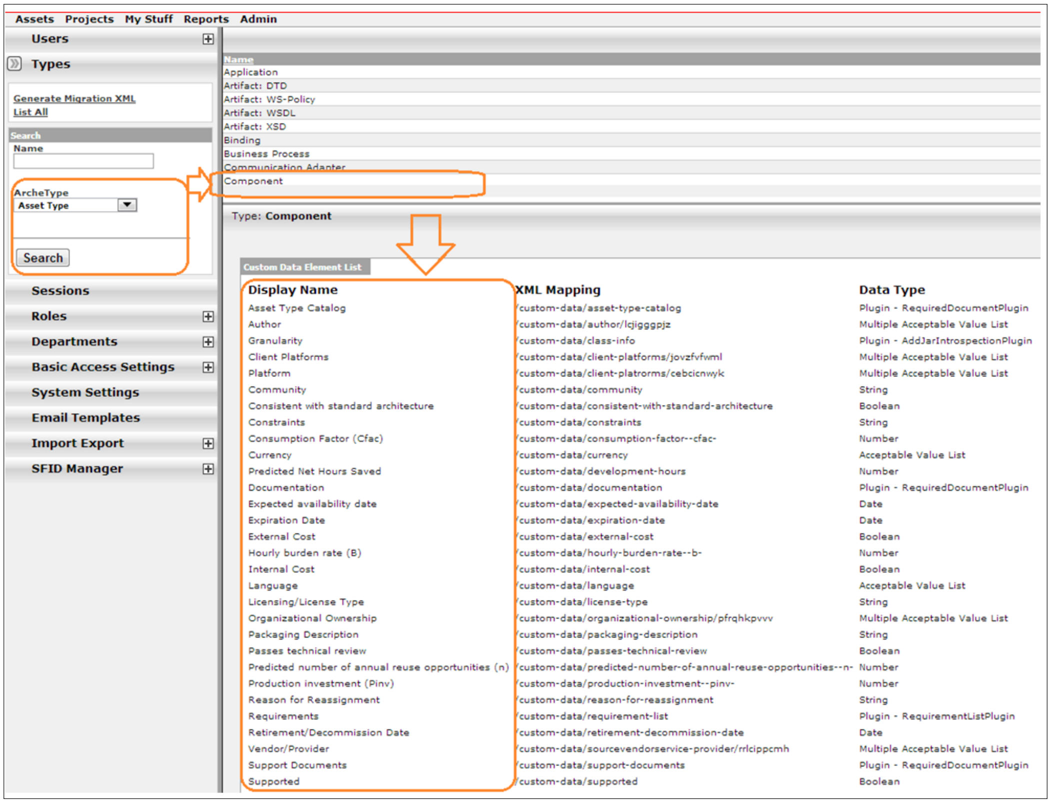
*Repository application name: oer

*Repository web resource application name: oer-web

Oracle Enterprise Repository installation

Now let's check what we have got right out of the box in terms of taxonomy and SAO mindset organization. Since any Repository is initially organized around Asset Management as Asset is the main atomic unit of operational handling, search for **All Assets of All Types** and take a look at the list. There are several samples of varying types—from the adapters to the frameworks. You can explore relations between assets by choosing **Asset** and clicking on the left button below the **Assets** list. Technology adapters will be presented with no relations, and application adapters will be linked to the isolated application (such as Siebel). Guess what you will see once you have selected the MVC pattern?

You will see an Asset registration and the means to visualize the dependency, but we still need to maintain the categories, types, and relations; some SOA guidance would be useful here. Click on the **Admin** menu option and search for **Asset Type** to see what kinds are available (see the following figure):



Name
Application
Artifact: DTD
Artifact: WS-Policy
Artifact: WSDL
Artifact: XSD
Binding
Business Process
Communication Adapter
Component

Type: **Component**

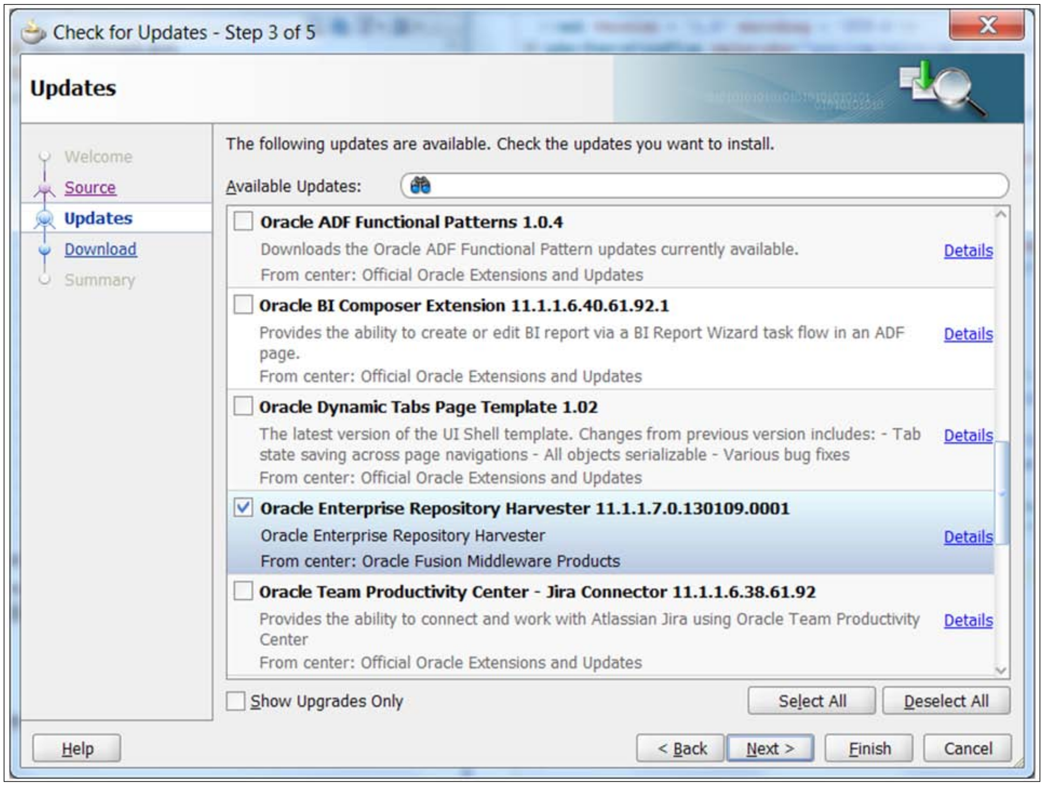
Custom Data Element List

Display Name	XML Mapping	Data Type
Asset Type Catalog	/custom-data/asset-type-catalog	Plugin - RequiredDocumentPlugin
Author	/custom-data/author/lcjigggpjz	Multiple Acceptable Value List
Granularity	/custom-data/class-info	Plugin - AddJarIntrospectionPlugin
Client Platforms	/custom-data/client-platforms/jovzfvfwml	Multiple Acceptable Value List
Platform	/custom-data/client-platrorms/cebcicnwyk	Multiple Acceptable Value List
Community	/custom-data/community	String
Consistent with standard architecture	/custom-data/consistent-with-standard-architecture	Boolean
Constraints	/custom-data/constraints	String
Consumption Factor (Cfac)	/custom-data/consumption-factor--cfac-	Number
Currency	/custom-data/currency	Acceptable Value List
Predicted Net Hours Saved	/custom-data/development-hours	Number
Documentation	/custom-data/documentation	Plugin - RequiredDocumentPlugin
Expected availability date	/custom-data/expected-availability-date	Date
Expiration Date	/custom-data/expiration-date	Date
External Cost	/custom-data/external-cost	Boolean
Hourly burden rate (B)	/custom-data/hourly-burden-rate--b-	Number
Internal Cost	/custom-data/internal-cost	Boolean
Language	/custom-data/language	Acceptable Value List
Licensing/License Type	/custom-data/license-type	String
Organizational Ownership	/custom-data/organizational-ownership/pfrqhkpvvv	Multiple Acceptable Value List
Packaging Description	/custom-data/packaging-description	String
Passes technical review	/custom-data/passes-technical-review	Boolean
Predicted number of annual reuse opportunities (n)	/custom-data/predicted-number-of-annual-reuse-opportunities--n-	Number
Production investment (Pinv)	/custom-data/production-investment--pinv-	Number
Reason for Reassignment	/custom-data/reason-for-reassignment	String
Requirements	/custom-data/requirement-list	Plugin - RequirementListPlugin
Retirement/Decommission Date	/custom-data/retirement-decommission-date	Date
Vendor/Provider	/custom-data/sourcevendorservice-provider/rrlcippcmh	Multiple Acceptable Value List
Support Documents	/custom-data/support-documents	Plugin - RequiredDocumentPlugin
Supported	/custom-data/supported	Boolean

OER Asset Management

The more interesting type is **Component**; it is possibly one of the building blocks of our SOA, but what we have got in the default data element list for this type is not exactly encouraging. **Hourly burden rate**, **License Type**, and all other elements are certainly useful for runtime auditing and project management organization (they are also based on Enterprise Repository; we will come to that later in this section). However, we see only a couple of elements we could use for our runtime discovery in the Composition Controller.

No problem; no one expects a complete runtime-ready taxonomy model immediately out of the box. We have been following the SOA methodology all the way in the last two chapters, so we can try to export all our previously built artifacts and see how they will fit the existing taxonomy skeleton and what kind of categorization and types will be added to the classification.



The following updates are available. Check the updates you want to install.

□ **Oracle ADF Functional Patterns 1.0.4**
Downloads the Oracle ADF Functional Pattern updates currently available.
From center: Official Oracle Extensions and Updates

□ **Oracle BI Composer Extension 11.1.1.6.40.61.92.1**
Provides the ability to create or edit BI report via a BI Report Wizard task flow in an ADF page.
From center: Official Oracle Extensions and Updates

□ **Oracle Dynamic Tabs Page Template 1.02**
The latest version of the UI Shell template. Changes from previous version includes: - Tab state saving across page navigations - All objects serializable - Various bug fixes
From center: Official Oracle Extensions and Updates

☑ **Oracle Enterprise Repository Harvester 11.1.1.7.0.130109.0001**
Oracle Enterprise Repository Harvester
From center: Oracle Fusion Middleware Products

□ **Oracle Team Productivity Center - Jira Connector 11.1.1.6.38.61.92**
Provides the ability to connect and work with Atlassian Jira using Oracle Team Productivity Center
From center: Official Oracle Extensions and Updates

□ Show Upgrades Only

Before we proceed with the OER console, please complete one more install; this one will now be on your JDeveloper. Go to the update center and install the Enterprise Repository Harvester (see the previous figure). You will have to restart JDeveloper after completing the updates. Actually, the harvesting can be seen as an Ant task in JDev (we have other options as well), so we need to perform three more steps (for JDeveloper11*g*):

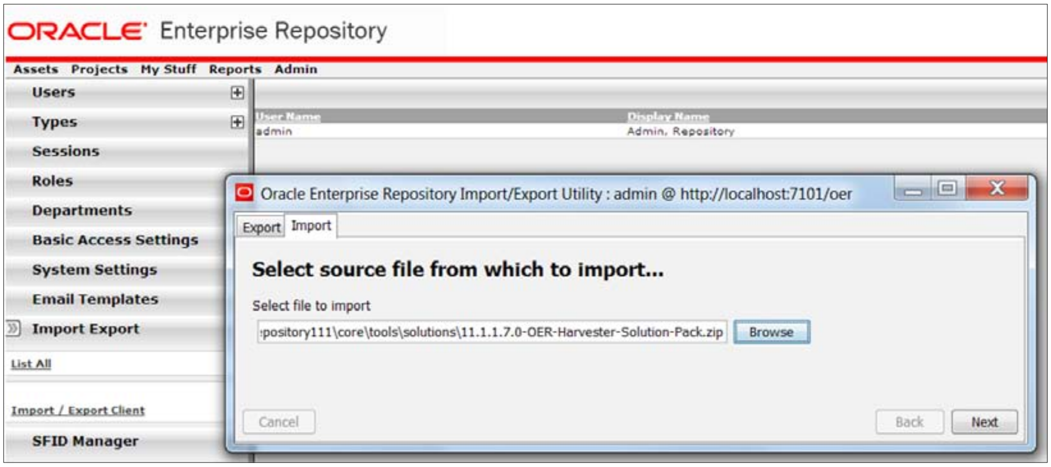
1. Create backups of all your JDeveloper config files.
2. Go to the `[JDev_home]/harvester` folder created during the first step and replace all the occurrences of `C:/oracle/middleware/jdev_5361/jdeveloper` in `tools11g.xml` with your current JDev path.

3. Merge the content of your `tools11g.xml` with `product-preferences.xml` in your `<user>/AppData/…/JDeveloper/<system_version>/o.jdeveloper` (mine is in `Users\spopov\AppData\Roaming\JDeveloper\system11.1.1.6.38.61.92\o.jdeveloper`). Copy the content if the XML hash node `oracle.ideimpl.externaltools.ExternalToolList` does not exist. Otherwise, simply replace it.
4. Check the OER connection detail in `[JDev_home]/harvester / HarvesterSettings.xml`. You can also establish a connection by navigating to **JDeveloper File** | **New** | **Connections** | **OER Connection**, but after that, encrypting the password would be a good idea. Just run `encrypt.bat` from the same folder.

The integration of OER and JDeveloper is one of the really effective OFM features, and we are sure that you will enjoy it. However, balancing it with one friendly warning would be in order.

> To browse the Repository, any browser will do, but for administrative and maintenance features in **Asset Editor**, please use Internet Explorer with the latest JDK (see the installation screen); otherwise, there is a high probability that you will get the `Premature end of file` error. Quite annoying, actually.

Now, since we are forewarned, we can go back to the OER console and complete the second part of the harvester's installation. We need to import the harvester solution pack with all the possible service taxonomy types and classifications. Click on **Admin** and then **Import/Export** and import the pack as shown in the following screenshot:



User Name	Display Name
admin	Admin, Repository

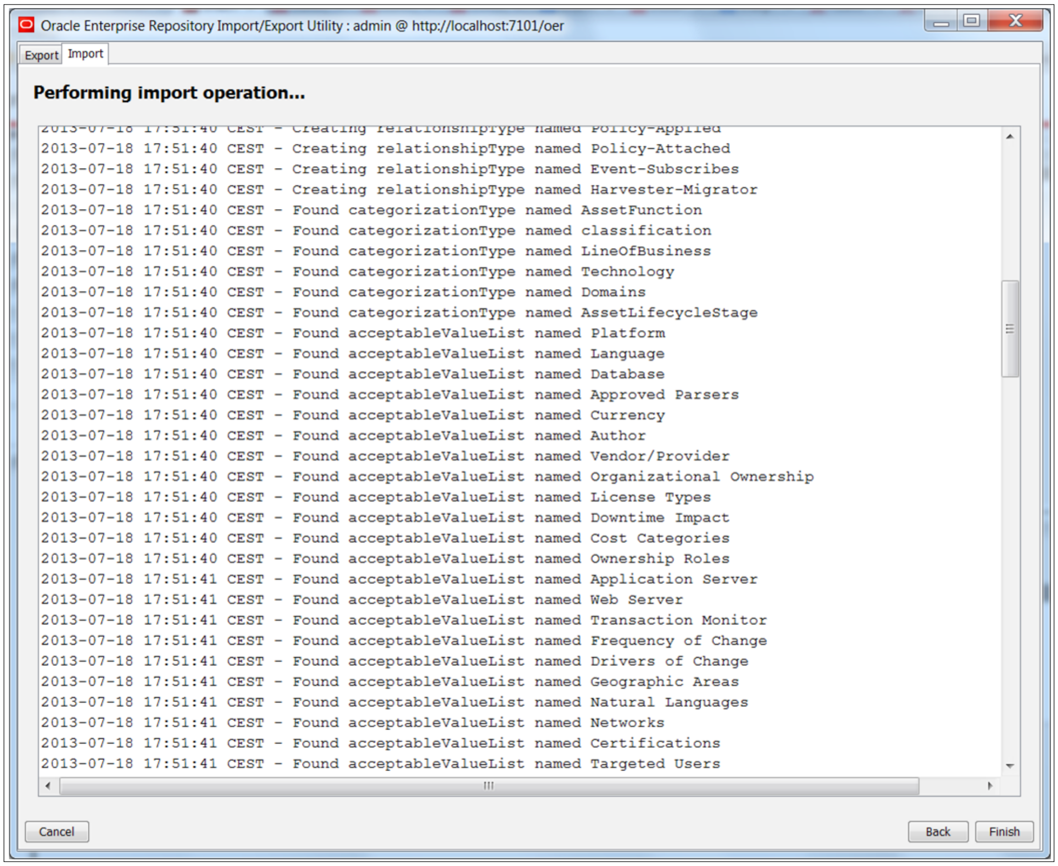
Oracle Enterprise Repository Import/Export Utility : admin @ http://localhost:7101/oer

Select source file from which to import...

Select file to import

:pository111\core\tools\solutions\11.1.1.7.0-OER-Harvester-Solution-Pack.zip

When the import initiates, you will immediately notice the number of asset and relation types being imported (please see the following figure):


Oracle Enterprise Repository Import/Export Utility : admin @ http://localhost:7101/oer
Export
Import
Cancel
Back
Finish

Performing import operation...

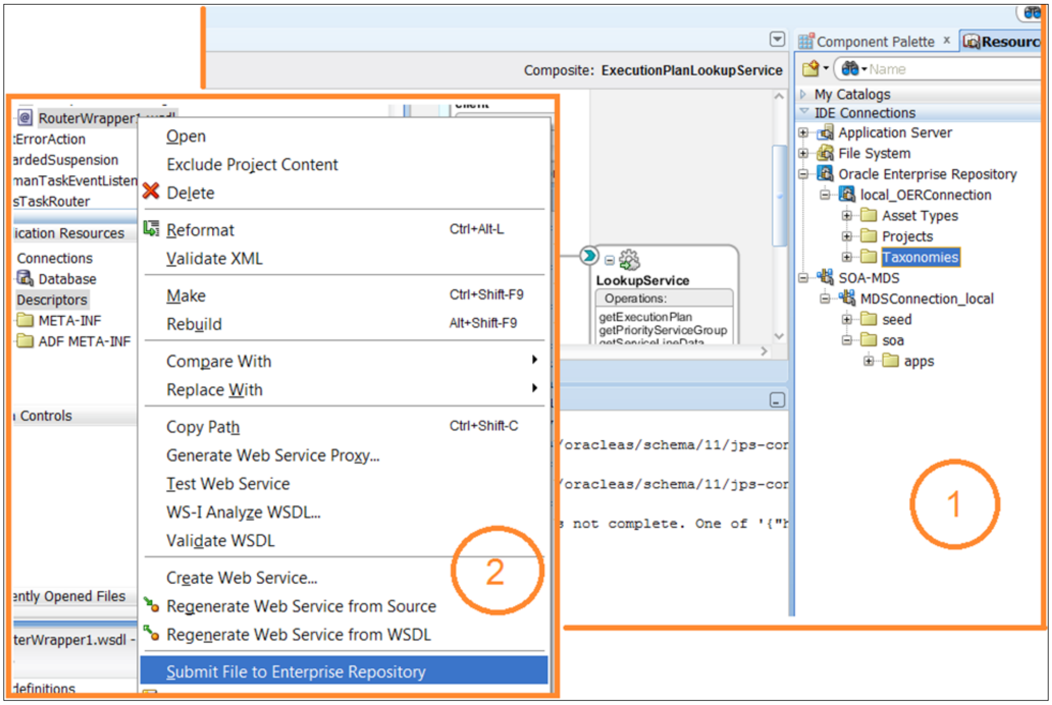
```
2013-07-18 17:51:40 CEST - Creating relationshipType named Policy-Applied
2013-07-18 17:51:40 CEST - Creating relationshipType named Policy-Attached
2013-07-18 17:51:40 CEST - Creating relationshipType named Event-Subscribes
2013-07-18 17:51:40 CEST - Creating relationshipType named Harvester-Migrator
2013-07-18 17:51:40 CEST - Found categorizationType named AssetFunction
2013-07-18 17:51:40 CEST - Found categorizationType named classification
2013-07-18 17:51:40 CEST - Found categorizationType named LineOfBusiness
2013-07-18 17:51:40 CEST - Found categorizationType named Technology
2013-07-18 17:51:40 CEST - Found categorizationType named Domains
2013-07-18 17:51:40 CEST - Found categorizationType named AssetLifecycleStage
2013-07-18 17:51:40 CEST - Found acceptableValueList named Platform
2013-07-18 17:51:40 CEST - Found acceptableValueList named Language
2013-07-18 17:51:40 CEST - Found acceptableValueList named Database
2013-07-18 17:51:40 CEST - Found acceptableValueList named Approved Parsers
2013-07-18 17:51:40 CEST - Found acceptableValueList named Currency
2013-07-18 17:51:40 CEST - Found acceptableValueList named Author
2013-07-18 17:51:40 CEST - Found acceptableValueList named Vendor/Provider
2013-07-18 17:51:40 CEST - Found acceptableValueList named Organizational Ownership
2013-07-18 17:51:40 CEST - Found acceptableValueList named License Types
2013-07-18 17:51:40 CEST - Found acceptableValueList named Downtime Impact
2013-07-18 17:51:40 CEST - Found acceptableValueList named Cost Categories
2013-07-18 17:51:40 CEST - Found acceptableValueList named Ownership Roles
2013-07-18 17:51:41 CEST - Found acceptableValueList named Application Server
2013-07-18 17:51:41 CEST - Found acceptableValueList named Web Server
2013-07-18 17:51:41 CEST - Found acceptableValueList named Transaction Monitor
2013-07-18 17:51:41 CEST - Found acceptableValueList named Frequency of Change
2013-07-18 17:51:41 CEST - Found acceptableValueList named Drivers of Change
2013-07-18 17:51:41 CEST - Found acceptableValueList named Geographic Areas
2013-07-18 17:51:41 CEST - Found acceptableValueList named Natural Languages
2013-07-18 17:51:41 CEST - Found acceptableValueList named Networks
2013-07-18 17:51:41 CEST - Found acceptableValueList named Certifications
2013-07-18 17:51:41 CEST - Found acceptableValueList named Targeted Users
```

After completing the import, you can go to the **Admin** page and check the assets types again. We have got plenty of new artifacts and business processes, but the elements for the existing ones haven't changed; check this using the same **Component** asset. You will always see the recent changes of the **Assets** structure or their usage on the first page. How to change the types and elements for the assets including the element relations we will see a bit later, when we get a better understanding of what we want. At the moment, all these assets will be needed for harvesting, that is, uploading previously-built SOA components into a Repository.

Now back to JDeveloper. You will be able to see the following two options by right-clicking, as shown in the figure that follows:

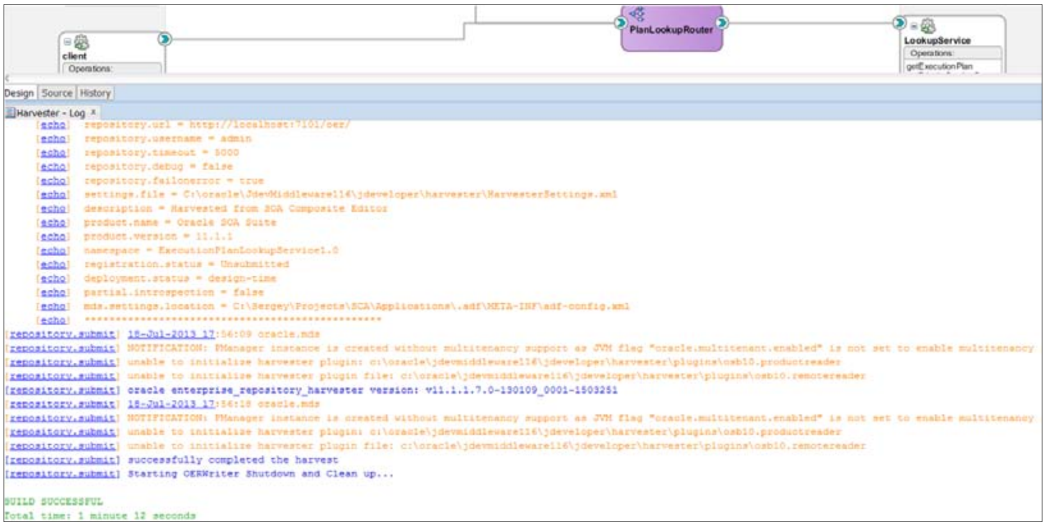
- **Submit Project to Enterprise Repository**
- **Submit File to Enterprise Repository**

The first one can be applied to the root (your project's name) and the second is suitable for artifacts such as WSDL and XSD (**2**). Before you continue, please check the connectivity to the OER (**1**). It's right above the MDS connection we used in the previous chapters:



Submitting artifacts to Oracle Enterprise Repository

We will harvest the entire **ExecutionPlanLookupService** project; please see the Ant log in the following screenshot:

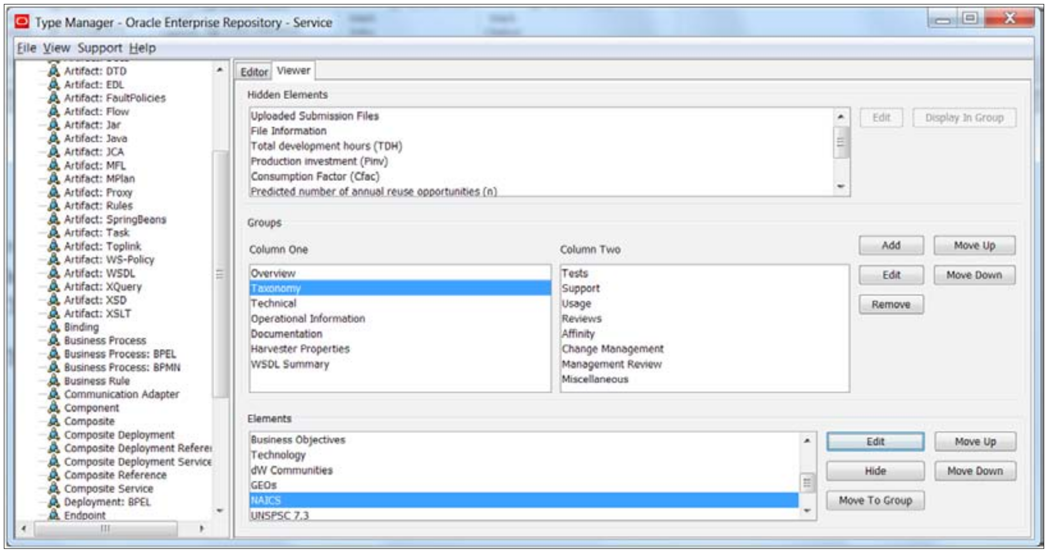


```
     [echo]  repository.url = http://localhost:7101/oer/
     [echo]  repository.username = admin
     [echo]  repository.timeout = 5000
     [echo]  repository.debug = false
     [echo]  repository.failonerror = true
     [echo]  settings.file = C:\oracle\JdevMiddleware116\jdeveloper\harvester\HarvesterSettings.xml
     [echo]  description = Harvested from SOA Composite Editor
     [echo]  product.name = Oracle SOA Suite
     [echo]  product.version = 11.1.1
     [echo]  namespace = ExecutionPlanLookupService1.0
     [echo]  registration.status = Unsubmitted
     [echo]  deployment.status = design-time
     [echo]  partial.introspection = false
     [echo]  mds.settings.location = C:\Sergey\Projects\SCA\Applications\.adf\META-INF\adf-config.xml
     [echo]  ************************************************
[repository.submit] 18-Jul-2013 17:56:09 oracle.mds
[repository.submit] NOTIFICATION: PManager instance is created without multitenancy support as JVM flag "oracle.multitenant.enabled" is not set to enable multitenancy
[repository.submit] unable to initialize harvester plugin: c:\oracle\jdevmiddleware116\jdeveloper\harvester\plugins\osb10.productreader
[repository.submit] unable to initialize harvester plugin file: c:\oracle\jdevmiddleware116\jdeveloper\harvester\plugins\osb10.remotereader
[repository.submit] oracle enterprise_repository_harvester version: v11.1.1.7.0-130109_0001-1503251
[repository.submit] 18-Jul-2013 17:56:18 oracle.mds
[repository.submit] NOTIFICATION: PManager instance is created without multitenancy support as JVM flag "oracle.multitenant.enabled" is not set to enable multitenancy
[repository.submit] unable to initialize harvester plugin: c:\oracle\jdevmiddleware116\jdeveloper\harvester\plugins\osb10.productreader
[repository.submit] unable to initialize harvester plugin file: c:\oracle\jdevmiddleware116\jdeveloper\harvester\plugins\osb10.remotereader
[repository.submit] successfully completed the harvest
[repository.submit] Starting OERWriter Shutdown and Clean up...

BUILD SUCCESSFUL
Total time: 1 minute 12 seconds
```

Harvesting SOA artifacts from the existing project

Although we successfully completed the harvest, some messages should attract your attention. We extracted all project objects and artifacts from the SOA Suite, but we have a lot references to OSB. So what about them? Oracle provided an individual harvester with OSB, which we can use for ESB object retrospection. We already have our OER prepared and the OSB configuration steps are similar to what we did in JDeveloper: modify `HarvesterSettings.xml` in the `[MIDDLEWARE_HOME]\<Oracle_OSB1>\harvester` folder, and set the correct OER connection parameters and path to our OSB project's export `sbconfig.jar` file. After that, encrypt the password. Now, to perform harvesting, run `setenv.bat` from the same folder and then `osb11g-harvest.bat`. You will see the OSB assets in the OER console by searching for the name of our project (for instance, we used Service Broker in the previous chapter). The Oracle documentation provides a complete reference to harvest in different runtime and design-time environments (`http://docs.oracle.com/cd/E23943_01/admin.1111/e16580/harvest.htm`). We suggest that you look closely at the relations between different assets harvested on different platforms.

So far so good. We have established a complete platform for bottom-up project development. However, browsing through the **Assets Types**, all we can see are collections of name-value pairs with basic relations, covering the generic needs of any Enterprise; these collections are not exactly SOA-oriented.



OER Type Manager's Taxonomies

The taxonomy of elements, presented for the service asset in the earlier figure, is too generic. Moreover, although it is suitable for dynamic invocation (Technical/UDDI registry), it still does not comply with all the requirements we expressed earlier for implementation using the Agnostic Composition Controller's capabilities.

Taxonomies such as **NAICS** and **UNSPSC**, mentioned in the preceding screenshot, are purely business-oriented and are not suitable for SOA as an architectural approach. To find the proper classification for service attributes, we should look at the public SOA taxonomies and ontologies.

Open standards for the SOA taxonomy

Generally, we have two sets of public standards: Repository and Registry.
Open Group came up with a wide range of standardization initiatives, of which two can be very beneficial to us:

- The SOA Governance Technical Standard (including the SOA Governance Reference Model (SGRM))
- The SOA Ontology Technical Standard

The Governance Reference model covers all aspects of the Enterprise SOA lifecycle and, naturally, all ontology features. One of the aspects covered is service harvesting, which is the third principle after service reuse and service description. We learned that it will give us little without the first two principles. Therefore, service metadata is the main element of the service description to support service reuse, one of the main SOA principles to maintain. There is a plethora of material on the SGRM standard for organizing projects, implementing change requests, and establishing and monitoring KPIs. However, in the context of this chapter, references to ontology are most important to us. A detailed presentation of the current version of Open Group's service ontology is provided on the organization's website, and we recommend that you take a close look at the two main diagrams on the introduction page, the ER diagram and hierarchy drawing—the entire ontology is graphically presented in these two forms. They are definitely something we could use for our OER taxonomy design, so review the diagrams intently.

> Remember that these public standards are constantly under development. Conduct your own Web Ontology Language (OWL) studies (`http://www.opengroup.org/soa/ontology/20101021/soa.owl`) at the moment of publication.

When studying the hierarchy, we cannot help but notice several aspects that might be crucial to the acceptance of this ontology as the basis for agnostic composition controller implementation:

- On the SGRM page, the first guiding principle is "SOA Governance must promote the alignment of business and IT". This is effective indeed and has always been one of the goals (an objective, not a principle) of SOA. Please look at *Chapter 1, SOA Ecosystem – Interconnected Principles, Patterns, and Frameworks*, again in which we discussed principles, goals, good wishes, common sense, and what separates them. Nevertheless, this statement rightfully promotes Process as the key element of any business-oriented SOA. However, this is not the case when you look at the hierarchy graphics; Process is a child node in the Composition.

Generally, a top-level Composition consists of Processes and presents the Master Composition, not the other way around. On the other hand, in the OWL schema, the Process consists of Compositions. At the same time, `ServiceComposition` is a subset of the Composition and has no direct relation to the Service itself. We agree that a Process, Composition, and `ServiceComposition` can (or cannot) be presented as a Service. However, in this case, a Service should be taken out of the hierarchy and placed separately, close to the Event.

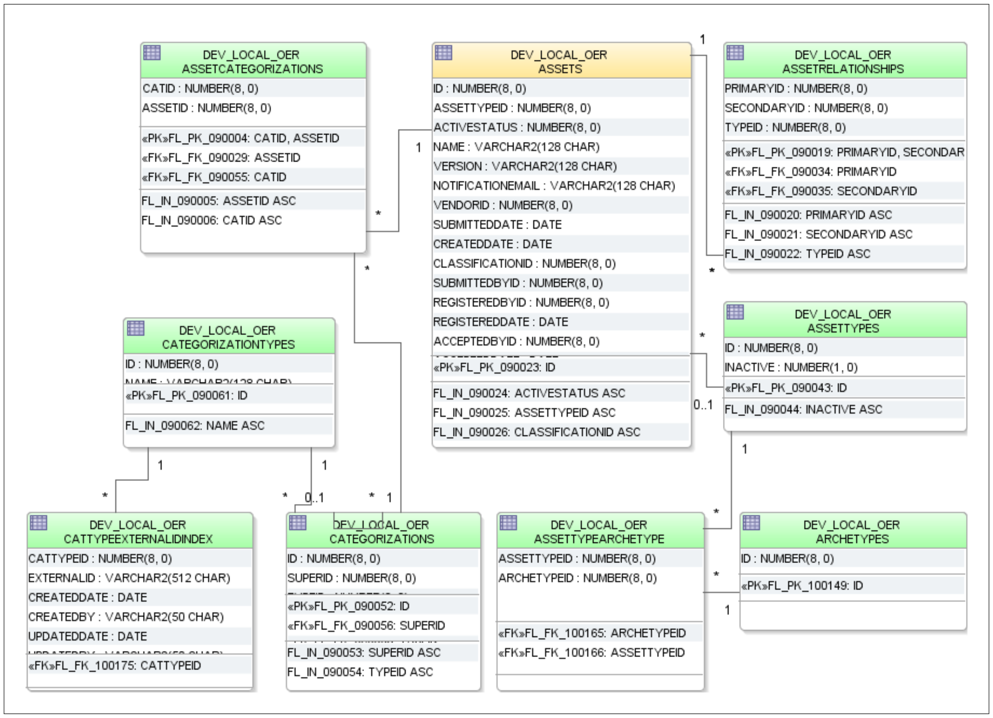
- The `ServiceContract` and `ServiceInterface` classes are separate elements of hierarchy, and this could create some confusion. Generally, Open Group describes a contract in terms of SLA, which includes parties involved in service activities and their legal obligations. `ServiceInterface` has a more technical nature and presents an RPC-like access point for message-based invocations (not a totally correct term as we have DOC-type services, not RPC-style ones; however, we hope that you get the idea). Therefore, we can expect some attributes as `Operation` or `Task` (again, in WSDL terms). Indeed, `Task` is the property of the `ServiceInterface` class. At the same time, you will find this property in the `ServiceContract` class. An example of the `Task` property is provided by OWL: "`WashWindows` is an instance of `Task`, performed by the service provider (John)."
- In addition to `Task`, both contract and interface have something called Effect. Effect represents the outcome of service interaction and holds value for the customer. Here, we have another inconsistency. If our interface is message-oriented and a message is a transportable form of an object, then the logical outcome of the service operation is a new (changed) state of the object. An object's noticeable change is an Event—"The weather has changed. Expect heavy rain in ... hours." So, what is the Event according to the Open Group ontology?
- An Event is described as an occurrence on an Element to which an Element may choose to respond. From this, we understand that an Element is, in fact, an Object. An Element is at the top of practically any hierarchy. You will find an Element at the top of the Open Group Ontology as well; only an Effect will be higher. An Event is detached and located at the bottom of the hierarchy. The logic underlying this decision is not entirely clear.
- The authors tried to avoid using the term "Object" and proposed a superclass for Element, `Thing`. What's wrong with the old "Object" is completely unclear as well.

- A Policy is defined as "a statement of direction that a human actor may intend to follow or may intend that another human actor should follow." First, when talking about SOA, we immediately exclude, during the functional decomposition phase, human operations from the composition logic, deeming them unsuitable for automation. We do not abandon them, but just have another approach for them. Thus, the policy should have a slightly broader meaning. Second, statements of directions are usually expressed through certain directives, that is, operations (tasks). These operations could be performed by units of logic without a contract or public interface (service agents, event-driven modules, *not* the services). Thus, the whole structure of the `Policy` class is rather questionable.

Despite these considerably small discrepancies with the classic OOP and generic SOA in terminology and relations ontology, we can take a lot from the presented approach and use it in our service metadata classification, optimized for dynamic and agnostic service invocation. We understand that the discussed ontology is an all-purpose model; therefore, we focus on relations suitable for practical implementation of Service Repository on traditional DBs.

Other standardization groups, excluding Open Group, have made proposals for the service metadata taxonomy. Noteworthy is the Reference Architecture Framework for the SOA (SOA-RA) initiative by OASIS (`http://docs.oasis-open.org/soa-rm/soa-ra/v1.0/csd03/soa-ra-v1.0-csd03.pdf`; please check for the latest release). At the time of writing, their draft seemed rather theoretical and the committee members were still debating on the definition of "service." When asked about the practical value of this reference model for SOA practitioners (at the fifth SOA & Cloud Symposium, where the first draft was presented), the presenter, Mr. Brown, responded that the purpose is to build a better SOA. Without a doubt it's a noble cause, and we believe that eventually it will produce something valuable for SOA practitioners. For now, taxonomy Elements such as *Willingness*, *Social Structure*, *Evidence*, *Real-World Effect*, and *Reputation* are out of the scope of the SOA patterns' implementation.

Actually, apart from the taxonomy, SOA-RA can be useful to understand the internal Oracle ER DB structure. As you already noticed, there are so many elements to maintain with so many relations in both public ontology frameworks that we couldn't avoid maintaining categorization using the name-value style. SOA-RA *Elements Common to General Description* (*Figure 14* in `http://docs.oasis-open.org/soa-rm/soa-ra/v1.0/soa-ra.pdf`) is generic and quite similar to the asset definition domain in OER (please see the following diagram):


DEV_LOCAL_OER
ASSETCATEGORIZATIONS
CATID : NUMBER(8, 0)
ASSETID : NUMBER(8, 0)
«PK»FL_PK_090004: CATID, ASSETID
«FK»FL_FK_090029: ASSETID
«FK»FL_FK_090055: CATID
FL_IN_090005: ASSETID ASC
FL_IN_090006: CATID ASC
DEV_LOCAL_OER
ASSETS
ID : NUMBER(8, 0)
ASSETTYPEID : NUMBER(8, 0)
ACTIVESTATUS : NUMBER(8, 0)
NAME : VARCHAR2(128 CHAR)
VERSION : VARCHAR2(128 CHAR)
NOTIFICATIONEMAIL : VARCHAR2(128 CHAR)
VENDORID : NUMBER(8, 0)
SUBMITTEDDATE : DATE
CREATEDDATE : DATE
CLASSIFICATIONID : NUMBER(8, 0)
SUBMITTEDBYID : NUMBER(8, 0)
REGISTEREDBYID : NUMBER(8, 0)
REGISTEREDDATE : DATE
ACCEPTEDBYID : NUMBER(8, 0)
«PK»FL_PK_090023: ID
FL_IN_090024: ACTIVESTATUS ASC
FL_IN_090025: ASSETTYPEID ASC
FL_IN_090026: CLASSIFICATIONID ASC
DEV_LOCAL_OER
ASSETRELATIONSHIPS
PRIMARYID : NUMBER(8, 0)
SECONDARYID : NUMBER(8, 0)
TYPEID : NUMBER(8, 0)
«PK»FL_PK_090019: PRIMARYID, SECONDAR
«FK»FL_FK_090034: PRIMARYID
«FK»FL_FK_090035: SECONDARYID
FL_IN_090020: PRIMARYID ASC
FL_IN_090021: SECONDARYID ASC
FL_IN_090022: TYPEID ASC
DEV_LOCAL_OER
CATEGORIZATIONTYPES
ID : NUMBER(8, 0)
«PK»FL_PK_090061: ID
FL_IN_090062: NAME ASC
DEV_LOCAL_OER
ASSETTYPES
ID : NUMBER(8, 0)
INACTIVE : NUMBER(1, 0)
«PK»FL_PK_090043: ID
FL_IN_090044: INACTIVE ASC
DEV_LOCAL_OER
CATTYPEEXTERNALIDINDEX
CATTYPEID : NUMBER(8, 0)
EXTERNALID : VARCHAR2(512 CHAR)
CREATEDDATE : DATE
CREATEDBY : VARCHAR2(50 CHAR)
UPDATEDDATE : DATE
«FK»FL_FK_100175: CATTYPEID
DEV_LOCAL_OER
CATEGORIZATIONS
ID : NUMBER(8, 0)
SUPERID : NUMBER(8, 0)
«PK»FL_PK_090052: ID
«FK»FL_FK_090056: SUPERID
FL_IN_090053: SUPERID ASC
FL_IN_090054: TYPEID ASC
DEV_LOCAL_OER
ASSETTYPEARCHETYPE
ASSETTYPEID : NUMBER(8, 0)
ARCHETYPEID : NUMBER(8, 0)
«FK»FL_FK_100165: ARCHETYPEID
«FK»FL_FK_100166: ASSETTYPEID
DEV_LOCAL_OER
ARCHETYPES
ID : NUMBER(8, 0)
«PK»FL_PK_100149: ID
1
*
1
*
*
*
0..1
1
1
1
*
*
0..1
*
1
*
*
1
1

You can check it yourself, but just remember that the previously presented relational model from the real SR database is simplified for brevity. Nevertheless, using the model, you can easily construct the `insert` statement to manually insert assets. In addition to asset relations being constructed this way, the metadata domain also has a similar structure.

> Another noteworthy, and rather important, point with regard to this type of DB model is that, although quite simple in design and, probably, in the initial value population, the key-value pair DB approach will most certainly turn into a nightmare from a maintenance standpoint later. Key-value pairs in relational databases are a constant headache for DBAs, SOA process owners, service custodians, and so on. Yes, the Oracle ER interface around this previous model will solve this problem gracefully, but the main issue will persist—the performance bottlenecks of your SQL statements during the runtime discovery on your Mediators and Service Brokers. And don't think of turning to the Big Data NoSQL model for this kind of metadata; this is not the case by any means. Quite soon, you will see that the taxonomy can be really lightweight and compatible with a relatively simple relational model.

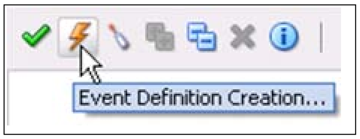
In *Chapter 1*, *SOA Ecosystem – Interconnected Principles, Patterns, and Frameworks*, we mentioned another standard published by the HL7 group, called **Service-Aware Interoperability Framework-Canonical Definition (SAIF CD)**. We kept this framework for the final discussion as we see it as an optimal model for lightweight repository implementation. Actually, it is not a single framework, it is an entire collection, covering Service Data Modeling, Governance, Enterprise Consistency, Conformity, and several others. Some of the frameworks (Governance) are completely based on the SOA concepts gathered in Thomas Erl's books. We will not repeat the concepts discussed in *Chapter 1*, *SOA Ecosystem – Interconnected Principles, Patterns, and Frameworks*. One framework is of particular value to us: the Behavioral framework; this provides the language necessary to explicitly and unambiguously define dynamic semantics used to specify the behavior of enterprise objects involved in shared purpose scenarios.

This is the core of the service metadata definition and classification, and it's attached from three directions:

- **Contract semantics**: In contract semantics, we can see only one unusual term, community, which represents the collection of interoperable objects aggregated by their business purpose or other similarities. The synonyms would be domain or group, but that's not as important. What's important is all relations between services, its possible roles and policies, and its permissions, prohibitions, and obligations are clearly and elegantly defined.
- **Operation-specific semantics**: This is even simpler and more straightforward, although the concept of an operation's pre- and post-conditions can be more clearly defined through policies (conditions apply to `ObjectContext`) or other operations performed before or after.
- **Process semantics**: This is probably the most complex in this framework, but we see a lot of similarities with our adoption of this concept, expressed by the execution plan object and Service Broker. The approach to the organization-specific implementation of SAIF-CD is basically derived from the SAIF Implementation Guide (IG).

Relations between these semantics and our implementation is presented in the following table:

SAIF BF process semantics	SAIF description	Oracle Composition framework
Process	Collection of invocations or operations	Atomic task-orchestrated service (BPEL/SCA) or individual execution plan
Flow elements	Sequence of steps in a process	BPEL Sequence or EP task group

SAIF BF process semantics	SAIF description	Oracle Composition framework
Activity	Service operation	EP individual task:
Event	Trigger	• Event is an element in Message Header for the Composition Controller • Event is described in the `.edl` file in JDeveloper (see screenshot below), where the SCA Mediator acts as a subscriber: 
Sequence flow	Ordered sequence of actions	Execution plan
Gateway	Control element that performs branching, forking, merging, and joining	Mediator in the SCA Service Broker (*Chapter 3, Building the Core – Enterprise Business Flows*) Adapter factory with a generic adapter in the OSB Service Broker (*Chapter 4, From Traditional Integration to Composition – Enterprise Business Services*)

As we can see from this reference table, SAIF-CD is pretty close to our understanding of the general service taxonomy. We will use the best of these three open frameworks to establish logically structured, universal, and most importantly, well-performing metadata storage for runtime and design-time discovery. One open standard remains, which is especially designed for runtime discovery; it's our "yellow" book, and we will look at it now.

The UDDI taxonomy (V.3) in Oracle OSR

In contrast to all the possible repository taxonomies and ontologies, Service Registry is very well standardized. UDDI was one of the cornerstones of contemporary SOA, and Oracle can offer the latest release compliant with Version 3 of the open standard. To understand this standard, we must take a look at the **tModel** concept (`http://uddi.org/taxonomies/UDDI_CoreOther_tModels.htm`) as the key aspect of UDDI organization.

In discussing the Open Group Ontology, we mentioned two entities that represent a service to consumers and other composition members: `ServiceContract` and `ServiceInterface`. We also mentioned that the definition of `ServiceInterface` is generic and needs more detail for practical implementation. So, now, we can formulate that tModel (the technical model) as a complex data type, used for defining and representing the interface of a service we are going to discover and invoke (dynamically in our composition controller). In the case of the web service, tModel will at least represent the Service's WSDL as the URL, its name, and the description text, which is sufficient for service discovery and interpretation.
We recommend that you view OWL and tModel side by side:

OWL Service Interface	UDDI tModel for Service Interface (org.uddi.api_v3.TModel)

<owl:Class
rdf:about="#Service
Interface">
  <owl:disjointWith>
    <owl:Class
rdf:ID="Service"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class
rdf:ID="Service
Contract"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class
rdf:ID="Effect"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class
rdf:ID="HumanActor"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class
rdf:ID="Task"/>
  </owl:disjointWith>
</owl:Class>
	<complexType name="tModel">
	   <complexContent>
	     <restriction base="{http://www.
	w3.org/2001/XMLSchema}anyType">
	       <sequence>
	         <element ref="{urn:uddi-
	org:api_v3}name"/>
	         <element ref="{urn:uddi-
	org:api_v3}description"
	maxOccurs="unbounded" minOccurs="0"/>
	         <element ref="{urn:uddi-
	org:api_v3}overviewDoc"
	maxOccurs="unbounded" minOccurs="0"/>
	         <element ref="{urn:uddi-
	org:api_v3}identifierBag"
	minOccurs="0"/>
	         <element ref="{urn:uddi-
	org:api_v3}categoryBag" minOccurs="0"/>
	         <element ref="{http://www.
	w3.org/2000/09/xmldsig#}Signature"
	maxOccurs="unbounded" minOccurs="0"/>
	       </sequence>
	       <attribute name="tModelKey"
	type="{urn:uddi-org:api_v3}tModelKey" />
	        <attribute name="deleted"
	type="{urn:uddi-org:api_v3}deleted"
	default="false" />
	     </restriction>
	   </complexContent>
	 </complexType>

The simplest XML service descriptor based on the tModel's XSD from the preceding table will be as follows:

```
<tModel tModelKey="uuid:9AF82501-E6A9-1ba3-A094-2C7FE45CD859">
   <name>Public interface for adding NEW Mobile Service into clients
Order bundle </name>
   <description xml:lang="en">WS Interface for addNew Mobile CTU
generic Order</description>
   <overviewDoc>
      <description xml:lang="en">The service's WSDL document</
description>
      <overviewURL>http://www.ctu.com/bss/ services/order/
provisioning/
             /addMobileOrder.wsdl</overviewURL>
   </overviewDoc>
   ….
</tModel>
```

As you can see, this model is uniquely identified by UUID, used as a reference to this model. For the service's invocation, the most important element is `overviewURL`, containing the pointer to the service WSDL/Endpoint descriptor. All other elements are self-descriptive.

Rather minimalistic, isn't it? Generally, what we can get is the reference to WSDL and information about its structure (remember the discussions about abstract and concrete in *Chapter 1, SOA Ecosystem – Interconnected Principles, Patterns, and Frameworks*). We can figure out what a server does and what effects to expect. So, is UDDI just a list of WSDL? What if we need some more information about a service and, more importantly, not just as a consumer where just the Endpoint is enough, but as an Agnostic Composition Controller and/or Agnostic Adapter Factory? By the way, what if our service provider is not SOAP/WSDL-based at all?

Well, generally, we can put any type of Endpoint in the `overviewURL` element. All other service particulars can be packed in a bag. Literally, there is an element named `categoryBag`; please see its schema in the following code:

```
<complexType name="categoryBag">
   <complexContent>
     <restriction base="{http://www.w3.org/2001/XMLSchema}anyType">
       <choice>
         <sequence>
           <element ref="{urn:uddi-org:api_v3}keyedReference"
maxOccurs="unbounded"/>
```

```
              <element ref="{urn:uddi-org:api_v3}keyedReferenceGroup"
maxOccurs="unbounded" minOccurs="0"/>
            </sequence>
            <element ref="{urn:uddi-org:api_v3}keyedReferenceGroup"
maxOccurs="unbounded"/>
          </choice>
        </restriction>
      </complexContent>
    </complexType
```

So, the XML portion of this bag will be in tModel as presented in the following code:

```
</overviewDoc>
…
<categoryBag>
      <keyedReference
         tModelKey="uuid: 9AF82521-F6A9-1ba3-C094-2C7FE45CD859"
         name="Another specification to web service or other endpoint
descriptor"
         value="SomeWSDLSpec"/>
   </categoryBag>
</tModel>
```

But wait, can you figure out what the structure is? Yes, you're right, it's a name-value pair. What else could it be? There is no other way to define plural properties. By the way, the schema for identifierBag is similar, but a bit simpler. Thus, the categoryBag element is a collection of keyedReferences, where each of them is presented as a tModel. Therefore, you can have a collection of technical specs in the form of tModels to express different WS categories: messaging protocol, transport protocol, portType references, MEP types, and so on. Actually, mapping between WSDL and UDDI V2-V3 is pretty straightforward, and this fact raises a question: Why do we need a description for the description?

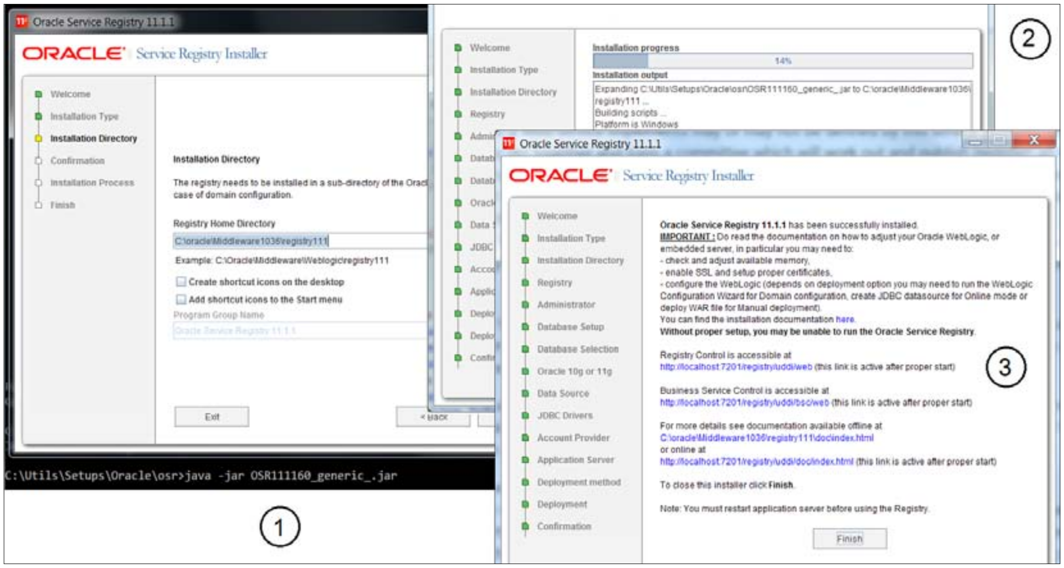
Well, individual services (for example, our milkman, postman, or doctor) could have a good description of their individual capabilities, but we need a structure to put all of them into our services' yellow book. Again, we have pretty good mechanisms to build this structure in the form of tModels, but we are (yet again) on our own in the quest of delivering it.

In the first chapter, we mentioned that our first attempt at establishing global service Discoverability was not a real success, and now you can see why. Initially, UDDI was not about web services, it was about business collaboration. tModels are just a way of describing entities and relations between them. The complete UDDI hierarchy consists of the following:

- `businessEntity`: It's your company, or business unit, that provides a wide variety of business services, and thus the model should be descriptive enough: the company's name, description of the line of business, contact details, and available business services. Previously mentioned business taxonomies, such as NAICS and UNSPSC, are very well suited for the description of this entity and are widely used in the `identifierBag` and `categoryBag` reference collections (via tModels of course). In this case, `identifierBag` answers the question "What is the company we are describing here?" and `categoryBag` addresses the questions "What are the functions of the company?" and "What are the services provided by the company?"
- `businessService`: The schema of this entity represents descriptive information about a particular family (or domain) of technical services. In addition to the name, the description, and `categoryBag`, this has one or several `bindingTemplates`, representing the technical description of publicly available services (as tModels of course).
- `bindingTemplate`: At the top, it has a human-readable description and the `AccessPoint` element holding the host URL. The URL type is provided as an attribute, such as an HTTP host. Detailed service info is gathered in the `tModelInstanceDetails/tModelInstanceInfo` collection.

With so much freedom provided by tModels and the lack of comprehendible taxonomies in the earlier 2000s, anarchy reigned in the SOA realm, quite severely damaging UDDI acceptance. With the new arrival of different API servers, we hope that UDDI will improve its reputation. It's still one of the existing strategic technologies shaping the SOA landscape, and Oracle plays its part quite well here.

To get a closer look at the tool, please proceed with the installation. It's quite similar to OER. Here, again, you should start with the DB preparation, although only one tablespace is necessary—the UDDI value-name pairs are quite light and do not require LOBs:



1

Installation Directory

The registry needs to be installed in a sub-directory of the Orac case of domain configuration.

Registry Home Directory: C:\oracle\Middleware1036\registry111

Example: C:\Oracle\Middleware\Weblogic\registry111

□ Create shortcut icons on the desktop

□ Add shortcut icons to the Start menu

Program Group Name: Oracle Service Registry 11.1.1

```
C:\Utils\Setups\Oracle\osr>java -jar OSR111160_generic_.jar
```

2

Installation progress: 14%

Installation output

```
Expanding C:\Utils\Setups\Oracle\osr\OSR111160_generic_.jar to C:\oracle\Middleware1036\
registry111 ...
Building scripts ...
Platform is Windows
```

3

Oracle Service Registry 11.1.1 has been successfully installed.
IMPORTANT : Do read the documentation on how to adjust your Oracle WebLogic, or embedded server, in particular you may need to:
- check and adjust available memory,
- enable SSL and setup proper certificates,
- configure the WebLogic (depends on deployment option you may need to run the WebLogic Configuration Wizard for Domain configuration, create JDBC datasource for Online mode or deploy WAR file for Manual deployment).
You can find the installation documentation here.
Without proper setup, you may be unable to run the Oracle Service Registry.

Registry Control is accessible at
http://localhost:7201/registry/uddi/web (this link is active after proper start)

Business Service Control is accessible at
http://localhost:7201/registry/uddi/bsc/web (this link is active after proper start)

For more details see documentation available offline at
C:\oracle\Middleware1036\registry111\doc\index.html
or online at
http://localhost:7201/registry/uddi/doc/index.html (this link is active after proper start)

To close this installer click **Finish**.

Note: You must restart application server before using the Registry.

Oracle service Registry installation

The installation is simple, as shown in the preceding screenshot (steps **1**, **2**, and **3** for training purposes). You do not have to create a separate WLS domain and install it on the existing OER domain. Just assign a different port for the console (we use 7201 as you can see in the next screenshot, step **1**). To get a better understanding of Oracle's approach to the tModels' implementation, include the demo data installation during the DB selection step. Upon browsing the DB schema, you will find that the internal DB tables' structure is clearly built around the UDDI taxonomy—there are dedicated tables for `BusinessEntity`, `BusinessService`, and `BindingTemplate`.

For each table, maintain separate bags for categories and identities containing references to tModels registered in the related table. Despite this model's simplicity, there are many additional service tables grouped in the service domains, as follows:

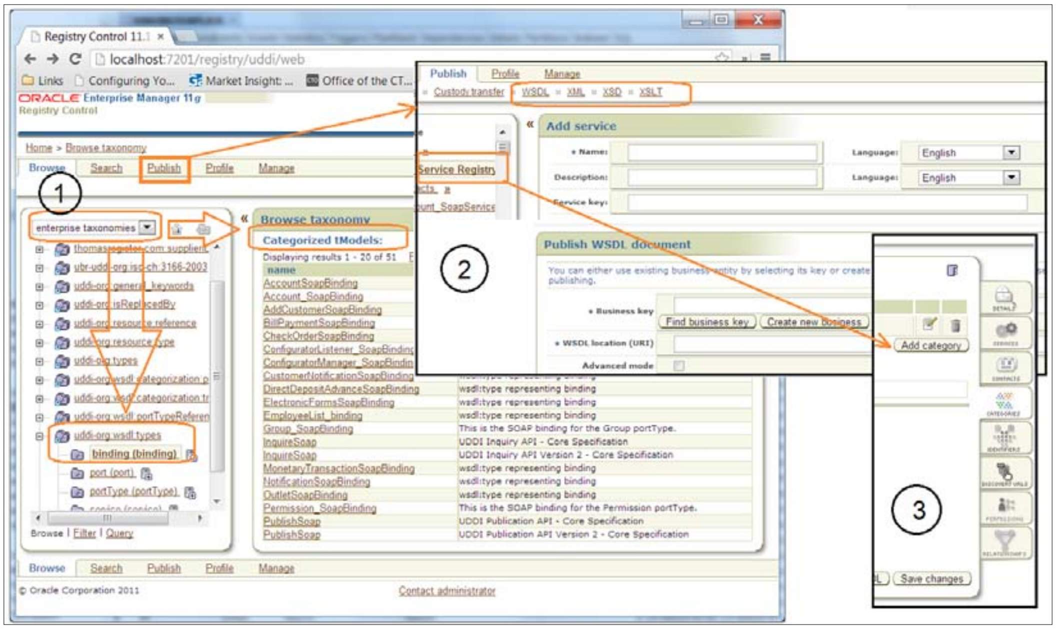
- Approval
- Access control
- Replication
- Events Reporting

This grouping should attract our attention as it has a direct relation with the physical realization of the SOA patterns and the Registry layering in particular. During the installation, you probably noticed that we can install the Registry in three different modes: Publication, Intermediate, and Discovery. This is how Oracle (quite cleverly) implements the double D in the UDDI standard. Naturally, as an architect, you will not allow anyone registering for new/updated services directly in your production registry. Initially, information should be injected into the Publishing registry (first D) and you can have as many as you need (per business domain, GU, or service roles). All these registries are stacked vertically, that is, they have the same rank. After approval, metadata will be propagated to the Discovery Registry (second D). You will find more details about the OSR data model/tModel relations and deployment topology in the Oracle documentation at `http://docs.oracle.com/cd/E14571_01/doc.1111/e15867/uddi.htm`.

Publishing service artifacts and taxonomy categories (see the following screenshot, part **2**) using the UDDI console is straightforward. Just follow the screen instructions after clicking on the menu options at the top (for WSDL and other XML-based artifacts) or the tabs on the left for **Details**, **Categories**, and **Identifiers** (see the following screenshot, part **3**) when modifying business entities or tModels.

According to the SOA Governance cycle, any artifact has to pass several approvals and be accepted by several custodians (for example, Service, XSD/Schema, and the Policy and Registry custodians). Thus, in addition to vertical layering, we will have a horizontal chain, presented by the intermediate registries between the publication and discovery.

It is wise to have individual Intermediate Registries in every individual test environment, JIT-UAT-ORT as well, and promote services to production only after passing all acceptance gates:

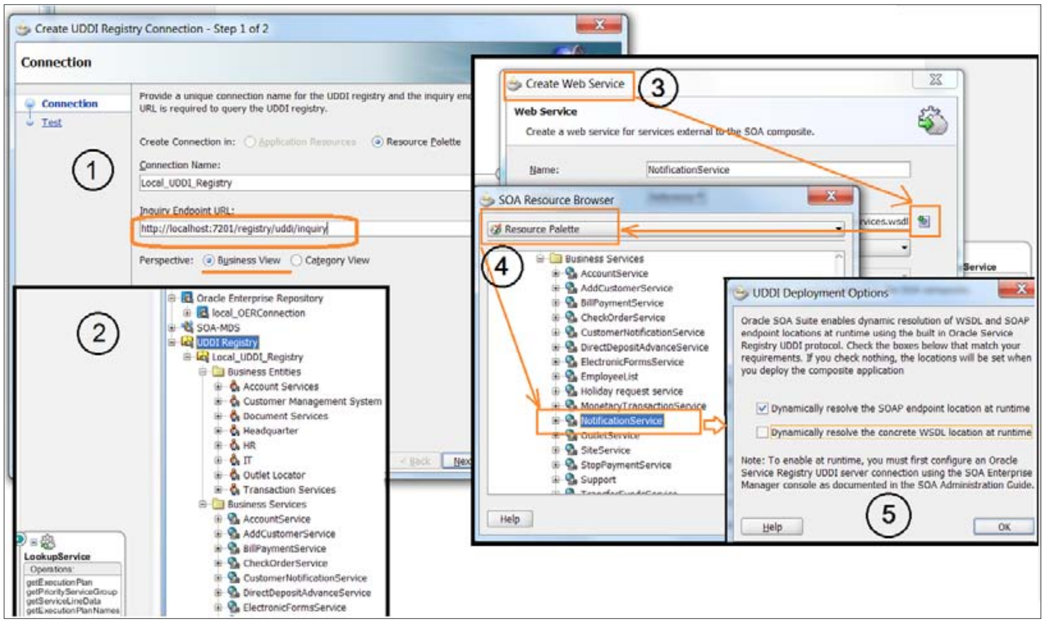

Registry Control 11.1
localhost:7201/registry/uddi/web
Links
Configuring Yo...
Market Insight: ...
Office of the CT...
ORACLE Enterprise Manager 11g
Registry Control
Home > Browse taxonomy
Browse
Search
Publish
Profile
Manage
1
enterprise taxonomies
thomasregister.com suppliers
ubr-uddi-org iso-ch:3166-2003
uddi-org general_keywords
uddi-org isReplacedBy
uddi-org resource reference
uddi-org resource type
uddi-org types
uddi-org wsdl categorization p
uddi-org wsdl categorization tr
uddi-org wsdl portTypeReferen
uddi-org wsdl types
binding (binding)
port (port)
portType (portType)
conico (conico)
Browse | Filter | Query
Browse taxonomy
Categorized tModels:
Displaying results 1 - 20 of 51
name
AccountSoapBinding
Account_SoapBinding
AddCustomerSoapBinding
BillPaymentSoapBinding
CheckOrderSoapBinding
ConfiguratorListener_SoapBinding
ConfiguratorManager_SoapBindin
CustomerNotificationSoapBinding
DirectDepositAdvanceSoapBinding
ElectronicFormsSoapBinding
EmployeeList_binding
Group_SoapBinding
InquireSoap
InquireSoap
MonetaryTransactionSoapBinding
NotificationSoapBinding
OutletSoapBinding
Permission_SoapBinding
PublishSoap
PublishSoap
wsdl:type representing binding
wsdl:type representing binding
wsdl:type representing binding
This is the SOAP binding for the Group portType.
UDDI Inquiry API - Core Specification
UDDI Inquiry API Version 2 - Core Specification
wsdl:type representing binding
wsdl:type representing binding
wsdl:type representing binding
This is the SOAP binding for the Permission portType.
UDDI Publication API - Core Specification
UDDI Publication API Version 2 - Core Specification
Browse
Search
Publish
Profile
Manage
© Oracle Corporation 2011
Contact administrator
Publish
Profile
Manage
Custody transfer
WSDL
XML
XSD
XSLT
Service Registry
Add service
Name:
Description:
Service key:
Language:
English
Language:
English
2
Publish WSDL document
You can either use existing business entity by selecting its key or create publishing.
Business key
Find business key
Create new business
WSDL location (URI)
Advanced mode
Add category
3
Save changes

Publishing service artifacts on UDDI

Although Oracle provides a clear OSR installation guide, we would like to advise you to be careful with the node installation sequence for security reasons. Naturally, all centralized assets concentrated in the Registry are protected by security policies and ACLs. We recommend LDAP for the storage of user accounts, but trust the relations between registries in the propagation chain, maintained by digital security certificates. Therefore, we advise you to start from the end. The digital security certificate of the Discovery Registry is needed when installing the Publication Registry.

Another highly important SOA pattern must be strictly observed here due to the critical nature of the Registry and its position as the single point of failure: redundant implementation. In the WLS-based topology, we choose from the very beginning; it's attained by means of a WebLogic Cluster. The Cluster operation is achieved by running multiple registries and combining their functionalities with a load balancer (proxy). The configuration of this infrastructure is common for WLS and well documented in the HA section of the OSR installation guide.

As the central point of the Enterprise SOA, OSR has all the possible connections to the SOA Suite, Enterprise Repository, OSB, and development environments (JDeveloper). In the following screenshot, you can see how, in five easy steps, we can initiate the dynamic resolving of the WSDL Endpoint location using Oracle Registry in JDeveloper. First we establish the connection (**File | New | Connections | UDDI Registry Connection**) and then verify it in the JDev Resource palette. Now you will see the three connections ready and at your service (including OER and the old MDS):



Create UDDI Registry Connection - Step 1 of 2

Connection

Provide a unique connection name for the UDDI registry and the inquiry end URL is required to query the UDDI registry.

Create Connection in: ○ Application Resources ● Resource Palette

Connection Name: Local_UDDI_Registry

Inquiry Endpoint URL: http://localhost:7201/registry/uddi/inquiry

Perspective: ● Business View ○ Category View

Create Web Service

Web Service

Create a web service for services external to the SOA composite.

Name: NotificationService

UDDI Deployment Options

Oracle SOA Suite enables dynamic resolution of WSDL and SOAP endpoint locations at runtime using the built in Oracle Service Registry UDDI protocol. Check the boxes below that match your requirements. If you check nothing, the locations will be set when you deploy the composite application

☑ Dynamically resolve the SOAP endpoint location at runtime

□ Dynamically resolve the concrete WSDL location at runtime

Note: To enable at runtime, you must first configure an Oracle Service Registry UDDI server connection using the SOA Enterprise Manager console as documented in the SOA Administration Guide.

Configuring a service artifact's dynamic resolution

Drag a new web service to the right SCA swimlane and click on the icon for the WSDL selection; take it from the Resource palette and then select the earlier published (previous figure, step **2**) service.

After that, you will be prompted to select a runtime dynamic resolution type (the SOAP Endpoint or WSDL), and that's the essence of our runtime Discoverability using UDDI! Depending on the selected UDDI deployment option, the `composite.xml` file will have a different syntax for the `binding.ws` element, describing the Endpoint's binding location.

Endpoint type	**Binding realization**

SOA Endpoint	`<binding.ws port="http://ctu.com/wsdl/dev/uddi/services/#wsdl.endpoint(OrderStatusService/OrderStatusService)"`
	`location="http://localhost:7201/registry/uddi/doc/dev/OrderStatusService.wsdl"`
	`soapVersion="1.1">`
	`<property name="oracle.soa.uddi.`**`serviceKey`**`" type="xs:string"  many="false">`
	`uddi:9AF82521-F6A9-1ba3-C094-2C7FE45CD859`
	`</property>`
	`</binding.ws>`

WSDL	`<binding.ws port="http://ctu.com/fulfillment/order#wsdl.endpoint(OrderStatusService/OrderStatusService)"`
	`location="orauddi:/uddi: 9AF82521-F6A9-1ba3-C094-2C7FE45CD859"`
	`soapVersion="1.1">`
	`</binding.ws>`

Now you can use any service with the dynamic address resolution in any service composition. This basic yellow book functionality is perfectly fine, but we need more search capabilities with adequate granularity for search criteria, which is suitable for our agnostic controller. Here, we face the second type of limitation (apart from the lack of taxonomy guidance, which is mostly based on the WSDL binding)—limited search capability. In general, the classic UDDI provides us with search restricted to the WS name and its classification. Because of the name-value pair approach of the tModels, there is no uniform way to query services and their attributes. Some attempts were undertaken in the UDDI V.3 specs (see sections 4 and 5.1.8 of the Inquiry API functions, `http://uddi.org/pubs/uddi-v3.0.2-20041019.htm#_Toc85908076`). You will find 10 generic functions to find core UDDI entities (Business, Service, and Bindings) and get details about them, including tModels.

Oracle offers a solution for these limitations by providing the UDDI API, which covers both Ds—Description (Publishing API) and Discovery (Inquiry API). In the scope of dynamic composition controller functionalities, the latter is of higher interest to us. Technically, we have two Inquiry APIs: ClientSide and UI. The last one has only one operation, `get_entityDetail`, which will return the list of UDDI data structures. Using the ClientSide API, you can call any standard UDDI V2 inquiry function. The most commonly used parameters are `Name` (`find_<searchingEntity>.setName(new Name(Name)))`, `serviceKey` `(find_<searchingEntity>.serServiceKey(serviceKey)))`, and `tModelKey` `(find__<searchingEntity>. addTModelKey (tModelKey)))`. We can also define any String qualifier in our search, such as `find_tModel.addFindQualifier (findQualifier)`, when preparing the `find` object. This gives us some freedom in defining our search criteria, but we must be certain about what we are looking for and the parameters available to define our search in the SBDH-compliant Message Header. These parameters are marked in bold in the full version of `MessageHeader` in the following code. So, some of them can be categorized as Service Entities, while most of them qualify as tModels for the taxonomy we are about to build:

```
<?xml version="1.0" encoding="UTF-8" ?>
<urn:CTUMessage xmlns:urn="urn:com:telco:ctu:la:ctumessage:v01"
                xmlns:urn1=" urn:com:telco:ctu:la:messageheader:v01"
                xmlns:urn2=" urn:com:telco:ctu:la:processheader:v01"
                xmlns:urn3=" urn:com:telco:ctu:la:payload:v01"
                xmlns:urn4=" urn:com:telco:ctu:la:messagetrackingdata
:v01">
    <urn:MessageType>EBO</urn:MessageType>
    <urn:Version>0.1</urn:Version>
    <ns11:MessageHeader xmlns:ns11=" urn:com:telco:ctu:la:messagehead
er:v01">
        <ns11:RefId>604244_1</ns11:RefId>
        <ns11:RequestId>CMSA697</ns11:RequestId>
        <ns11:MsgId>EBM109</ns11:MsgId>
        <ns11:RefDateTime xsi:nil="true" xmlns:xsi="http://www.
w3.org/2001/XMLSchema-instance"/>
        <ns11:Sender>
            <ns11:SenderCode xsi:nil="true" xmlns:xsi="http://www.
w3.org/2001/XMLSchema-instance"/>
            <ns11:CountryCode>BR</ns11:CountryCode>
            <ns11:Affiliate xsi:nil="true" xmlns:xsi="http://www.
w3.org/2001/XMLSchema-instance"/>
            <ns11:Instance>BR_IP</ns11:Instance>
        </ns11:Sender>
        <ns11:ObjectReference>
            <ns11:ObjectName>Order</ns11:ObjectName>
            <ns11:ObjectKeyName>OrderID</ns11:ObjectKeyName/>
            <ns11:ObjectKeyData> CMSA-697BR09521</ns11:ObjectKeyData/>
            <ns11:Domain>Fulfillment</ns11:Domain>
        </ns11:ObjectReference>
        <ns11:ObjectContext>
            <ns11:ParameterValue name="ActionType">DROP</
ns11:ParameterValue>
            <ns11:ParameterValue name="ProductType">IP</
ns11:ParameterValue>
            <ns11:ParameterValue name="ServiceType">SERVICE</
ns11:ParameterValue>
        </ns11:ObjectContext>
    </ns11:MessageHeader>
```

Now that we are equipped with the knowledge of OER and OSR functionalities and open SOA ontology standards, we will continue with the functional analysis of runtime Discoverability requirements for the composition controller.

Runtime Discoverability analysis

We will start with the runtime Discoverability requirements because it seems a bit easier—we are already using runtime lookup for the service particulars and different XML artifacts in the EBF and EBS service composition controllers, and all requirements are expressed in the execution plan's structure (look at `ExecutionPlanLookupService`). This is our Service Registry and is currently based on MDS, but our intention is not to isolate Registry and Repository, but to rationally combine them into one management pack. In this respect, Oracle has two products to offer, OER and OSR, with a utility for the synchronization of metadata between them (orrxu, `http://docs.oracle.com/cd/E21764_01/doc.1111/e16580/oereu.htm`). Adding the metadata harvesting capability in OER for reverse engineering and the requirements for OSR integration with WLS for the automatic registeration of new service deployments in the Service Registry will complete the picture of Oracle's response to the Discoverability principle and SOA Governance.

But isn't it too complex? Yes, it is. The simple fact that the OER DB schema has 145 tables (Release 11.1.1.7.0) for "all weather conditions" doesn't make our life easier. From the very beginning, it was clear that managing complex technologies such as SOA would not be easy, but we should expect a bit more methodological support for SOA runtime compositions in particular. We are about to provide this support to the best of our ability, focusing on vendor-neutral SOA principles first and then extending them to Oracle's product realization (OSR and OER).

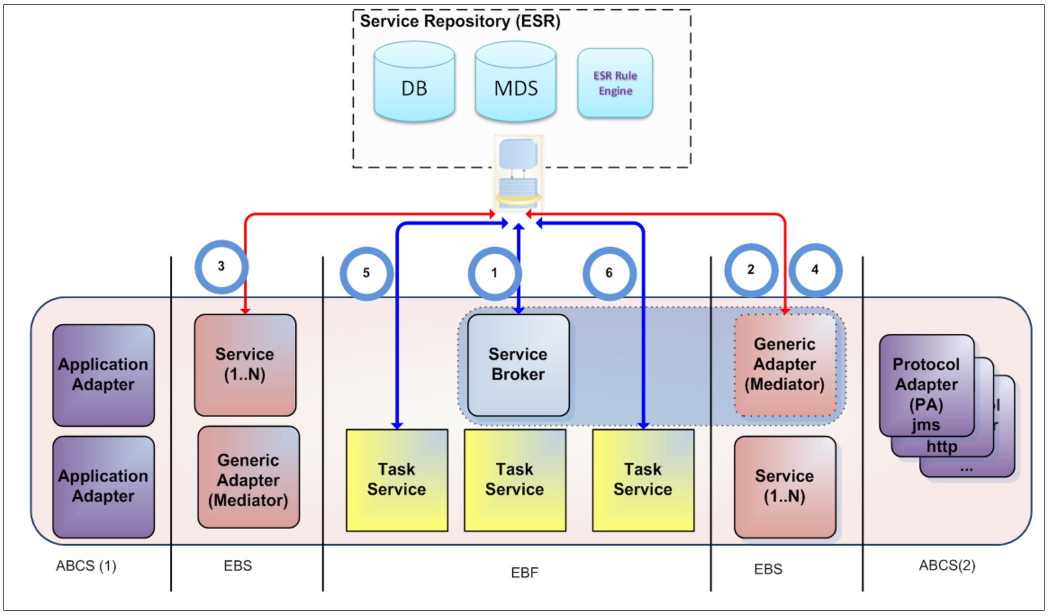
We will do this exactly how we did in the previous chapter: discuss a generic Message Broker requirement first and then implement it as a Service Broker on OSB. Speaking of which, we must mention certain things related to its complexity:

- Runtime and Design-time lookups are closely related and the physical segregation of Registry and Repository is not always optimal from a performance point of view in regards to the complexity of queries. Registry realization is UDDI compliant; thus, we will have three major XML data structures, presented in tModels. In general, it's a representation of service interfaces (for instance, WSDL for WS), and all other related extra features come in value-name pairs. It may be flexible (in terms of the value-name pairs), but not always interpretable and, therefore, discoverable.
- Continuing with a repository's segregation issues, we can say that simply mixing SOA project data, service design particulars, runtime logs, service session data, results from load/stress—only because they cannot be easily defined in the tModels taxonomy or are not related to the Registry—will definitely not improve our SOA Governance (see a single schema of OER). Keeping the Governance under control, we can clearly identify the boundaries of `AuditLogs`, `ErrorLogs`, project control data, and the Service Repository itself. This consideration requires a precise definition of Inventory Endpoints' interface(s), suitable for all types of Governance actions.
- To continue with Governance, a services metadata harvesting tool is a really effective feature. However, ask yourself: if you, as an architect, have been devising an enterprise Service Inventory for a while now and still need to perform reverse engineering to reveal hidden dependencies, then how valid is your service taxonomy? Could it be that you overlooked some services/entities during the design phase? No, we are not saying that a harvester is unnecessary; that's not the point. Indeed, you should run it regularly (it's an Ant task in JDeveloper), and if in OER Asset Editor's search report you see anything new or something that you haven't seen in the `Unsubmitted/Unfilled` folder, you can proudly ask for a raise.
- Whatever technical realization you choose for our Service Inventory (building blocks such as services or components; languages such as Java, C, PL, and SQL; and so on), your primary concern is the availability of your Service Registry/Repository for the flawless support of Discoverability, along with other utilities from the first table in this chapter. Please take a second look at the design rules in this table.

What would be the most logical approach to address availability and performance issues expressed in the preceding bullet points? Correct, to position SR as close to the composition controller as possible. But how would this be possible? Only by the segregation of the service taxonomy model from its physical realization. Doing so, we should be able to re-implement it on any technical platform, easily accessible by the concrete composition controller. Remember, Secure Gateway in DMZ is an ESB too, and Service Broker is common to both. Thus, service metadata lookup is not an extraordinary feature. Will you query your production OER or OSR from DMZ? Think twice. You could have the discovery node in DMZ, but what about the physical storage? Oracle DB? Again, think twice when it comes to security. You should be quite close to the iron. How will you securely synchronize your Production and DMZ discovery nodes?

Runtime lookup

One of the practical ways to classify services and artifacts' taxonomy is to detect the type of data we see on every service layer. For vertical infrastructure layering, we suggest that you use the Oracle AIA service layer notation. This leaves us with three main layers: Adapters, Enterprise Services (usually hosted or available through ESB), and task-orchestrated services in the Enterprise Business Flow layer. Note that vertical stratification for the three main service models still remains; vertical layering is presented in the following figure:



Based on the preceding figure, in the following table, we consolidated all the possible lookup and entity types that our Service Repository must maintain and reliably provide. The table has a numerical index for Lookup Types (runtime discovery use cases) for simple reference in this and further chapters.

Lookup type 1: The service business delegate is looking for a service worker:

Role	Location	Entity	Example
Composition controller / Composition subcontroller	EBF, EBS	Service as a URL, Component as a URL	• One BPEL process invokes another within the business domain depending on the context. • One ESB service needs to relay a message to another. A URL as a variable is returned.

Lookup type 2: The service is looking for the Endpoint(s):

Role	Location	Entity	Example
Composition subcontroller, Dispatcher, and Mediator	EBF, EBS	The TP Endpoint URLs could be as follows: File, JMS, HTTP, and FTP	• A BPEL process as a final worker wants to deliver a message to its final destination/ESB. • ESB wants to deliver a message to SCA or to the Endpoint. A URL as a variable is returned.

Lookup type 3: The service wants to perform data transformation/validation:

Role	Location	Entity	Example
Dispatcher and Mediator	EBS	• URL to XSLT • XQuery String	The ESB Service resolves parameters for transformation/ Enrichment

Lookup type 4: The service is looking for Endpoint particulars (`bindingTemplate` in the tModel notation):

Role	Location	Entity	Example
Dispatcher and Mediator	EBS	Object (EBO/SDO)	ESB is looking for the Endpoints' particulars (Transport, Proxy, Port, and Username/ Password).

Lookup type 5: The service is looking for an internal task's parameters:

Role	Location	Entity	Example
Service participant	EBF	Object (EBO/ EBM)	• WSIF invocation, similar to Oracle E-Business Suite SOA Gateway realization; see the Metadata definition section at `http://docs.oracle.com/cd/E18727_01/doc.121/e12169/T511175T513090.htm`. • Java callout from BPEL or Mediator to obtain the additional object's parameters.

Lookup type 6: The service is making a decision(s):

Role	Location	Entity	Example
Dispatcher, Mediator, Service-participant	EBF, EBS	Object (EBO/SDO), TP Endpoint URL	• BPEL uses the Rule Engine to get a value (any value including `true` or `false`) or an object. • The ESB Service is looking for another service.

Entity types

Summarizing all the entity types from the preceding table, we come up with the six main types, which are in line with the public classifications proposed by Open Group and SAIF BF:

1. Objects (top hierarchy entity).
2. Messages or message particulars (XML or a representation of a serialized object in transit).
3. Services, tasks, or a task's particulars (including WSDL and its parts).
4. An application's Endpoints or Endpoint particulars (SOAP or other types).
5. Rules or Rulesets.
6. Enterprise Business Events.

The SR physical implementation can be as follows:

- File-based (for instance, temporarily MDS has been used in all previous chapters)
- DB-based (planned for flexibility and performance):
    - The OER DB schema with complex assets relations
    - The OSR entities with metadata elements in tModels
    - Custom DB with OER/OSR synchronization (a custom DB structure is presented later in *The SQL Implementation of the service taxonomy* section in this chapter)

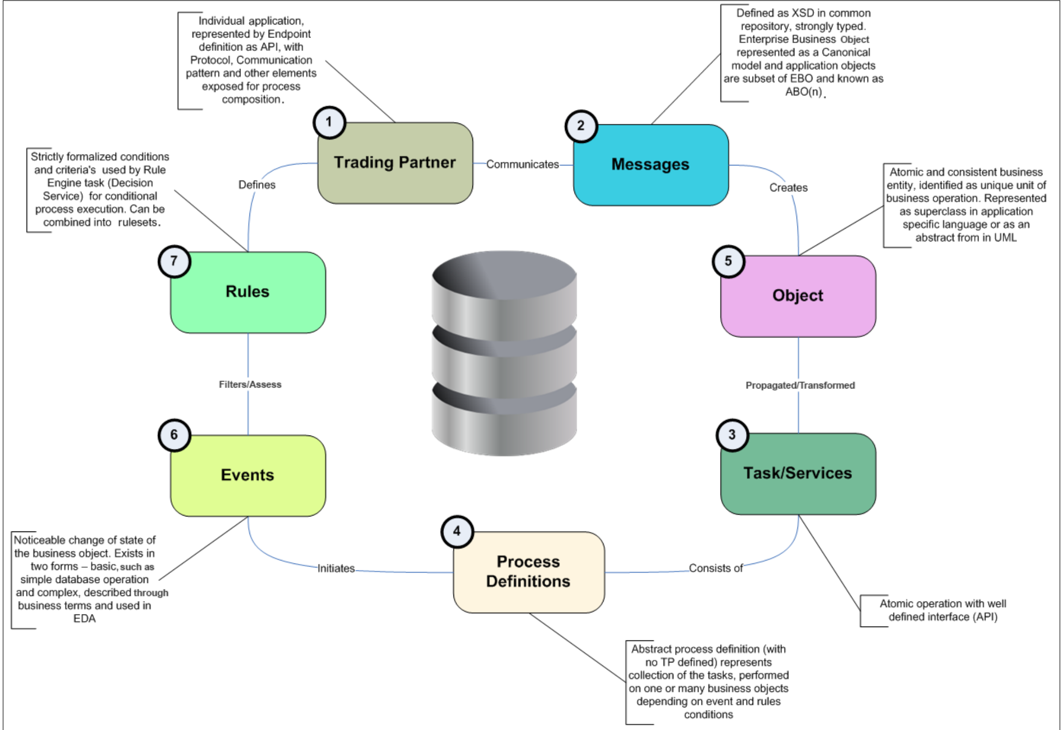
The composite entities (such as tasks) can be constructed as:

- Static from the file location
- Dynamic from the DB query
- Static from the DB
- Dynamic from the RE

A common rule for the ER implementation for all approaches is to have a unified ESR endpoint for an entity lookup, with the `MessageHeader` elements as an input parameter.

Entity types' relations

With entity types accounted for and identified, to save your time, we will jump right to their relations and explain their roles later:


Individual application, represented by Endpoint definition as API, with Protocol, Communication pattern and other elements exposed for process composition.
Defined as XSD in common repository, strongly typed. Enterprise Business Object represented as a Canonical model and application objects are subset of EBO and known as ABO(n).
1
Trading Partner
Communicates
2
Messages
Strictly formalized conditions and criteria's used by Rule Engine task (Decision Service) for conditional process execution. Can be combined into rulesets.
Defines
Creates
Atomic and consistent business entity, identified as unique unit of business operation. Represented as superclass in application specific language or as an abstract from in UML
7
Rules
5
Object
Filters/Assess
Propagated/Transformed
6
Events
3
Task/Services
4
Process Definitions
Initiates
Consists of
Noticeable change of state of the business object. Exists in two forms – basic, such as simple database operation and complex, described through business terms and used in EDA
Atomic operation with well defined interface (API)
Abstract process definition (with no TP defined) represents collection of the tasks, performed on one or many business objects depending on event and rules conditions

SR entity relationships

In the center, we have our Entity Repository storage that holds all the metadata in a secure, interpretable, and discoverable fashion. According to Conway's law (`http://www.melconway.com/Home/Conways_Law.html`), it can be organized in at least four different ways:

- Decentralized (project-based)
- Domain
- Cross-domain
- Enterprise

We will start with the simplest project-based realization.

Decentralized realization

The idea of decentralized realization is to avoid lookups of any kind and maintain the orchestration logic as a static process. This way, processes will be reconfigured only through recoding and reimplementation. Simply put, this option can be described as *do nothing*. We identified all the downsides of this approach at the beginning of *Chapter 3*, *Building the Core – Enterprise Business Flows*, when discussing the assessment of the CTU SOA solutions.

The application project store

The primary goal of this realization is to establish a centralized metadata repository for all artifacts developed within a single SOA project. This repository can support runtime lookups (for instance, in SCA Mediator), but for entities designed for a specific SCA, limited by a single project or a small group of projects. The structure of metadata, its taxonomy, and ontology will be completely at the discretion of project's team lead. You will certainly remember the small exercise from *Chapter 4*, *From Traditional Integration to Composition – Enterprise Business Services*, when we implemented a basic proxy on OSB. In the first step of this exercise, we created a common folder structure for XML-based artifacts. We realize that many of you found it far from optimal and different from your usual classification. This is exactly our point. A project-centric repository, based on a similar (custom) approach and maintained using Oracle Metadata Services (MDS), is extremely flexible and convenient for a single department; however, it requires constant vigilance from SOA architects and Governance specialists. In fact, as is, it fits most of the needs of a small department. The positive side is that the performance of the MDS lookups (for file- and DB-based MDS realization) is quite good. Based on the runtime lookup scenarios' individual tables from the preceding section, we can identify the lookup types and entities as follows:

- **Lookups types in use**: The types are none or limited, which use the oramds protocol
- **Entities maintained**: The entities are none and are project specific in MDS

The first approach in project-based SR realization is straightforward:

- The first and second assumptions (from the *General objectives* section) are taken for granted and the provisioning flow is functionally decomposed at the level where agnostic common services are separated from the functionally complete GU-specific business services.

  As a result, the layers of the Service Inventory are established for the task and entity services.

- Design new, full-scale, functional compositions to minimize the creeping of business logic, that is, reduce the number of compositions for maintenance and reusability purposes. This is the classic *top-down approach*.

  A top-down approach means to devise a complete analysis upfront. It does not just take a lot of time, but considering a dispersed GU, it requires deep and precise knowledge of the entire business operation everywhere, not to mention a substantial budget. In general, it is too late to conduct a top-down analysis at this stage, although the analysis itself is a positive practice.

- Use SCAs to implement compositions in a static BPEL way. Dynamic Service/Endpoint invocations with lookups, avoiding the creation of BPEL flows, are more visible when it comes to inexperienced developers.

  The Utility Services layer in the Service Repository is deliberately neglected for the purpose of simplification. In fact, it can potentially lead to the implementation of hybrid services. In this case, reliability (Objective 1; the first objective from the objectives table at the beginning of this chapter) is reduced.

  If vendors' SOA knowledge is deliberately not considered very high (for the man/hour cost reasons), it will broaden the choice of vendors; however, this potentially invites inexperienced solution providers and affects reliability.

  In general, processes will be identical to minor alterations and developed using the copy and paste approach. Maintainability (Objective 3) will be severely affected. With no common Utility components as the single point of failure, reliability can be high. However, without design time discovery, after several implementation laps, it will be virtually impossible to maintain the desired level of reusability (Objective 2).

  Reliability (Objective 1) can vary by process, depending on the complexity of the orchestration logic. The more complex the "if-else" logic used, the more prone to errors the process will be. As a workaround solution, SCA mediators with static dispatching logic can be implemented. Math for mediator filters/branches can be the same as that for the number of processes.

- Service Endpoint handling in ESB is similar to the SCA solution. The number of services will equal the number of channels multiplied by the number of affiliates. Goals will be affected in a similar way.

Centralized realization

Centralization denotes constant reuse through runtime resources lookup and discovery. The types of lookups and objects are defined in the table in the *Runtime lookup* section. The alteration of the Governance rule by configuration will provide the most profound benefits when maintained centrally. The following approaches practice the same lookup paradigm with different degrees of centralization and lookup frequencies.

It is obvious that the number of cross-platform lookups should be limited due to performance requirements, and the scope of the returned objects must be adjusted to its transactional scope. For this purpose, according to the third assumption, we implemented the Message Container with the Process Header (SBDH compliant), where the business object and transaction-related values must be persisted and propagated along the way via all the layers. So, a certain trade-off must take place to optimize the size of the process transaction-specific data, the number of lookups, and the transaction MEP.

The Message Container implementation with PH/MH also allows us to have significant independence from the platform vendors.

Domain Repository

Establishing a centralized Service Repository with global lookup capabilities for the entire enterprise is not only expensive but also unnecessary. The reasons can vary from dispersed geographic locations of business units (GU) to dissimilar business models. For instance, for a telecom company with a dedicated OSS/BSS division, several of their services and artifacts will be useful only within this single domain. That is, order management-related services are not concerned with the technology domain, responsible for maintaining Software-defined Networking. At the same time, the number of correlated projects and their common SLA requirements make a decentralized approach not only feasible, but also dangerous. Let's take a look at what artifacts and lookup types we can employ in this situation:

- **Lookup types in use**: 1-4
- **Entities maintained**: 1-4

This approach can be a good choice in the following scenarios:

- A GU has a great deal of independence in order to stay more flexible in terms of business operations
- The GU is supplied with all the necessary SOA guidelines and has a strong SOA sponsorship that is willing to follow the Enterprise Integration Center of Competence (ICC) guidelines (expressed in the first table in this chapter)
- The GU is capable of maintaining its own SOA assets and infrastructure
- The GU SOA assets are mostly GU specific, so establishing an Enterprise Repository is simply impractical

Usually, we expect that lookups 1-4 are used. A domain Service Broker(s) will be implemented to perform service dispatching. Part of the Service Broker, the service locator, must discover enough information to support end-to-end transactions and supply the Message Broker (ESB) with all the information for 3-4.

The design rules are as follows:

- **Synchronous MEPs**: This includes one DR lookup per transaction and persisting data in Process Header
- **Asynchronous MEPs with Global Correlation ID**: This comprises one DR lookup and one PH lookup by CorrID
- **Asynchronous MEPs without Global Correlation ID**: This involves more than one DR lookup, depending on the number of services/operations to invoke

Again, the preceding rules are subject to trade-offs and depend on the level of service granularity, message size, and process simplification.

This approach is also very traditional and requires you to perform the following steps:

1. Functional decomposition does not have to be completed before implementation. Only the Utility Services must be clearly identified upfront, which is simple as these services are well patterned: Service Broker, Translator, Transformer, RE Endpoint, DE Endpoint, and Message Broker. Business services can be presented initially in big chunks; this is suitable for further decompositions. This is the typical meet-in-the-middle approach, where top-down and bottom-up benefits are combined. As a result, the optimal delivery time with measurable and attainable performance is significantly better than that with a decentralized approach. Also, reliability is constantly maintained in a balanced manner along the decomposition.

2. Business logic and complex composition logic are removed from the Composition Controllers and subcontrollers in order to make the composition adjustable through simple configuration files (entities and rules), and not by recoding. The important thing here is that Composition Controllers don't have to be totally abstract and agnostic because they are predefined in the business and/or GU domain. This is also a way to provide trade-off reusability for time to market. The borderline is where the EBO for the Composition controller is implemented in the Message Container as the payload `<any>` or within the Message Container namespace. It could be acceptable to have an alternative approach in this type of realization. This causes a positive impact on reusability. Maintainability is significantly increased. Performance could be potentially less than that in a direct coding approach, but it can be easily justified through the resizing of compositions. This is attainable through configuration. Reliability can also be negatively impacted as we have implemented some single points of failure here, but caching and redundant implementation can solve this problem just as easily.
3. The transactional part of an extracted configuration persisted in the Message Container and Process Header (execution plan, set of transactional variables, or routing slip). Process Header will be propagated end-to-end to the adapter framework before the ultimate receiver. This positively impacts all characteristics.
4. EBS in ESB will use PH values for Transformation (enrichment), Validation, Filtering, and the Invocation service, or its ABCS. A minimal number of lookups is allowed as this layer must be a good performer. However, if necessary, Java callouts to the MDBs (such as RE MDB) are allowed. This approach is in alignment with the Delivery Factory pattern, where groups of adapters to the ultimate receiver can be abstracted through the factory layer. Grouping is usually done by MEP and the transport protocol. This positively impacts all characteristics.

The Cross-domain Utility layer

Apparently, even in a decentralized enterprise, some domain-specific services can be utilized between dissimilar domains. First, this could be completely agnostic Utility Services. Of course, their utilization across different domains will be carefully evaluated using performance numbers derived from their usage statistics and SLA declarations. Demands could be too high for the installation of a single utility service, and we should use redundant implementation in order to address it. Nevertheless, this is the same service, and we did not reinvent it. We just discovered it and added a new service node.

Entity services are also good candidates for cross-domain implementation. For instance, the Customer entity service (usually from OSS/BSS) can provide vital information for authentication and authorization purposes to all other services in the enterprise. Let's take a close look:

- **Lookups types in use**: 1-5
- **Entities maintained**: 1-5

The main disadvantage of the previous approach is that we identified and implemented, but didn't reuse it across all the domains' Utility layers within the Service Repository. The Utility layer is too generic, so with some effort, it could be totally reusable. These efforts were clearly identified in the previous approach as follows:

- Make a broker payload-independent, presenting the `<any>` block in the Message Container
- Implement an SBDH-compliant Message Header as a reference to the payload
- Implement the Process Header as a persistent container to route the slip/execution plan
- Implement the Audit/Message Tracking Data to track message information

The last point is a positive outcome of this implementation, as a universal Message/Service Broker will endorse the implementation of other OFM common patterns, for example, Error Hospital, Common Audit, and Centralized Logging. This is highly important to maintain a unified contract for all Service Broker-connected components, which can be fairly simple with the implementation of a Message Container as described previously.

Cross-layer Utility Services will be more thoroughly reviewed and tested against a domain-specific implementation. Moreover, it will have a positive impact on reliability and performance. The SB scalability must be treated with the utmost care as it may be a success factor for a cached/clustered implementation. Apparently, SB/SR become single points of failure; therefore, rigorous stress testing is absolutely required. A dedicated framework must be devised for this. Due to the implementation of a Canonical Endpoint for all composition members, an alternative implementation based on J2EE can be presented with relatively little effort.

Reusability and maintainability will be higher than those with previous methods. However, as it is still at the domain level (GU), it is not yet at the top level and is quite decentralized.

All four of the preceding steps are identical to the previous solution, except that the implementation of the Utility layers/patterns is governed from a central location.

The Enterprise Service Repository

The Enterprise Services Repository (ESR) is the central repository, where we define, access, and manage SOA assets such as services and data types. The repository stores the definitions and metadata of enterprise services and business processes. According to the reference tables, we have seen quite simple definitions for entities and lookup types:

- **Lookups types in use**: All
- **Entities maintained**: All

If a GU prerequisite from the Domain Repository is left out, the implementation of the Domain Repository with/without the common utility layers becomes problematic. In this case, decentralization will effectively lead to the same disorder and the implementation of the application's project store. The key success factor for the Domain Repository is to maintain the Canonical Endpoints and a unified configuration for the PE, as mentioned in the previous section.

With this approach, all entities must be maintained centrally. This is important because development and maintenance are already performed centrally in the Latin-American HQ. Therefore, all decomposed and recomposed GU-related flows will be maintained and configured in a single Service Repository. According to an evaluation, up to 80 percent of all business flows across GUs have the same structure and logic (we are in the telecom business after all), and therefore, the main lookup types will be 2, 3, and 4.

All four implementation steps will be similar to those in the Domain Model with the Cross-Domain Utility layer, with some exceptions:

- The Functional Decomposition will be on the GU and affiliate level.
- The possible decomposition parameters for Service Broker and Message Broker will be stored centrally as execution plans/routing slips.
- Before implementing a new process, a diligent investigation must be conducted. This could result in the further decomposition of existing processes, with the next decomposition occurring according to new requirements. This may end with the implementation of a new execution plan.
- The Rule Engines will be used very extensively. The main concern with this will be the implementation of inexpensive rules (avoid the "explosion" of rules). This is already done by splitting the rule tables per domain.

As a conclusion of this approach, we can expect the performance to be the same as that in the previous one. Maintainability and reusability will be at the top, and stability will be our main concern and the subject of proper infrastructure implementation.

Creating a lightweight taxonomy for dynamic service invocations

Now it is time to assemble a single practical taxonomy for all service entities that our expert teams have identified in the runtime lookup scenarios exercise. The ultimate goal is to present a complete, but a very compact, ESR DB schema.

Service as an entity model

We will start by putting together the basic artifacts, most of which we already identified when counting our lookup entities. It is obvious that everything begins with the Object and that it is the most abstract entity in a hierarchy. Furthermore, we will see that on an enterprise level, we are really dealing with quite a limited number of truly unique things, usually described in the earlier stages of a project's MDA exercises (in UML form). It is also obvious that the transportable (or serializable) form of an object is a message, which is a bit less abstract, but it can still exist as a message. A *message* is commonly described as an XSD. An Object does not exist alone. The Object and its context live and evolve within the application, presented as a set of components and resources interfaced by the API (or contract), which comprise a complex artifact called, in the old EDI times, a "trading partner ".

That's not exactly an SOA term, but it is quite capable of describing an application, application user(s), and API (contract) with related protocol(s) and interchange pattern(s) as a composite entity. A message, representing an application object, will be constructed and propagated when a certain noticeable change in the object's state occurs, denoting an event. This so-called basic or primitive event can be specific to this application, or can be taken in the broader SOA context, that is, it can be an enterprise business event. Event filtering and recognition, message construction, and mediating are usually controlled by a set of rules. These rules play a significant role in every aspect of Service activities and composition's policies. When combined together in a certain order, they represent rulesets.

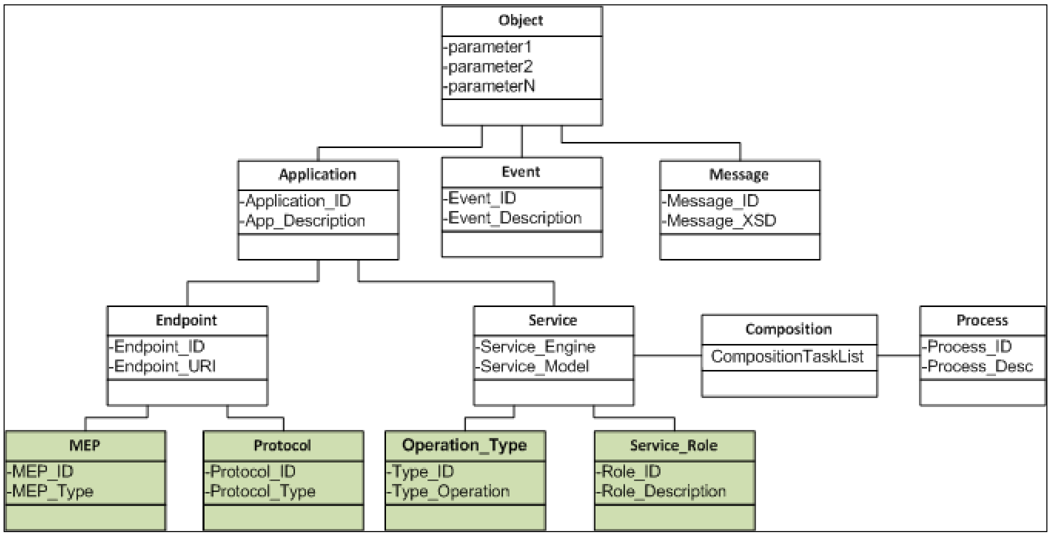
The mentioned services are the essence of the SOA, and through their models and realizations, assume different roles in every framework of the SOA landscape. For this taxonomy, it would be enough to say that the Service/Task is the executable module with a clearly identified API (an API that is presented as a contract is separated for WS and implanted for components) and a service engine responsible for the execution.

In this sense, the term "executable" means that every service or component must support dynamic execution or invocation in the appropriate form—web services must be composable (via the support of a standardized service contract, statelessness, and loose coupling), Java components should support the IoC/dependency injection, PL/SQL packages should be written to support NDS, and so on. The last common abstract thing is the Process—a composition of the previously registered Services/Tasks.

Following the basic rule of the separation of abstract and concrete taxonomy parts (as in the standard WSDL), we can initially assume that these seven entities are abstract, with the following properties as exceptions:

- **Trading Partner:** TP's Application Endpoint URI cannot be abstract. It is usually defined at the late stage of the project.
- **Rules:** As we will discuss later, rules can be registered as functional, XPath-based, and references to the decision tables. In all cases, they cannot be abstract.
- **Services:** Depending on the runtime role, a service can be the ultimate receiver, which makes it the TP's Endpoint, where a URI cannot be abstract. Another property related to the registered Component is an Engine, which must also be concrete.

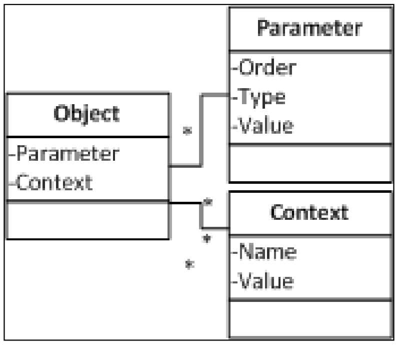
So, the abstract entities' hierarchy can be shaped as follows, where the primary artifacts are presented in white:



To finalize the practical implementation of this semantic, we will go through all the entities and identify all the properties essential for the dynamic service compositions. Further, we will define how these properties can be implemented in a message structure utilized by Service Broker.

Object

In the serialized form, Entity, Business, or Service-related objects present message business payload. The types of Enterprise Business Objects are always quite limited: product, resource, customer, and so on. Information about the transporting Object is one of the mandatory parts of message identification, used for Object filtering, routing, and transformation (the last one naturally shall be avoided). This information must be presented in the `MessageHeader` together with `ObjectContext` for the realization of the Event Identification and State Messaging pattern.

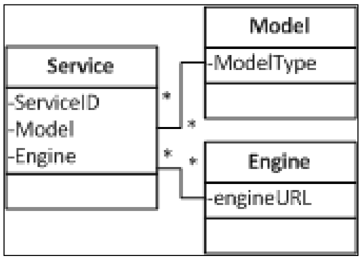


The properties of the object are `OBJECT_ID`, `OBJECT_NAME`, and `UML_REF`.

Service/Task

The most simple part of the taxonomy is defined in SOA terms. We register the executable module that can be a part of the composition (fulfill the role of composition member) and/or something we can dynamically execute (for instance, the rule function). Thus, a role cannot be a property of a service as it's related to a service within the composition. Engine and Model are the Service properties. Part of the design-time service definition is a registration of service event messages, provided to the runtime log. This information will be used for runtime Audit and testing purposes.

Obviously, services can be a part of the bigger composition (SCA), which is also a service, and can be registered in the repository for further reuse.

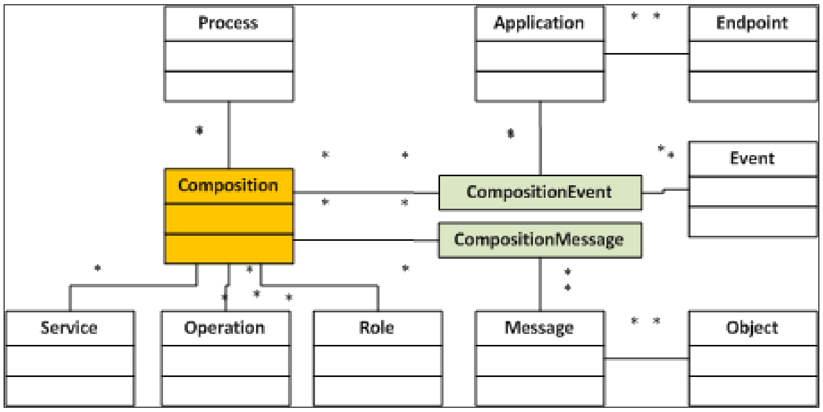


The properties of the Service/Task are `SERVICE_ID`, `SERVICEMODEL_ID`, `ENGINE_ID`, `SERVICE_NAME`, and `SERVICE_DESCRIPTION`.

Composition/Process

Services' composition is actually what we are aiming for. A simple and descriptive representation of the service composition is the main goal of the Service Repository's taxonomy and the foundation for Service Broker.

So, a Process as a Service Composition is a sequenced collection of Services performing the Operations and assuming the Roles in order to fulfill the complex business logic of transporting (with possible transformation) of the Object in the form of a Message between Applications Composition members, presented via public Endpoints (contract, interfaces), initiated by the Business Event, occurred in the Composition Initiator.



This statement pretty much formalizes the XML entity we will further denote as composition Execution Plan (EP).

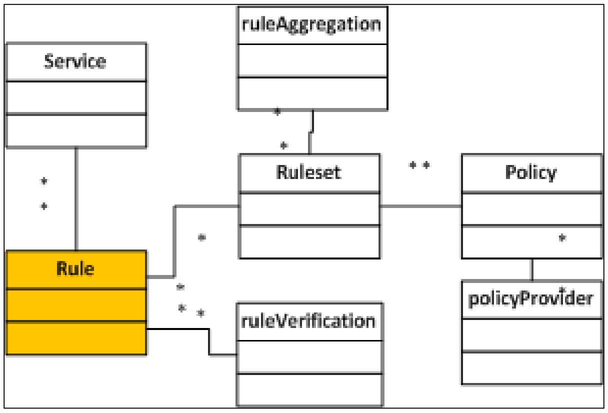
It is also visible from the description of the execution plan that, for its discovery, we need the names of three entities as input parameters:

- Object
- Event
- Service-Composition Initiator

The properties of the Compositon/Process are `COMPOSITIONLIST_ID`, `PROCESS_ID`, `SERVICE_ID`, `TASK_ORDER`, `OPERATIONTYPE_ID`, and `ROLE_ID`.

Rules

Rules are governing conditions used to control the common aspects of service activities. Rules Centralization is one of the core SOA patterns employed to establish the Service Repository in general. However, rules are special because the rule storage realization depends on a particular Rule Engine. Rule Engine (RE) is a special form of Service Engine with very high performance demands, and these demands are usually accommodated by placing rule storage as close to the engine as possible. The most obvious repository realization is on DB, thus making us aware of possible DB rule storages from various vendors. Oracle from DB Version 10*g* has a rule management package, `DBMS_RLMGR`, with rather extensive functionality, although it doesn't mean that our Rule Engine realization will be also on DB. The avoidance of a vendor's lock-in is not always the primary concern. The main problem here is the visibility of defined rules, so we must provide a rule with a name referencing from the Repository to the actual rule storage. The rule reference in the Service Repository leads to the actual ruleset in RE storage and, therefore, is part of the design-time Rules Centralization exercises.



In one possible scenario, Inventory Endpoint using the previously mentioned Message Header's input parameters will recognize the decision table's name and route object to this table for rule evaluation.

RE can be used to find XML EP by name during runtime if EP storage is file-based, but that would not be our first choice.

Interestingly, the requirement to store rulers (rulesets) close to the RE can also be fulfilled from another direction by moving RE toward the storage. Again, it rests on the realization of your particular RE. However, depending on the actual complexity of your business rules, it wouldn't be so difficult to implement a very lightweight, but effective, custom rule engine if open source engines (Drools for JBoss) are not an option for some infrastructural reasons. Detailed realization is the subject of a separate discussion, but the basic idea is quite straightforward:

- Describe the supported rule types: let's say, XPath and Functional.
- XPath-based rules will evaluate the XML object using single Java, `getNodeValue()`, or the XDK function, `valueOf(xmlDoc, xpathRule)`, where the rule is the XPath expression, `(xpathRule)`.
- The Functional rule realization is based on the dependency injection, where a rule's predefined function is executed dynamically for an object's data evaluation. A returned value is compared with the data stored in the decision table.
- Define acceptable rule operations and implement the handler for them (=, >, <, !, =, and so on).
- Define supported rule aggregations (weak and strong, that is AND, OR, respectively) and implement the aggregation handler accordingly.

It is quite obvious that the implementation of Rule Centralization does not require the presence of the single rule engine. Only the rulesets storage will be common (and preferably, ER-based). Following the steps from the preceding bullet points, it is probable, and sometimes desirable, to implement RE in several key points of the infrastructure—in the ESB layer for rule-based routing and invocation and in DB for functional Audit and KPI evaluation. The latter is related to the operational policy and for its monitoring, rules are essential. Thus, we will place the policy entity close to the rules definitions.

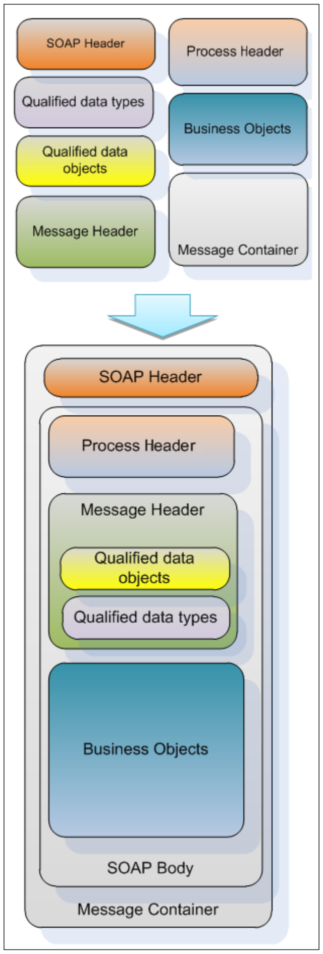
Event

The implementation of an event's taxonomy is also straightforward. We just have to reflect our understanding that the event is the change of an object's state, the object exists within the application (service or composition), and from the whole enterprise application prospective, the event could be seen as a primitive (basic) or enterprise (complex). Events filtering and recognition is the task for the Rule Engine, which is usually inside the northbound adapter layer.

The event is part of the Message Header structure, together with three other mandatory elements, and the key part of composition coordination and context management. The biggest part of this management is apprehended by the message elements and recognition is the task for the rule engine.

Message

A business message is a serialized business object. In terms of Service Brokering, it is presented as an **Enterprise Business Object (EBO)** XSD in the CDM stack. This is an actual payload, but we need to define and implement other essential parts of a Message Container to make agnostic service brokering possible. We have already mentioned two of them: **Message Header (MH)** and the composition execution plan. What other possible moving parts of the Message Container do you see in the following figure?



Message Header (MH) is an SBDH-compliant construct (`http://www.gs1.org/docs/gsmp/xml/sbdh/SBDH_v1_3_Technical_Implementation_Guide_i1.pdf`) with the main purpose of conveying additional metadata about the message or a payload in the message. Originally, from its specification, it can also provide some generic processing information.

To support the object referencing and process coordination parts, we propose to separate the processing construct in the Process Header, which will contain the composition EP as coordination instructions for the Service Broker.

Business Object (EBO), Message Header, and Qualified Data Types (QDT) are subjects of versioning and should have a namespace's postfixes according to individuals' current versions.

In addition to the QDTs, Qualified Data Object (QDO) can be maintained and governed. For example, governing should take control over the distinction of QDT and QDO:

- **QDT**: Examples include Credit Card Data Type, Local Timestamp, Local Date, and Specific ID
- **QDO**: Examples include Address, and so on

> Qualified types and object can also be part of payload (EBO).

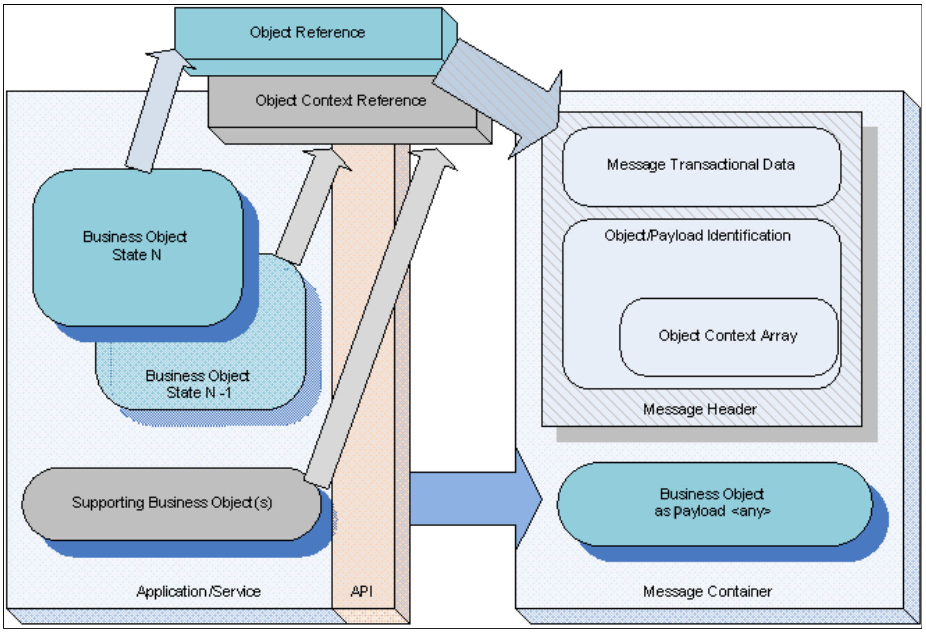
In a Message Container, EBO will be presented as `<any>`:

```
<xsd:element name="Payload">
   <xsd:complexType>
        <xsd:sequence>
             <xsd:any processContents="lax"/>
        </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
```

This type of granularity is definitely negative for security reasons, but as long as we are building SB on the EBF layer, our primary concern is the ability to perform message/service brokering agnostically. The adapter framework, together with the service perimeter guard, will be responsible for message screening, validation, and wrapping ABO/EBO in the container.

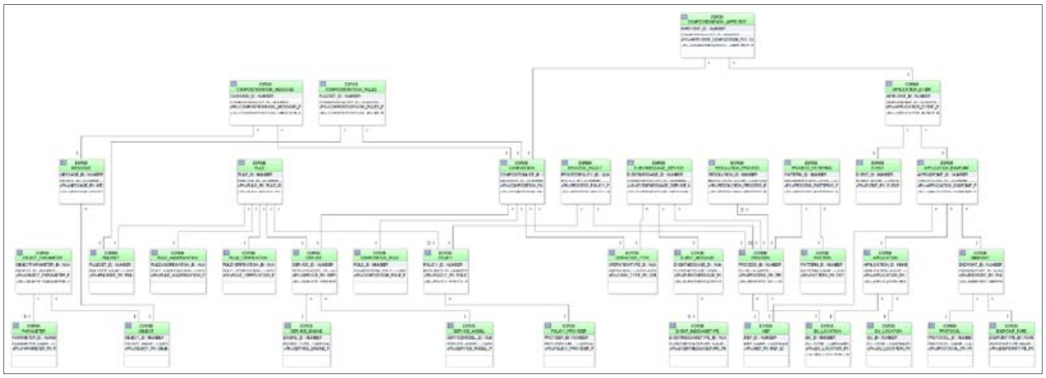
Thus, the application (or service) acting as a composition initiator (the most common role of a trading partner is sender) is not required to construct a message container. It can be done by adapters in the ABCS layer, but all necessary elements for this must be presented in the Message Header:

- The elements include transactional data, timestamps, the object ID, object primary key name, and key data. Basic event identification will be provided in a way an application sender sees it. This will be used for object identification in the Agnostic Message Container.
- To identify a complex business event and possible service composition associated with it, information only about an object is not enough. An object exists in the operational environment in conjunction with other objects and in transit from one state to another. This information is presented in the Message Header as name-value pairs, as the ID previously described in the object-context relation (See the *Object* section).


Object Reference
Object Context Reference
Message Transactional Data
Object/Payload Identification
Object Context Array
Message Header
Business Object State N
Business Object State N -1
Supporting Business Object(s)
Business Object as payload <any>
Application /Service
API
Message Container

The SQL implementation of the service taxonomy (example)

As an outcome of the realization of the service taxonomy, we can now demonstrate the Enterprise Service Repository database schema, which is quite simple but fully functional and capable of supporting runtime and design-time Discoverability for practically any line of business (not only telcommunication). You can use it as an example for your own implementation. Relations between entities are clearly visible and very suitable for exposure by quick JAX-WS Service Inventory Endpoint. But we do not need to do that for our agnostic controller. It will be no problem to modify `ExecutionPlanlooupService` for the extraction of EP information from DB. We will have to add the DB adapter and use it instead of the rule component. The SQL query in the adapter will be based on the view built around the relational model presented earlier, where the composition linking the table is the cornerstone. The view will isolate the `select` statement in the adapter from further development in the Custom ER schema.



Custom SR DB-schema

The XML implementation of Execution Plan

Finally, the Execution Plan (synonymous with Task List or Service Routing Slip) is the XML representation of the service composition we described earlier. It's a sequence of the tasks with Endpoint URLs, where a task can have steps for synchronous MEPs and the definition of Service Engines responsible for task execution (Oracle OSB in the following example).

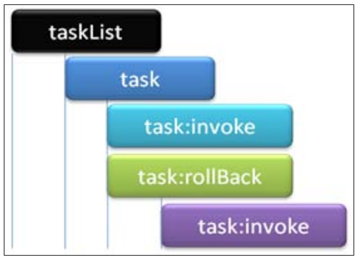
As you can see in the following working example, a list of the actual task's operation is rather limited: invoke and transform. You can add your operation, but this minimal set for Service Broker functionality is quite enough as we do not have any plans to create another BPEL. For error handling, task pairs can have the rollback or compensation part, which will be executed by Service Broker if the response from the first execution is negative. In general, compensation can be another execution plan, that is, if the rollback scenario is complex.

The rule of thumb is simple: rollback tasks can be implemented in the same execution plan. Complex compensation activities shall be combined in a separate execution plan and the master Service Broker will delegate the execution to the subcontroller (another instance of the Service Broker). These operations are controlled by the error handler, which also communicates with the Service Repository to find the error resolution:

```
<?xml version = '1.0' encoding = 'UTF-8'?>
<phs:ExecutionPlan xmlns:phs="urn:com:telco:ctu:la:processheader:v01">
   <phs:taskList>
      <phs:task>
         <phs:invoke taskDomain="OrderFulfillment"
taskName="changeStatus" mep="sync">
            <phs:serviceTask>
               <phs:taskSteps flow="request" technology="OSB">
                  <phs:transform location="Fusion/Xquery/
transformation/ChangeOrderRequest" type="OSB-Resource"/>
                  <phs:invoke protocol="HTTP"
endpoint="http://<physical_address>"/>
               </phs:taskSteps>
```

```
                  <phs:taskSteps flow="response" technology="OSB">
                     <phs:transform location=" Fusion/Xquery/
transformation/ChangeOrderResponse" type="OSB-Resource"/>
                  </phs:taskSteps>
               </phs:serviceTask>
            </phs:invoke>
         </phs:task>
      </phs:taskList>
   </phs:ExecutionPlan>
```

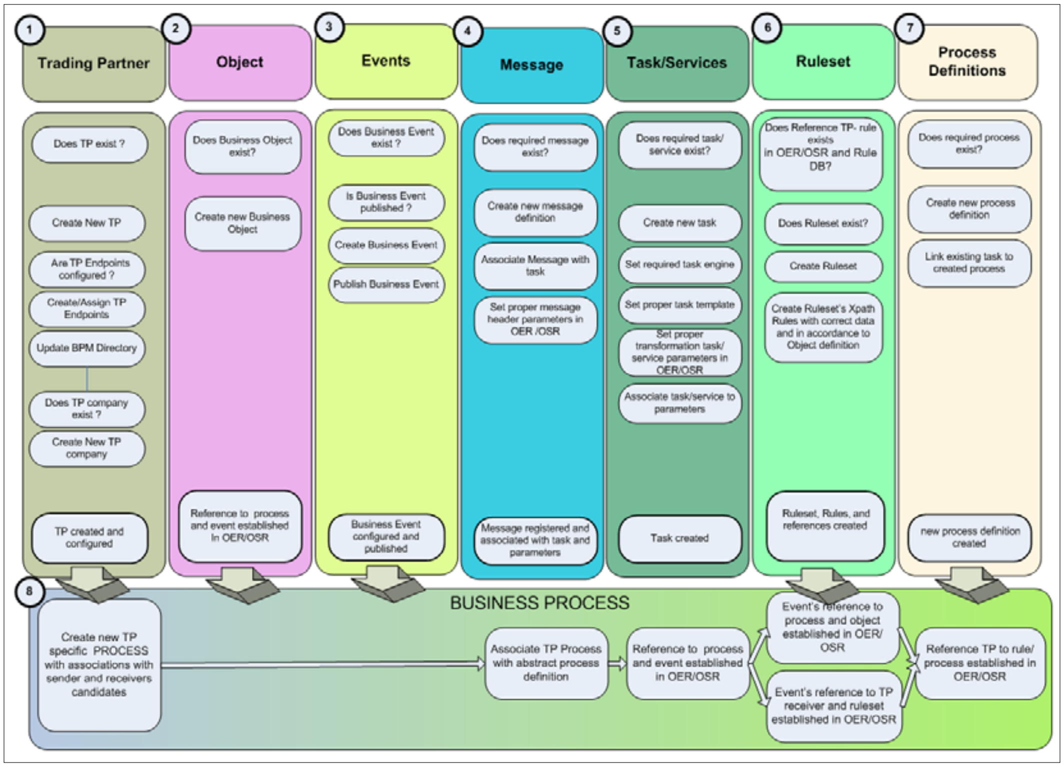
The physical implementation of the Execution Plan is presented in the following figure:



Managing Service Repository

During the runtime Discoverability analysis phase, we mentioned deployment options on which we could roll up our taxonomy model. OSR and OER are obvious, where an asset's definition will be used in OER and tModels in OSR. For OSR-based implementation, we can see a good match for entities such as message and object. Task/Service can also be easily implemented in the case of WS-type, with generic WSDL-tModel mapping. So, we can cover two and a half Runtime Discoverability cases by defining these entities on OSR, and that's not good enough. Together with OER, we will cover all the Discoverability scenarios for all entities we identified earlier; however, whatever deployment option we chose, we will have to follow the taxonomy reference model during the deployment process.

Please see the following figure:


1
Trading Partner
2
Object
3
Events
4
Message
5
Task/Services
6
Ruleset
7
Process Definitions
Does TP exist ?
Create New TP
Are TP Endpoints configured ?
Create/Assign TP Endpoints
Update BPM Directory
Does TP company exist ?
Create New TP company
TP created and configured
Does Business Object exist?
Create new Business Object
Reference to process and event established In OER/OSR
Does Business Event exist ?
Is Business Event published ?
Create Business Event
Publish Business Event
Business Event configured and published
Does required message exist?
Create new message definition
Associate Message with task
Set proper message header parameters in OER /OSR
Message registered and associated with task and parameters
Does required task/ service exist?
Create new task
Set required task engine
Set proper task template
Set proper transformation task/ service parameters in OER/OSR
Associate task/service to parameters
Task created
Does Reference TP- rule exists in OER/OSR and Rule DB?
Does Ruleset exist?
Create Ruleset
Create Ruleset's Xpath Rules with correct data and in accordance to Object definition
Ruleset, Rules, and references created
Does required process exist?
Create new process definition
Link existing task to created process
new process definition created
8
BUSINESS PROCESS
Create new TP specific PROCESS with associations with sender and receivers candidates
Associate TP Process with abstract process definition
Reference to process and event established in OER/OSR
Event's reference to process and object established in OER/ OSR
Event's reference to TP receiver and ruleset established in OER/OSR
Reference TP to rule/ process established in OER/OSR

Maintaining entity relationships in SR

The preceding sequence is based on the custom DB model we presented earlier. However, every entity can be presented as an OER asset or expressed using tModels in OSR. The concrete Business Process (**8**) will be extracted as an XML Execution Plan in `LookupService` using the DB adapter.

1. First, the atomic entity is a Trading Partner, representing the application/service with concrete Endpoint and Endpoints Service Contract. The TP role could be a Sender or Receiver.
2. The TP application is a logic that handles objects, including business objects. All business objects must be registered as Business Entities.
3. Event reflects the change in an object, so we have to maintain a Registry for all the business events.
4. TP, Object, and Event are the key elements of the message identification (via Message Header), and message payload is an EBO carrying a business object.

5. Composition participant, controller, subcontroller, and service providers (TP) perform tasks, expressed as service/components operations. Tasks' activities are based on service engines capabilities and controlled by parameters. Message-related tasks, such as translation or transformation, will require registering message-processing XML artifacts (XSLT, XQuery, Xpath as tModels, artifacts, or references to the file location).
6. All operations are governed by the rules, starting from events recognition (business/nonbusiness or basic/complex) to message routing and composition controller activities. Rules are combined in rulesets and based on tasks. Rule collections are the essence of the policies.
7. Process can be defined as a tasks assembly. In this step, we define it as an abstract process, describing business context, rules, and conditions.
8. Process, associated with TP/Endpoints, is a business process and can be presented as an Execution Plan, or an individual SCA, depending on factors we discussed earlier. A completely descriptive process must be registered as a service for runtime/design-time discovery and dynamic invocation.

Summary

The Governance of the Service Inventory using the Service Repository and Registry is a complex multiphase cyclic process, covering a service's lifecycle aspects—from the cradle to the grave. It's not possible to even scratch the surface regarding most of them in a sixty-page chapter; an entire book is not enough (although, we would like to recommend Thomas Erl's *SOA Governance: Governing Shared Services On-Premise and in the Cloud, Prentice Hall*)! Oracle has the necessary building blocks to cover these requirements:

- A Registry for runtime Discoverability
- A Repository to store all SOA artifacts with their relations and descriptions for design-time Discoverability and comprehensive project management
- The utility for the synchronization of Registry and Repository to compensate tModels' limitations and expand the runtime Discoverability options
- An API and simple UIs to handle and control all the artifacts in Registry and Repository
- Connectors to a developer's tools and application servers for developers and administrators
- Connectors to error handling facilities and business monitoring tools for policy enforcement and the control of service statistics

All of this would be not possible without a clear understanding of an artifact's taxonomy and relations. This chapter demonstrated this aspect and the ways of expressing these taxonomies in the XML forms for the entire message, Message Header, and Process Header (execution plan). The concepts of Domain and Enterprise Inventories patterns were explained from a Discoverability requirements standpoint together with the Cross-Domain Utility layer. As a practical outcome, you were presented with a custom lightweight Service Repository schema, suitable for implementation on OER. We will use this concept further in the next chapters dedicated to the adapter framework and error handling.

6
Finding the Compromise – the Adapter Framework

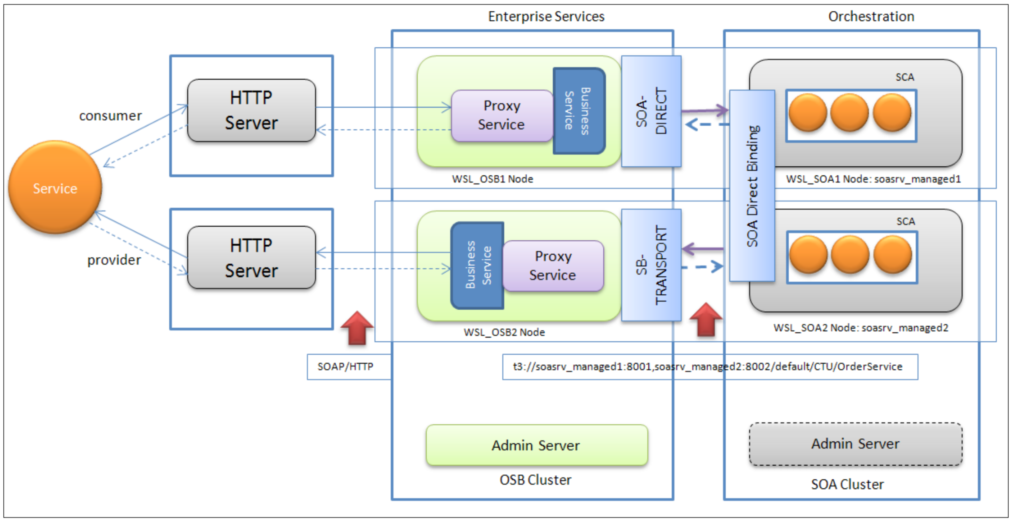
This chapter concludes our discussion on the core compulsory SOA frameworks, presenting the last framework (ABCS according to AIA notations) and all the SOA patterns associated with it. This framework is also optional and probably even more undesirable for pure (theoretical) SOA, but it is still one of the most extensive and heavier layers in most technical infrastructures of modern enterprises. Why? Because of all the legacy applications enterprises have? Let's check our calendar—the concepts of unified standard contracts and canonical APIs (Official Endpoint, Canonical Schema, Canonical Expression, and Canonical Protocol SOA Patterns) weren't invented yesterday; they are more than ten years old and have been cornerstones of contemporary SOA since 1999. With an average lifespan of any core ERP/CRM/SCM bundle in an enterprise of about six years, all that dinosaurs requiring adapters for smooth plugging to the EBS framework (Service Bus) should be as good as gone. So, what are the reasons to keep this layer irrationally thick nowadays? Integration efforts are considerable not only on domain edges, but also within domains, and that raises questions about domain boundaries in general.

How can we minimize our integration efforts and focus mostly on services/components collaboration? Can we avoid building adapters? What SOA Patterns could help us overcome the interfaces disparity? Are there any ways of reusing adapters? We will try to cover these and some other questions in this chapter. We will continue developing the transport adapter framework attached to the Adapter Factory, as discussed in *Chapter 4, From Traditional Integration to Composition – Enterprise Business Services*. We will also touch upon more traditional ways of establishing adapters using BPEL/SCA, but our main objective is to demonstrate some ways of making applications more service-oriented in order to avoid creating extensive Adapter frameworks, and here we will utilize some service metadata taxonomies developed in the previous chapter.

Optimizing the Adapter Framework

The best adapter is the one you do not have to implement. The end. Ah, if only we could get rid of them so easily. Requirements for an adapter within a domain usually signify that something in your service inventory went wrong and you overlooked the discrepancies in your data models, formats, and/or messaging/transport protocols. Regarding protocols, you could actually anticipate that a single protocol would not be enough and the Dual Protocol SOA Pattern (`http://soapatterns.org/design_patterns/dual_protocols`) can be justified in the cases explained next.

Your service activities on both the north and south sides are the canonical SOAP over HTTP, but between servers, handling every individual layer of your SOA frameworks (`ABCS<->EBS`, `EBS<->EBF`, and `EBS<->EBS`), you would like to have something faster, without XML processing overhead. In this case, the RMI-type protocols could be the optimal choice, such as `iiop/iiops` or `t3/t3s`. While `iiop` is used by the a CORBA-enabled Java clients and requires CORBA naming context, the `t3` protocol is the internal WebLogic protocol for Java-to-Java connections. In fact, this protocol from the very beginning was the essence of Application Server's interoperability, even before it became WebLogic. This protocol is quite fast and supports all interoperability features such as Load Balancing, including complex Load Balancing algorithms (Weight-Based, Random). For this scenario, you can actually build services with contracts supporting only one canonical SOAP over HTTP and let WLS handle the `t3` protocol. Thanks to Oracle, most of the job can be done through the configuration. The `t3` protocol is a foundation for the SOA-Direct protocol and SB-Transport protocol used in SOA Direct Binding, and the difference between them is explained in the following figure:

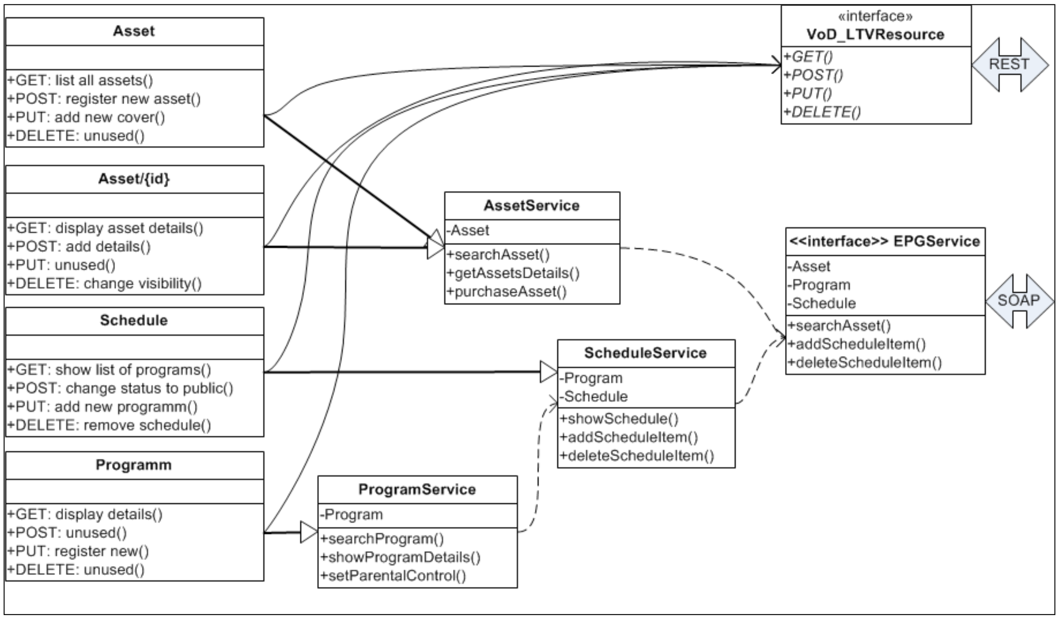


You have both SOAP and non-SOAP clients, which prefer a plain HTTP protocol for several reasons; one of those is external Load Balancers, where just HTTP is the best. Another is the most obvious one—the service consumer prefers plain HTTP. Thus, you will need to handle both the transport and messaging protocol, and you could do it on the service's side, maintaining two contracts through the application of the Concurrent Contract SOA Pattern: one for SOAP, where you consequently use JAXB (discussed earlier in *Chapter 2*, *An Introduction to Oracle Fusion – a Solid Foundation for Service Inventory*) and SOAP wrapping for the object, and another where you just do the O/X Mapping (again using JAXB or the Spring framework), and then the classic HTTP Post routines—prepare request, set header parameters, open connection, make the post, read response, and so on. What is important for us here is that we always have to anticipate these requirements in our atomic service design and prepare placeholders for X/O Mappers and basic inner adapters in the form of a Service Facade SOA Pattern.

This facade will sit between your contract and actual component representing service logic. In fact, that's what you get on applying REST/SOAP technologies from most mature marshaling/unmarshaling frameworks (annotations in POJO, Spring, JiBX, and so on). However, what if your service cannot support facade injections or if the variety of required protocols is a bit wider than two (out of the scope of the Dual Protocol pattern)? Well, then a traditional way to handle this is to present the adapter layer in ABCS and handle it through the application of the Protocol Bridging SOA Pattern. That's the classic integration approach with transformation service agents in the ABCS framework. By the way, please note that the Concurrent Contract is always based on the application of the Service Facade. Also, just for the sake of SOA exam preparation (S.90.xx), keep in mind that Facade in SOA terms is something present *inside* the Service.

What if you have the HTTP as your solid Canonical Protocol for transport, but your messaging protocols can be SOAP over HTTP, plain HTTP(s), and HTTP REST-style? Yes, that's quite a common situation when some clients would like to have unified protocol operations (`POST`, `GET`, `PUT`, `DELETE`), and others prefer more meaningful operations, expressed in WSDL.

That's a really interesting issue; please see the following figure:



Why would you need that? For example, in the context of telecommunication primes which we discussed in *Chapter 3*, *Building the Core – Enterprise Business Flows*, we are quite aware of the common trends in current video entertainment options, which are listed as follows:

- **Video on Demand** (**VoD**) and classic Linear TV should not be seen as separate delivery channels, but presented as a combined media resource with consolidated assets for live (or near-live) programs and on-demand sources. These assets should be accessible through a unified entry point, suitable for all types of assets/channels (YouTube, HBO, live programs, and so on). Naturally, all services handling these sources have different messaging protocols such as REST, SOAP, and proprietary.
- Inside the telecom organization, we also have services with different messaging protocols due to their technical nature. Services related to the STBs operations (the small black box under your TV) are RESTful in general, but some of them are also parts of Billing and Order Fulfillment, which are traditionally SOAP-based. The telecom business domain is not exceptional; you can see it everywhere.

With the modern "**Any** content on **Any** device, **Anywhere**" paradigm, telecom providers should ensure a seamless experience on TV and tablet/mobile devices or any device for the same content. Thus, our APIs must be universal enough to handle content on managed and unmanaged devices, supporting REST and SOAP simultaneously (some call it Triple A, but it fact, we have a fourth A: **Anytime**).

Arguably one of the most complex APIs in telecom for all types of media resources is a consolidated search. Your client could watch *The Bourne Identity* on his tablet and at some moment decide to figure out how many Austin Minis were destroyed while shooting a single chase scene on the stairways in Paris or who else in the history of a spy's paperback lost his memory and got conscience convulsions instead. You must instantly provide him with options to purchase the subscription to Top Gear where Jeremy Clarkson will teach your client how to drive a Morris Marina or purchase the *The Spy Who Came In From the Cold* movie from HBO Go (or an audio book from Amazon instead, depending on your business affiliations). What could be easier, some say. Those of us who hit some REST limitations wouldn't be so eager though. How many query parameters can we put in a browser's search string? What is the maximum length of the query string? How can we parse the complex query string with plural parameter combinations? Indeed, some of these questions can be solved by the Google YouTube Search API, for instance, but it would be much better and easier to use classic SOAP/XML to construct the query XML object. You will not have limitations on the query string, at least.

Generally speaking, the mapping of standard HTTP methods (`POST`, `GET`, and so on) to meaningful business operations was never easy and can be done more or less directly only for Entity Services (Recourses in that matter), where `PUT->add`, `DELETE->del`, `GET->get`, and so on. For more complex cases, you will have to do some sort of "triangulation", where the meaning of your operation will depend on the combination of three things—the resource instance, data type, and basic operation. Yes, you could have as many combinations as you want, but please do not forget to put it with utmost diligence into the repository from the previous chapter and make it public; otherwise, discoverability and interpretability of your service will be just a little less than nothing. And that's the common rule for all the RESTful services, as they do not present formally discoverable contracts and require the resource hierarchy with clear meaning of operations.

> Just in case if someone got an idea—the first and foremost thing to remember is that we do not want to start pointless discussions such as SOAP versus REST. We never participate in such discussion and we use them proportionally in all technical infrastructures where necessary. We are just pointing out that REST simplicity has a price and some declarations such as "SOAP is dead, all migrate to REST!" are over-exuberant at least.
>
> We truly admire the knowledge, style, and presentation skills of Craig Larman (`http://www.craiglarman.com/wiki/index.php?title=Main_Page`), one of the most influential Agile Experts and his ways of making jokes. Although he claims that his jokes lack humor, we think his jokes are brilliant. Instead, we can try one bad joke here and we hope that you do not read this book to your kids before sleep. When a beautiful young Princess was born, the evil witch predicted that the Princess would die of pricking her finger on her 15th birthday. So, to save his daughter's life, the good King ordered to cut off all the Princess' fingers.
>
> Simplicity of the REST services came from the visualization of linking resources in an Internet style, based on a unified protocol with a unified (and highly limited) set of commands. Cutting lots of SOAP functionalities initially gave the REST concept a considerable boost. Now, when REST services are facing similar challenges as SOAP, simplicity is not that obvious. Actually, the same thing we can say about OAuth 2.0 (half-dead, half-alive), and even about SOAP itself. Look, it's not simple, it's not exactly about the object, it's not only about the access, and it's not truly the protocol.

And speaking about the mapping of basic HTTP methods a in REST services to business procedures, we should mention that situation when `DELETE` in fact could mean `CREATE` in business terms is very common and that can be quite confusing.

So, where will we put all this bridging? That would be the same approach as in the previous point. Plan for facade from the very beginning. Implement two mappers for the same component: one for REST service implementation and another for Web Services. Obviously, in most IDEs, all it takes is a right-click, but be careful with the code-first approach. Luckily, contract first is also supported.

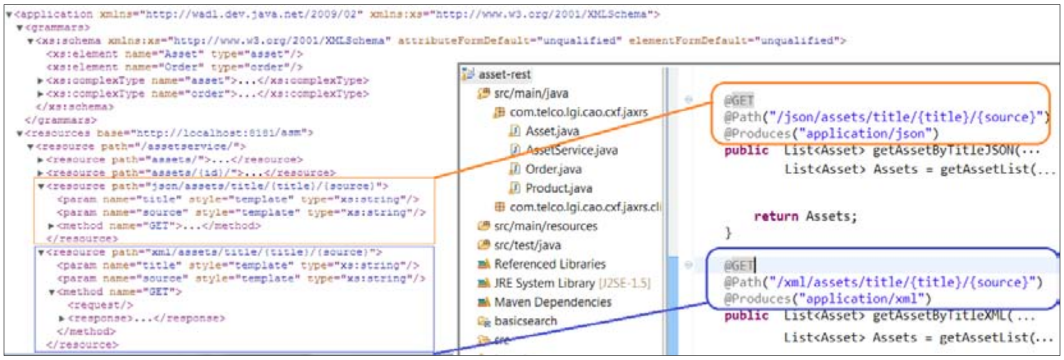
Draw the Service contract (WSDL) as you see it, generate the code for the component, and then build REST from this component, exposing the required operations using JAX-RS (as one possible option). The second option would be to put the `REST<->SOAP` conversion on the Adapter framework and OSB.

This solution also can be perfectly justified; for instance, the search of Video Asset can be based on many search engines and API (both internal and external): the YouTube API, IMDb, RottenTomatos, and your internal metadata store. You would need the adapter if you will have to do the Transformation/Translation/Bridging. If all your APIs are RESTful and you can extract data elements according to your Video Asset EBO, you can do this kind of aggregation on OSB. In the case of any transformation, consider the presence of the Adapter framework. All core patterns for the Proxy service were discussed in *Chapter 4*, *From Traditional Integration to Composition – Enterprise Business Services*.

Remember, this type of aggregation doesn't make this service task-orchestrated because it remains totally business agnostic. It is also synchronous and we do not need any kind of state deferral from the BPEL engine. Thus, placing this type to BPEL/SCA will definitely be a mistake making the service slow and unreliable (more infrastructural elements that necessarily do not improve reliability). In general, individual requestors are interfaces to different search APIs and can be presented as separate EJBs/POJOs and then you can assemble them on OSB using the JEJB transport. The JEJB proxy service serves as a stateless session bean to the EJB client interface.

The next logical outcome from the previous point would be the JSON/XML conversion for the same type of **Enterprise Business Message** (**EBM**). Yes, this is not really a Protocol Bridging; it's a Data Format Transformation SOA pattern as we have different formats to deal with. Again, transformation can be avoided if the Concurrent Contract SOA pattern is observed at the early stages of service design. Indeed, that's quite common (and prudent) to have services designed for XML/JSON messaging format support and in case of REST services, such as `http://ctu.com/services/v2013/fusion/ltv/json/Channels/sort/LogicalNumber` or `http://ctu.com/services/v2013/fusion/ltv/xml/ Channels/sort/LogicalNumber`.

We have plenty of libraries and frameworks to the marshal/unmarshal Object to XML/JSON as mentioned earlier, and JAX-RS is one of the most powerful framework to do the job; please see the following figure:



```
▼<application xmlns="http://wadl.dev.java.net/2009/02" xmlns:xs="http://www.w3.org/2001/XMLSchema">
 ▼<grammars>
  ▼<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" attributeFormDefault="unqualified" elementFormDefault="unqualified">
     <xs:element name="Asset" type="asset"/>
     <xs:element name="Order" type="order"/>
   ▶<xs:complexType name="asset">...</xs:complexType>
   ▶<xs:complexType name="order">...</xs:complexType>
   </xs:schema>
  </grammars>
 ▼<resources base="http://localhost:8181/asm">
  ▼<resource path="/assetservice/">
   ▶<resource path="assets/">...</resource>
   ▶<resource path="assets/{id}/">...</resource>
   ▼<resource path="json/assets/title/{title}/{source}">
      <param name="title" style="template" type="xs:string"/>
      <param name="source" style="template" type="xs:string"/>
    ▶<method name="GET">...</method>
    </resource>
   ▼<resource path="xml/assets/title/{title}/{source}">
      <param name="title" style="template" type="xs:string"/>
      <param name="source" style="template" type="xs:string"/>
    ▼<method name="GET">
       <request/>
     ▶<response>...</response>
      </method>
    </resource>
```

```
@GET
@Path("/json/assets/title/{title}/{source}")
@Produces("application/json")
public  List<Asset> getAssetByTitleJSON(...
        List<Asset> Assets = getAssetList(...


    return Assets;
}

@GET
@Path("/xml/assets/title/{title}/{source}")
@Produces("application/xml")
public  List<Asset> getAssetByTitleXML(...
        List<Asset> Assets = getAssetList(...
```

Avoiding the service's Data Format Transformation using annotations

We have the `Video Asset` class exposed as a REST service. As you can see, we have REST WADL describing how the same operation "Get Asset by Title from particular Source" exposed through the Service Bus can be linked to two functions-wrappers with different annotations: `@Path` and `@Produces`. Behind it we have the same `getAssetList`. This is about the `GET` method, but what about `POST`? The annotation in this case will be `@Consumes` and you can combine the `json` and `xml` formats in one annotation. Now, `Client` should include an appropriate content-type header in his `POST` request to be properly handled. For more information about JAX-RS and JAXB, please see the Oracle documentation at `http://docs.oracle.com/javaee/6/tutorial/doc/gkknj.html`.

In addition to that, Google supplied us with Gson to do simple serialization with JSON. (We are using it currently in our projects at the moment of writing. Feel free to use Jackson instead. Don't forget to look at YQL from Yahoo REST API Console; it's very ingenious.) For a quick demonstration, let's just return to the media Search service we mentioned earlier and focus only on one external API from IMDb. The following REST API will return JSON containing all records related to the Bourne Identity request.

> The search URL of the Bourne Identity request is `http://www.imdb.com/xml/find?json=1&nr=1&tt=on&q=Bourne%20Identity`.

Casting it to the object will require just a couple of lines of code, shown as follows:

```
public SearchObject getSearchResultObject() throws Exception {
    try{
        InputStream source = getResultStream(search_url);
        Reader reader = new InputStreamReader(source);
        Gson gson = new Gson();
        SearchObject response = gson.fromJson(reader,
          SearchObject.class);
        ...
        reader.close();
        return response;
    }
    catch (Exception e) {
        log.error(getClass().getSimpleName(), "Error for URL "
          + search_url, e);
    }
    return null;
}
```

Surely `SearchObject` is completely based on the IMDb JSON response. Constructing it will also require no time, thanks to the Google library, `jsonschema2pojo`. Just install it, save the IMDb JSON in a file, and do the conversion (use your own parameters if needed), using the following command line:

```
jsonschema2pojo --source imdb.json --target java-gen -R -a GSON -T JSON
```

You will find a complete class hierarchy in the `java-gen` folder, suitable for consumption by Gson. Well, not exactly suitable, to be honest. What we just demonstrated is *not* exactly the code-first approach, as we have generated classes *from* the message, but this is *not* a Canonical Schema (Message) by any standard, and anyone who looked at the Google YouTube JSON, for example, knows how many "specific" things are in there. IMDb is not an exception. Without compromising the classes' structure and hierarchy, you really can make POJOs more logical and compact, and we advise you to do that. Anyway, the previous code will work right away, but is not advisable from the SOA's standpoint.

To complete the following example, we demonstrate the `getResultStream` function. It's entirely based on the Apache HTTP client and exceedingly simple (thanks to the Apache community):

```
....
DefaultHttpClient client = new DefaultHttpClient();
```

```
        HttpGet getRequest = new HttpGet(url);

        try {
            HttpResponse getResponse = client.execute(getRequest);
            final int statusCode =
              getResponse.getStatusLine().getStatusCode();
            if (statusCode != HttpStatus.SC_OK) {
                log.error(getClass().getSimpleName(), "Error " +
                 statusCode + " for URL " + url);
                return null;
            }
            HttpEntity getResponseEntity = getResponse.getEntity();
            return getResponseEntity.getContent();
        }

        catch (IOException e){…}
```

You can expose this service using OSB now as shown in the previous screenshot. Job done, no Protocol Bridging of Data Format Transformation required for this simple implementation.

As you can imagine, a complete enterprise-ready implementation will require a little more effort and another common adapter-related pattern will be necessary. What we will need to do is:

1. Implement URL Builder for constructing the REST query string accepting more than one parameter (for example, `title`, `List<sources>`, `actor`, `rating`, `num_hits_returned`).
2. Implement SearchObject (ABO) for every external resource/application.
3. Validate canonical SearchObject (EBO) for internal resources.
4. Implement casting of individual ABOs to the consolidated EBO.
5. Finally, expose consolidated services on ESB (OSB), for JSON and XML.

Step four signifies that the Data Model Transformation SOA pattern (`ABO->EBO`) should be used.

Traditionally, in a classic BPEL-based Adapter framework for SOAP services with XML payload, XSLT transformation is the easiest choice. With visual mapping and a drag-and-drop approach, you can achieve desirable results in minutes for simple cases. Regretfully, complex cases would require manual coding, and from a performance standpoint, XSLT is probably the slowest.

Naturally, complex coding in XSLT will make it impossible to be seen in Visual Designer. One can argue that complex coding is hardly possible in XSLT and, anyway, the temptation to do Java Callouts will be incredibly strong. By the way, the JSON payload will require some manual coding anyway. Considering this, our architectural directions could be as follows:

- Expose every individual's Search Services via the Adapter framework to the Service Bus and aggregate service calls to different sources on OSB. Only one service will be exposed to the customers; the result will be provided in the canonical form (XML/JSON formats). The solution's modularity will be quite high, as every source will be individually wrapped. Performance will not be at the best, as we will have to do lots of serialization/deserialization and data model conversions. The Adapter framework could be quite heavy.
- Still, only one service will be exposed, but instead of wrapping every individual source into an adapter and assembly on OSB, we will do all object-to-object mapping behind the main service, presenting custom searches as separate EJBs or POJOs. This approach could be suitable when performance requirements are high, the number of individual sources are limited and/or static (due to business agreements with external providers), and when aggregation logic is quite complex; you will have to put it into a separate Java module anyway. You remember the rule from *Chapter 4*, *From Traditional Integration to Composition – Enterprise Business Services*, dedicated to Oracle Enterprise Service Bus—do not place complex orchestration/service collaboration logic on OSB!

  Actually, from a modularity prospective, both approaches are pretty much identical. The second approach also can be quite modular with a proper Java classes design. It also can be swift, as we will not leave Java premises and do all mappings on a low level. Spring 3 Object Mapping is a very good choice (see `http://static.springsource.org/spring/previews/mapping.html`), but we have lots of other libraries from Apache, such as Commons-Lang and Commons-Convert, ModelMapper (`http://modelmapper.org/`), and Dozer for JavaBeans. This list is not complete. Therefore, with proper placing of the SOA patterns, the ABCS layer still can be very thin and manageable.

- And finally, Protocol Bridging on OSB will be simply inevitable if we have services in our inventory communicating synchronously and asynchronously (via queues). In this case, the most logical approach is to establish OSB Service Broker as we did in *Chapter 4*, *From Traditional Integration to Composition – Enterprise Business Services* with some Adapter framework around it.

> For the SOA exam preparation (S.90.xx), please keep in mind that Data Format Transformation, Data Model Transformation, and Protocol Bridging SOA patterns are the parts of Service Broker compound patterns, which is an essential part of ESB implementation. You also should remember that you can apply any of these patterns at any layer of your SOA infrastructure, but necessity of this application signifies that your design is not really optimal.

We can come to a conclusion that after a quick walk-through of these of these use cases (five in total) described previously. We can clearly see that Protocol Bridging pattern will be highly demanded only in one of them. With the implementation of the Canonical Schema SOA pattern, Data Model Transformation can be avoided and Data Format Transformation can be solved on the service side. All of this will minimize our integration efforts and help establish a more predictable and reliable Service Inventory, well suited for long- and short-running compositions.

How can you achieve this? Sorry, the answer is quite non-technical and has only one word—Governance. You, as an architect, are the Mayor of this SOA metropolis and it's you who is responsible for establishing policies and watchdogs around the crucial elements of your SOA infrastructure and in context of the Adapter framework— the Federated Endpoint Layer. This compound pattern is a result of the combination of the following patterns, which we discussed earlier:

- **Canonical Protocol**: Clearly, it's impossible to have only one protocol (transport and messaging) in our Service Inventory. From the previous use cases, you can expect that at least three will be required. Direct binding can be achieved through configuration and will not require (significant) efforts for bridging. Watch for others two and do not let them spread around. You can mitigate the risks by promoting the Service Facade and Concurrent Contracts at earlier stages of the service design.
- **Canonical Schema**: Establish the EBO model and EBM schemas as your first step. Stay alert about any amendments in XSDs.
- **Canonical Expression**: Operations such as `doTask` and `getOrdforPrevRepPeriodInVideoDomainperCountry` are two extreme cases of expressing service capabilities and neither is consistent or interpretable. In *Chapter 1*, *SOA Ecosystem – Interconnected Principles, Patterns, and Frameworks*, we mentioned that the Java naming convention and XML naming standards documentation could be a good start for establishing a proper naming system.

- **Service Normalization**: Presence of redundant logic should be strictly justified and generally avoided. Services should not have overlapping functional boundaries, and if you have the same services working in a cluster for the reliability and performance reasons, it means that you have a Redundant Implementation SOA pattern in place, not a denormalized inventory. Constantly watch for new services delivered by different project teams. Prevent any attempts to implement "another version of that service, just a bit faster and better suited for our (projects) compositions". Almost immediately after implementation, you will have to integrate this "better service" in your service domain/enterprise and migrate customers.
- **Official Endpoint**: This is another Compound pattern, based on Logic Centralization and Contract Centralization. Logic Centralization seems similar to the Service Normalization pattern, but generally focusing on gathering agnostic logic in atomic units of work and presenting them as unique blocks for reuse. The Contract Centralization pattern in this case will guarantee the application of the Loose Coupling principle and prevent negative coupling. Simply put, in order to keep our inventory normalized, we will avoid building hybrid services with partial/incomplete/mixed logic and strictly prevent access to service resources bypassing service contract.

Thus, SOA methodology supplies us with all the necessary means to minimize the Adapter framework in order to reduce integration efforts. We believe that some technical demonstration will help you understand ways of service, "canonicalization" in this sense. Arguably, the most common adapters in our Adapter layer are DB-Adapters, so, probably we should concentrate our attention on them first in our examples.

Also, we would like to demonstrate how to use the elements of the custom service repository we discussed in the previous chapter in our adapters design, mostly because of its flexible taxonomy. Our intention is not really to build the Adapter layer, but find ways to get rid of it. Therefore, for the next technical exercise, we would like to put CTU Telecom aside for the moment (we will return to it soon, discussing Error Handling frameworks) and select DB-centric Enterprise with lots of business logic concentrated in PL/SQL packages and a huge integration layer based on different BPEL processes, where most of the flows are DB-adapters.

Because of the large number of the Oracle E-Business Suite installations around the globe, it's not that difficult to picture such an enterprise, and this time it will be the biggest Scandinavian logistic operator, Reindeer Rudolph Delivery AS, the corporate moving all kinds of goods around Northern Europe, from post and parcels to IKEA products (again, the company is counterfeit).

Another reason for selecting this type of primer is Oracle's SOA products history, which we have discussed in *Chapter 2, An Introduction to Oracle Fusion – a Solid Foundation for Service Inventory*. From the table in the *The Oracle SOA development roadmap – past, present, and future* section in *Chapter 2, An Introduction to Oracle Fusion – a Solid Foundation for Service Inventory*, we can see that the first revolutionary step in Oracle SOA adoption was in 1998-99 with the introduction of Oracle 8*i* and a full-fledged XML Developer Kit (XDK, latest version `http://www.oracle.com/technetwork/developer-tools/xmldevkit/index.html?ssSourceSiteId=opn`). Together with other two (Collaxa and BEA acquisitions), it was truly a cornerstone for all the SOA portfolios (acquisition of Sun/Java was just logical continuation in this direction). Using this opportunity, we would like to express our deep appreciation to Steve Muench for his efforts in composing one of the best SOA books of the earlier SOA days (2000).

> Many DB versions changed, lots of XDKs come and go, and some methods demonstrated by Steve Muench are quite obsolete (starting from 9*i*) nowadays, but an elegant architectural approach, clear vision of optimal solutions, and good balance in the tools selection (Java, PL/SQL) could serve as a good illustration of the rational design and we will use it in our following example. Surely, 11*g* DB and modern XDK functionality will be preferred. After all, it doesn't matter if it is modern or not, all that matters is if it works according to our principles or not. The color of the cat does not matter, as long as it catches the mice.

Logistic primer

The following example is based on the sequence of SOA and integration projects undertaken over several years on one of the oldest Scandinavian Company, aimed to improve costs savings and operational agility. The details about the company and the project undertaken are explained in the following sections.

Basic facts about the company

Some of the details about the company and its market position are as follows:

Logistic operator	**Business domain**	**Governing and type of ownership**	**Number of employees**
Reindeer Rudolph Delivery (RRD) AS	Logistic and distribution with more than 300 years of service	State owned/ Government	20,000

RRD history

RRD is a well-established big logistic operator, freight forwarder, and parcel delivery service, operating across Scandinavia and Northern Europe. Its main assets are cargo transportations units, cargo and transshipment terminals and collection and distribution centers.

All operations are very centralized and governed from a single operation center (Stockholm HQ). Development and implementation tasks are under the control of the local IT department with high level of skills in centralization. Although up to 50 percent of implementations are done by internal IT forces, the presence of external vendors, carrying on development, is substantial due to the magnitude of ongoing tasks.

Technical infrastructure and automation environment

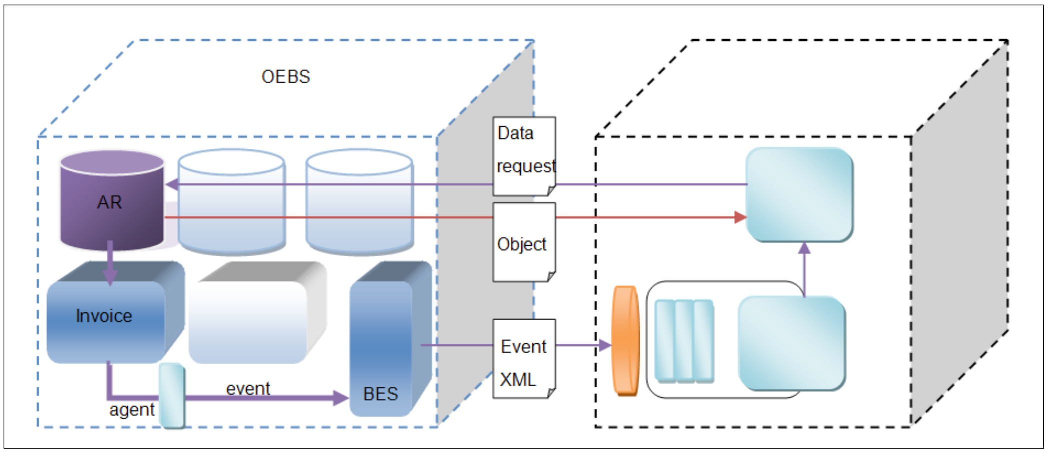
The application landscape just reflects the governance model as all core applications are implemented by the centralized Oracle E-Business Suite bundle running on very beefy hardware (a modern mainframe), on the HA topology.

A very extensive Adapter framework is employed in order to handle a sizeable amount of remote service consumers and service providers. In order to handle a desirable level of concurrency, asynchronous interchange patterns are considered as a primary way of communications. The Oracle E-Business Suite is the main source of business events, which are identified and distributed for message construction and processing.

The challenges here can be described as follows:

- Using the highest level of logic centralization in one application suite provides a desirable level of flexibility for all concurrent vendor's teams to implement independent solution logic without disrupting each other's work
- Make sure that adapters and message transformation modules will not become a bottleneck for the performance
- Considering the first two points, it becomes clear that implementation of the Event-Driven Network is one of the strategic directions for RRDs service-orientation

The **GetAResInvData** service (account receivable invoice data) is one of the core services exposed for external systems. The following figure shows the service architecture:



Oracle EBS events handling sequence

The existing design specification for this service describes the parts of this service architecture, as follows:

- The Oracle E-Business Suite is a unified and centralized invoice handling system (among others, functions such as Account Payable, Account Receivable, and OM), presenting a solid ERP, and thus, GetAResInvData is the single access point for the invoice data from the middleware side towards the RP.
- The **Account Receivable** (**AR**) part of OEBS is concurrently accessed by many internal modules, including the frontend.
- Every invoice object alteration in an AR domain is controlled by service agents, propagating change state to **Business Event System** (**BES**) for event recognition and filtering. The BES validates the type of event and helps constructing the Event XML message, if the event is business, as well as message-oriented.
- A very small XML message with the event data is sent to the event subscriber, which is a part of the message-oriented middleware.
- Service-subscriber pass the event particulars to the data-extraction module, which queried AR DB for the invoice object.
- The Invoice Object is extracted from OEBS and transformed in the required format.

Initial analysis

Something can be spotted immediately such as the service name(s). Yes, it's quite far from canonical by any standard. Minor thing, you say? Unfortunately not. As mentioned, a lot of vendors and solution providers are concurrently implementing different projects using centralized OEBS instance and an Adapter framework around it. How can we be sure that the service purpose and context can be unambiguously understood and then properly used? Regretfully, a strict governance policy and centralized control weren't established the best from the very beginning of Service Inventory creation, which opens the door for redundant logic implementation and service denormalization in general. This fact affected the Adapter frameworks quite severely, leading to multiple adapters of the same nature and hybrid services instead of task-orchestrated services in a right block of figure in the *Optimizing the Adapter Framework* section.

External programmers and Trading Partners became confused by the magnitude of interfaces and business operations offered in unstructured and unclear ways, which inevitably leads to a strong desire for most of them (internal TPs, developed and controlled by solution providers) to get direct access to underlying OEBS resources.

This is not the worst yet. We already mentioned the Oracle Business Event System as a core component for handling business events in OEBS, acting as a subscribe-publishing mechanism to all consumers for all business events occurred in any business domain. By default, BES has more than 1,000 seeded events, preconfigured for most common business cases per domain. Generally, it gives you a great flexibility to propagate the Event message in order to trigger any process in OFM or run any custom module from the custom schema. The configuration of events and event subscribers is relatively simple and can be done in a few clicks using the OEBS Admin console. The problem here is that without proper guidance, each vendor (such as IBM, Accenture, Capgemini, and lots of others) who is involved in project delivery in RRD has their own understanding (or misunderstanding) of business events and their relations. Some might think that there is no required event or no proper action for the event and so on. Even with a common understanding of the concept, there is enormous temptation to make a shortcut under the pressure of a tight project schedule. And the result is hundred lines of code (literally!) in dozens of triggers, associated with main business tables in each domain. Even more—lots of weird events registered in BES, connected to dead or malfunctioning business procedures. After a year of such implementation, the BES finally failed completely as a module and as a concept and RRD administrators just shut it down.

Still, most of it can be considered as a Governance issue and lack of policy enforcement, especially from the ownership prospective. There is nothing bad related to adapters implementation, the topic of this chapter. Or is there?

Let's look at the sequence of messages exchange logic again as follows:

- First, an event is detected. Event subscribers are recognized and the event message is issued to all recipients through the common channel (AQ in most cases).
- Recipients recognize the event and activate the object extraction though the standard OEBS API which we mentioned in *Chapter 1, SOA Ecosystem – Interconnected Principles, Patterns, and Frameworks*.
- For complex business cases, one extraction is not enough, we have to enrich **extracted business objects** (**EBO**) several times or build a custom procedure in an XXCU custom schema for complex data aggregation and use a standard DB adapter for that.
- We will process the business object and return the completion status back to OEBS.

What can we say about the classic Hollywood principle here ("don't call us, we'll call you")? Well, apparently, if we have more than one call here, the principle is entirely broken. Someone is calling us and asking to call back. So much for low coupling and high cohesion, isn't it? We have a callback MEP where it's not really required. Try to imagine this type of interchange applied in stock exchange and utilized by stock trading robots: the robot will receive notification about an index change (time) and then make a decision for index data extraction (some more time) and after that do the actual extraction (and more time). Are we going long or short? No matter what, money will be lost anyway.

You can say that this case is kind of extreme; some latency will exist anyway in any system, and such rapid response is not really required for RRD business cases (we use AQs after all). However, the main question remains—why not provide EBM for the actual object itself in the first place, not just events notification? We've been talking about schema standardization quite a lot, but here it seems only the event message is standardized in the `APP_WF_EVENT_T.xsd` schema. The main reason for that was mentioned earlier—lack of governance, which resulted in many versions of the same EBM.

In fact, they are ABMs with no centralized EBO. The object generally provided by the standard OEBS API is disregarded as too complex or vague. Most of the processes handling the Event notifications in SOA Suite SCA should be aware of the data models, locations, and constraints to pull the data out for object construction and enrichment, and for a more custom version of EBO, you need a more complex extraction for the procedures, bypassing standard OEBS APIs. Finally, we have the hybrid type of services, mixing adapters functionality and task-orchestrated service operations in one service.

There is no service bus like the one we discussed in *Chapter 4*, *From Traditional Integration to Composition – Enterprise Business Services*, it's pretty much a two-layer architecture with one core application and lots of denormalized hybrid services with overlapped boundaries, multiple adapters, some generic (for event) and many proprietary, extracting similar business objects with small variations.

Inside the main application, we can see a lot of custom triggers with overlapping triggering conditions and elements of business code, resulting quite often in **ORA-04091: table <tablename> is mutating…**.

Now the picture is quite clear, we hope. So, what would be the action plan? Ultimately, we should put BES back to operation as the standard Oracle event system, but ideally we should propagate canonical EBM to task services around OEBS, not events. In other words, we should get rid of the callback pattern and eventually minimize the Adapter framework footprint here. We have the perfect opportunity for that, as we have a snowflake architecture (classification you can find in *Application Integration: EAI B2B BPM and SOA* from Bernard Manouvrier and Laurent Menard) in place with a single "source of truth" (OEBS with linked business domains) and lots of trading partners, capable to accept RRDs canonical objects. Therefore, the proposed steps are as follows:

1. As a "table mutation" is the immediate problem, remove all triggers from core business tables and present one unified way of event detection.
2. Consolidate all enterprise objects representations under canonical models, present them as generic XSDs, and make them public and available for all external/internal developing teams. SOA MDS and external HTTP servers will be good options.
3. Analyze message construction procedures and consolidate them inside OESB in order to eliminate the Adapter framework (at least minimize the footprint). All SCAs/BPELs should receive the required EBM without enrichment or excessive transformations.
4. We are not only constructing messages, we receive them as well. Supply every EBO with a unified individual parser/loader. It will conclude the message-handling components set (constructor/parser).
5. After removing table triggers, we have to put all event's recognition/filtering logic somewhere. That will be our rule functions and the XPath artifact, which we also have to register in the Service Repository (or BES subsystem when it will be operational again).
6. Our canonical protocol will be JMS/AQ. Thus, we will have to implement AQ handling for enqueuing and dequeuing inside the main application OEBS.

Refactoring the DB-centric Fusion Application

Some of the presented actions can be achieved by standard OEBS components—XML Gateway, Oracle Application Service Adapters, and BES. We will rely on them as well, but again, our main goal here is to demonstrate how to avoid building the Adapter framework and how to utilize service taxonomy, which we presented in the previous chapter. Then you will be able to roll it up on any DB, simplify the DB Adapter framework, and have better understanding of the internal BES structure of OEBS.

Events registration

The table mutation problem as a first task can be solved quite quickly and drastically, although not everyone will be happy about the radical approach: we just remove all triggers from every Entity business table and implant only one Universal trigger, presented as follows:

```
CREATE OR REPLACE TRIGGER XXCU.XXCU_EHS_REGEVENT_TRG
BEFORE DELETE OR INSERT OR UPDATE ON XXCU.XXCU_EHS_TESTADDRESS
REFERENCING NEW AS New OLD AS Old
FOR EACH ROW
DECLARE
   new_trgBasicEvent  XXCU_EHS_EVENT.BasicEvent_rec;
   msgtype varchar2(20) := 'ERR';
   msgid   varchar2(20);
   msgtext varchar2(2000);
   msgsrc  varchar2(200):= 'XXCU_EHS_REGEVENT_TRG.register_event';
BEGIN
   new_trgBasicEvent.v_srcref_type      := 0;
   new_trgBasicEvent.v_object_name      := 'ADDRESS';
   new_trgBasicEvent.v_objectkey_name   := 'ADDDRESS_ID';
   new_trgBasicEvent.v_objref_data      := :NEW.ADDDRESS_ID;
   new_trgBasicEvent.v_objref_group     := 'EHS';
   new_trgBasicEvent.v_objref_processed := 'N';
   new_trgBasicEvent.v_objref_detected  := 'N';
   new_trgBasicEvent.v_job_id           := 9999;
   new_trgBasicEvent.v_status           := 1;
   new_trgBasicEvent.v_retry_attempts   := 0;
   new_trgBasicEvent.v_transferred      := SYSDATE;
   new_trgBasicEvent.v_processed        := SYSDATE;
   new_trgBasicEvent.v_appuser_code     := 1;
   if inserting then
    new_trgBasicEvent.v_objref_action   :='1';
   elsif updating then
    new_trgBasicEvent.v_objref_action   :='2';
   elsif deleting then
    new_trgBasicEvent.v_objref_action   :='3';
    new_trgBasicEvent.v_objref_data     := :OLD.ADDDRESS_ID;
   end if;
   ---------------------------------------------------------------------
------------
   -- Composite events key registration
   --  . . .
   ---------------------------------------------------------------------
------------
   XXCU_EHS_EVENT_HANDLER.register_event(new_BasicEvent =>
     new_trgBasicEvent);
 EXCEPTION
     WHEN OTHERS THEN
         msgtype := 'ERR';
         msgid   := SQLCODE;
         msgtext := SQLERRM (msgid);
         msgsrc  := 'XXCU_EHS_REGEVENT_TRG.register_event';
         XXCU_COMMON_LOG_RT.msglog (msgtype,
                                   msgtext,
                                   msgid,
                                   1,
                                   msgsrc,
                                   9999
                                  );
END XXCU_EHS_REGEVENT_TRG;
/
```

We decided to put almost the complete code for several reasons; some are as follows:

- From declaration of the `BasicEvent` record, you can immediately understand the structure of the Event Log table, which we will use for physically decoupling basic Events Registration and further outbound processing. For demo purposes, we show the Address object events registration, although it's not entirely "business", it's more like the QDO (see *Chapter 5, Maintaining the Core – the Service Repository*), and it's quite essential for logistic operations.
- You can see the runtime logging call and its data structure. We will use it later while discussing Error Handling.

- The last reason is deducted from the common question—it seems that the Basic Event record contains only the object key, object name, key ID data, CRUD operation indicator, and some timestamps. Would it be enough? The common answer is, amazingly, yes! With proper DB organization, you can construct the Business Object by just using these parameters (OEBS DB is based on Oracle Trading Community Architecture, TCA standard—`http://docs.oracle.com/cd/E18727_01/doc.121/e13570/toc.htm`—and is really well organized). Still, we can supply any Event Handler with ADT array, which is part of the BasicEvent record, holding 20 elements for any object / object context-related data you will ever need in further message construction and propagation. You can increase the number of elements, but in practice, you will never exceed the proposed number. Naturally, the Event Log registration table has a column based on this ADT type and you can query it without any difficulty. So the compound event key registration part looks as follows:

```
--1
  new_event_attr.ParameterName        := 'FND_GLOBAL.USER_NAME';
  new_event_attr.ParameterValue       := 'NA';
  new_event_attr.BusinessName         := 'FND_GLOBAL.USER_NAME';
  new_event_attr.ParameterType        := 'VARCHAR2';
  new_event_attr.ParameterOrder       := 1;
  new_event_attr.ParameterStageIndex  := 1;
  new_event_attr.ParameterStage       := 'old';
  v_EHS_evtcontext.EXTEND;
  v_EHS_evtcontext(1) := new_event_attr;
--2
  new_event_attr.ParameterName        := 'FND_GLOBAL.APPLICATION_
NAME';
...
/*------------------------------------------------------------------
----------------------
                         OBJECT SPECIFIC, OPTIONAL
-------------------------------------------------------------------
----------------------
 Put your values here below, up to 15
-------------------------------------------------------------------
---------------------*/
   --6
  new_event_attr.ParameterName        := 'OLD_ADDRESS_ID';
  new_event_attr.ParameterValue       := 'old_address';
  new_event_attr.BusinessName         := 'Old Address';
  new_event_attr.ParameterType        := 'VARCHAR2';
```

```
        new_event_attr.ParameterOrder       := 6;
        new_event_attr.ParameterStageIndex  := 1;
        new_event_attr.ParameterStage       := 'old';
        v_EHS_evtcontext.EXTEND;
        v_EHS_evtcontext(6) := new_event_attr;
      .....
```

As you can see, the first five are OEBS-specific `FND_GLOBAL` attributes, and you have 15 more elements of your own. The `register_event` procedure just puts the basic event record into the event log table. That's it, registration complete, and no more activities on the Entity table whatsoever. A completely separate process, based on Oracle Job (Scheduler) will scan this table for the new entries and decide on further steps. We have completely decoupled event registration and further object construction. From the number of processing flags in the event record, you can understand that locking mechanism is established on the Event Registration table (see `flags v_retry_attempts`, `v_transferred` and `v_processed` in the previous code) and you could have as many event handlers as you like, implementing the parallel processing: vertically (per Objects type: one for Orders, one for Invoices, and so on) and horizontally (by number of records—500 records per handler, for instance). Therefore, event registration and event recognition parts will never be your bottleneck. However, one problem has to be mentioned. Even though the triggers and event log table footprints are so insignificant that we can hardly consider this technique as invasive, their implementation can be problematic in environments that do not entirely belongs to us. If the vendor is sensitive to his internal DB structure, you will have to find noninvasive ways of events registration.

> If you are an architect on the client's side and during the RFP process, you discover such kind of "sensitivity" combined with the inability of your vendor to explain how events are registered, filtered, and later used for EBM construction, be extra cautious. The presence of the Callback pattern for Events processing also has to be justified.

Implementation of noninvasive ways of event recognition and filtering (or change data capture, CDC) is not more difficult than what was discussed earlier. Even more, some solutions can be much faster with highest selectivity and precision, with zero CDC misses. One possible way can be based on the already existing data replication solution for your critical mission applications (and OEBS is definitely one of them). There are many tools you can employ for this task and SharePlex could be a good option—it has no footprint on the source database; it's extremely quick, and most importantly, it performs replication for data and statements, allowing you to "sniff" certain operations on certain DB objects.

Even more, SharePlex does support data transformation before applying on the target DB and it can work in the hub-and-spoke mode, aggregating data from many sources, not to forget to mention that it can move a tremendous amount of data. It's relatively simple, so establishing filters for capturing events and recording them to the event log for further processing won't be a problem. The solution can be truly elegant, as you kill two (or probably more) birds with one stone—establish CDC with no footprint on the source DB and reliable 24 x 7 zero loss DB replication. Comparing different data replication solutions is not the subject of this book, but looking to products such as Golden Gate, Streams (from Oracle), and SharePlex (now from Dell) from our experience, we can say that SharePlex would be the optimal choice (at the moment of writing), especially considering the price and CDC capabilities. GG is quite similar to SharePlex and also has data transformation capabilities, so if you have budget for it, you can try implementing CDC on it as well.

The last thing that you can see from the previous code—all object- and context-related variables are compliant with the SBDH model we presented in the earlier chapters, so we will use it quite extensively in all our examples.

Events filtering

Earlier we explained the difference between a basic and complex event. It's obvious and generally related to the amount of the object's context information utilized during the event filtering/recognition phase. Initially, the event is registered in a completely business agnostic way, just as an evidence of the business objects changes. All that we know is the business object, primary key data, and business domain. Of course, we have composite context data recorded as an ADT array. What we want to know are two things—who is the recipient (or in what compositions object, propagated as a message should participate), and what would be the format of the message. We have already declared our main goal—minimization of transformation and bridging; therefore, we will focus on recognition of the TP. Thus, we should link the object/object data with potential recipients, as we explained when discussing Service Taxonomy in the previous chapter. Establishing the rules, combined into rulesets, does it. Every object and list of associated events is linked to one or more rulesets, where rules can be functions or XPath-based, as we discussed in the previous chapter. When we have an object in native code for outbound processing, the functional rules are more appropriate; please see the following code. Here we have the simple function, returning location by ID. For Rule Engine simplification, we decided to make all rule functions accepting only one parameter, which is initially our primary key:

```
FUNCTION get_AddressLocbyID(ip_addressid IN
  xxcu_EHS_testaddress.adddress_id%type)
      RETURN VARCHAR2
   IS
      CURSOR AddressbyID_cur (ip_addressid
        xxcu_EHS_testaddress.adddress_id%type)
      IS
        SELECT xxcu_EHS_testaddress.adddress_id,
               xxcu_EHS_testaddress.location,
               xxcu_EHS_testaddress.description
        FROM xxcu_EHS_testaddress
        WHERE xxcu_EHS_testaddress.adddress_id = ip_addressid;
      AddressbyID_rec   AddressbyID_cur%ROWTYPE;
      v_address         VARCHAR2 (400)  := NULL;
   BEGIN
      OPEN AddressbyID_cur (ip_addressid);
      FETCH AddressbyID_cur   INTO AddressbyID_rec;
      IF AddressbyID_cur%FOUND
      THEN
         IF AddressbyID_rec.location IS NULL
         THEN
            v_address := NULL;
         ELSE
            v_address := AddressbyID_rec.location;
         END IF;
      END IF;
      CLOSE AddressbyID_cur;
         msgtext :=  'XXCU_EHS_TEST_ADDR.get_AddressLocbyID.
           Return: '|| v_address;
         XXCU_COMMON_LOG_RT.msglog ( ...    );
      RETURN v_address;
   EXCEPTION
      WHEN NO_DATA_FOUND
      THEN
         v_address := NULL;
         RETURN v_address;
      WHEN OTHERS
      THEN
         msgtype := 'ERR';
         msgid := SQLCODE;
         msgtext := SQLERRM (msgid);
         msgsrc :=
            'XXCU_EHS_TEST_ADDR.get_AddressLocbyID. Unable to
              resolve address location for address ID '
            || ip_addressid;
         XXCU_COMMON_LOG_RT.msglog (..... );
   END get_AddressLocbyID;
```

In general, all rule conditions can be expressed in a function receiving only one parameter or a group of functions assembled in one ruleset using logic aggregation operators (AND and OR). Actually, Rule Engine has no limitation for the number of inbound parameters, but using only one, we can simplify construction of the agnostic decision table; that is, if a dynamically executed function F returns result A when the input parameter is B and the expected result was C, RE considers the rules outcome as negative if the rule condition was = (equal). If this rule was aggregated into the ruleset using a strong rule (AND), the execution is terminated and the decision is negative (no TP-receiver found, for instance). With weak aggregation, execution will be continued until the first hit. The RE counts the number of hits for weak and strong rules, allowing rule grouping (which are not really necessary, as it would be much easier sometimes to implement two rulesets—for weak and strong rules separately). An implementation with multiple conditions (=, <, >, !=, and so on) is also simple. The execution of rule functions is based on NDS and implementation of the dynamic facilitator is exceedingly simple; see the following code snippet:

```
PROCEDURE dynstatexec (
      statement  IN   VARCHAR2,
      nargs_in   IN   NUMBER := 0,
      arg1_in    IN   VARCHAR2 := NULL,
      …
      arg9_in    IN   VARCHAR2 := NULL,
      v_result   OUT VARCHAR2
...
IF nargs_in = 0
         THEN
            EXECUTE IMMEDIATE statement || '; END;';
         ELSIF nargs_in = 1
         THEN
         EXECUTE IMMEDIATE statement || '(:1, :2,); END;'
                      USING arg1_in, OUT v_result;
         ELSIF nargs_in = 2
         THEN
…...
```

You can build it for any number of inbound parameters and for any type of returning value. This approach is highly universal, lightweight, and can be used everywhere, especially combined with a DB schema for service artifacts from the previous chapter: the statement is an executable module registered as a task (service) and linked as a rule in the ruleset, associated with the event/object. If you prefer a more standard approach (and heavy) based on Oracle tools, please proceed with a similar implementation using Oracle BRE API for Java and implement PL/SQL wrappers for it. In this case, you will use the standard rule repository. We believe that the approach presented earlier is a bit more suitable for implementation within OEBS.

One more thing to mention: after execution of business rules, we will have a list of Trading Partners (or none, if nobody subscribed to this event), and a list of messages we have to produce (we agreed that it will be only one Canonical per Entity, but still we have the capability to construct different messages). We can fire execution right away, or we can record discovered information back to the Process Log table. Another process (via Scheduler) will start actual construction. This two-step processing will increase decoupling even more, providing you with great flexibility and deferred execution options, but adds extra latency at the same time. You can select the architectural model more suitable for you.

Message construction

Actual construction is pretty much straightforward. We just have to keep in mind that we are constructing two types of objects: Message Header and Payload. As CLOBs, they will be combined together at the final stage of construction.

Using XDB construction functionality we can build any XML using a standard set of commands (`XMLELEMENT`, `XMLAGG`) in a standard SQL `SELECT` way; see the following constructor for a bank's Credit Monitoring message:

```
SELECT DISTINCT xmlelement("CreditMonitoring",
  xmlagg(XMLELEMENT("Country", XMLELEMENT("Country", 'SE'),
  XMLELEMENT("CustomerCode",
  fnd_profile.VALUE('XXCU_AR_SEBNORDBANKIUID')),
  XMLELEMENT("CustomerCode", v_blank_value),
  XMLELEMENT("CountryCode",  p.country) ,
  XMLELEMENT("CountryRegistrationNumber",
  REPLACE(cm.organization_no, '-','')),
  XMLELEMENT("DUNSnumber", NVL(p.duns_number, '1234567890')),
  XMLELEMENT("CustomerId", v_blank_value) ,
  XMLELEMENT("CustomerId2", v_blank_value),
  XMLELEMENT("Belop1", v_blank_value),
  XMLELEMENT("Belop2", v_blank_value)))).getclobval() into
  v_xml_message FROM xxcu_ocm_credit_monitoring_all cm,
  xxcu_ocm_credit_monitoring_v cm,  ar.hz_parties p
  WHERE cm.utmeldt_dato IS NULL  AND NVL (cm.status, 'x') = 'A'
  AND cm.party_id = p.party_id
```

The output will be CLOB containing our XML message. The presented message is fairly simple and so the statement is, but beware—real-life messages (Orders, Bill of Ladings, Cargo Manifests) can be several pages long. Still, it's probably the most common way of XML construction, as the XSU XDK utility (9*i*, 10*g*) is rather obsolete. The XML message can be constructed in a familiar way and sent directly to TP (using AQ).

The important point here is that the ways of XML message construction are probably important for SOA realization but not directly related to the minimization of the Adapter framework and efforts related to it. From the task definition, we know that in RRD, we already have a huge amount of reusable SQL cursors (available in package specifications) and recreating the same using XDB would be really a big waste of time and effort. Thus, we have to find the ways to reuse existing cursors in XML construction procedures. Obviously, we will again use the custom approach and we are about to present it to you in a very concise way. Teaching you PL/SQL is not the purpose of this book, so we believe that you can figure out the simple commands behind the `XXCU_COMMON_UTIL_*` packages we employ in our examples.

The custom message constructor can be implemented by performing the following simple steps:

1. Make sure that all necessary cursors are public and valid.
2. Create temporary CLOB for further EBO construction and open it using the following line of code:

   ```
   v_olob := XXCU.XXCU_COMMON_UTIL.createtmpclob;
     DBMS_LOB.OPEN (v_olob, 1);
   ```

3. Open initial tags according to your XSD and open cursor using the following code:

   ```
   XXCU.XXCU_COMMON_UTIL_XML.open_xml_tag (v_olob,
     'CreditMonitoring');
     XXCU.XXCU_COMMON_UTIL_XML.print_xml_tag
     (v_olob,'CountryCode', ip_countrycode);
     OPEN creditmonitoring_cur(ip_countrycode);
     LOOP FETCH creditmonitoring_cur INTO
     creditmonitoring_rec; EXIT WHEN
     creditmonitoring_cur%NOTFOUND;
   ```

Construct the Business Object XML. If you have nested nodelists—call procedures for their construction. To do so, just pass your temporary CLOB to a subprocedure and use the same construction steps. Do it in a loop if you have recordsets with many rows, shown as follows:

```
XXCU.XXCU_COMMON_UTIL_XML.open_xml_tag (v_olob, 'Company');
  v_country_id_attr := XXCU.XXCU_COMMON_UTIL_XML.get_attribute
  ('CountryCode', creditmonitoring_rec.country_code);
  XXCU.XXCU_COMMON_UTIL_XML.print_xml_tag (v_olob, 'Country',
  creditmonitoring_rec.country );
  XXCU.XXCU_COMMON_UTIL_XML.print_xml_tag (v_olob,
  'CustomerCode', creditmonitoring_rec.customer_code);
  XXCU.XXCU_COMMON_UTIL_XML.print_xml_tag (v_olob,
  'CountryRegistrationNumber',
```

```
    creditmonitoring_rec.country_registration_number);
    XXCU.XXCU_COMMON_UTIL_XML.print_xml_tag (v_olob, 'DUNSnumber',
    creditmonitoring_rec.duns_number);
    XXCU.XXCU_COMMON_UTIL_XML.print_xml_tag (v_olob, 'CustomerId',
    creditmonitoring_rec.customer_id);
    XXCU.XXCU_COMMON_UTIL_XML.print_xml_tag (v_olob, 'CustomerId2',
    creditmonitoring_rec.customer_id2);
    XXCU.XXCU_COMMON_UTIL_XML.print_xml_tag (v_olob, 'Belop1',
    creditmonitoring_rec.belop1);
    XXCU.XXCU_COMMON_UTIL_XML.print_xml_tag (v_olob, 'Belop2',
    creditmonitoring_rec.belop2);
    XXCU.XXCU_COMMON_UTIL_XML.close_xml_tag (v_olob, 'Company');
  END LOOP;
```

(Please compare this approach with the pure XDB-based sample from the previous snippet.) Close the tags, cursor, and temporary CLOB. Remember that CLOB is your output parameter shown as follows:

```
  CLOSE creditmonitoring_cur;
    XXCU.XXCU_COMMON_UTIL_XML.close_xml_tag (v_olob,
    'CreditMonitoring');
    op_olob := v_olob;
    XXCU.XXCU_COMMON_UTIL.closetmpclob (v_olob);
```

That's it, construction completed. Now you can construct any canonical message (cursors are common and reusable, that was the purpose), or create individual messages for selected TPs quite easily, because the presented approach provides the highest granularity possible. Either way, transformation could be avoided, as we will have individual constructors. Alternatively, to minimize the number of constructors, XDK transformation functions could be used. Next, we will shortly explain the purpose of functions used in a message construction as shown in the following table.

> Generally, the functions are based on `DBMS_LOB.WRITE` not on `DBMS_OUTPUT.PUT_LINE`.

Function	**Description**
`XXCU.XXCU_COMMON_UTIL_XML.open_xml_tag`	Open an `XML` tag for a nodelist or group of nodelists. The attribute can be used as `<Element>`.
`XXCU.XXCU_COMMON_UTIL_XML.close_xml_tag`	Close the `XML` tag for a nodelist or group of nodelists as `</Element>`.
`XXCU.XXCU_COMMON_UTIL_XML.get_attribute`	Create an attribute for further use.
`XXCU.XXCU_COMMON_UTIL_XML.print_xml_tag`	Create fully formatted element, attribute available as `<Element>data</Element>`.

Alternatively, you can use the standard DBMS_XMLGEN package (or older DBMS_XMLQUERY, which is implemented in Java and therefore in not supported with Oracle DB Express Edition, thus this technique is not truly universal). Here NULL values and date values are supported natively by DBMS_XMLGEN.setnullhandling() and NLS_DATE_FORMAT.

Message parsing

To complete the message-handling routines, we will discuss the inbound flows. The processing of inbound messages is slightly different and not only because of the opposite flows direction. First, we do not need Event Registration tables—we have AQs instead, and they are pretty much tables as well. Secondly, you can have as many AQs as you want; naturally, one table does not limit you. You can have separate AQ per TP and/or Object or group of objects. You can even have only one AQ per object, serving all TPs though the single object queue. All what you have to do is to declare this AQ as ADT-based and then you will have the perfect possibility to dequeue messages by individual handlers per TP. However, most importantly, for inbound XML messages, you will have only one rule function based on XPath's valueof() and all rulesets will be just a collection of XPaths with expected values. You can achieve the highest level of parallelism and keep everything manageable at the same time. Most probably you will decide to handle an inbound message at the moment of dequeueing, so balancing the amount of handlers and queues is your primary task as an architect. Once message parsed, all its elements will be presented as a recordset and we will just call INSERT statements we already have for core Entity tables (business APIs). Thus, the implementation sequence will be as follows:

1. Declare records parsed from XML values. The message header record v_SBDHShort for message header is mandatory and shall be preserved. At least one business object record must be declared. This record(s) will be later used as input parameter for your business API. The record can be based on business OEBS table, or any other structure from a custom RRD business object as:

   ```
   v_tst_address_rec xxcu_EHS_tstaddr_parser.tst_address_rec;
   v_SbdhShort XXCU_EHS_MESSAGE_HEADER.SBDH_rec;
   ```

2. Declare XML parsers, nodes, and nodelists. The number of nodes and nodelists must be equivalent to the number of related objects in the received XML. Reserve the first nodelist for message header values. We could use the values from it later in your API call (it's our object context):

   ```
   v_inXmlDOM XDB.DBMS_XMLDOM.domdocument;
   parser XDB.DBMS_XMLPARSER.parser;
   style XDB.DBMS_XSLPROCESSOR.stylesheet;
   ```

```
curnode0 XDB.DBMS_XMLDOM.domnode;
curnode1 XDB.DBMS_XMLDOM.domnode;
msg_msgheader XDB.DBMS_XMLDOM.domnodelist;
address_proplist XDB.DBMS_XMLDOM.domnodelist;
theDocElt XDB.DBMS_XMLDOM.DOMElement;
```

3. Oracle XDK parsers are namespace-aware. We have to take it into account parsing different namespaces with different prefixes, or just strip them completely using search and replace, leaving only the MH namespace in place. This solution is rather dirty as we handle XML as a plain text, but if no security concern is expressed, this solution can be really quick and simple.
4. Parse the document (inbound CLOB). Another thing to remember is that Oracle Applications can have several DOM and SAX parsers, so we should use the DOM parser associated with version *10gR1* and higher (RRD current solution is on *11gR2*). In *R11*/*R12*, the parser is in the XDB schema:

```
parser := XDB.DBMS_XMLPARSER.newparser;
XDB.DBMS_XMLPARSER.parseclob (parser, olob);
v_inXmlDOM := XDB.DBMS_XMLPARSER.getdocument (parser);
XDB.DBMS_XMLPARSER.freeparser (parser);
```

5. The `get_SBDH` procedure returns the standard business document header using `XMLDOM` as an input parameter.
6. Start the main parsing loop and populate nodelists and nodes, using the following code:

```
address_proplist := XXCU_COMMON_UTIL_XML.selectnodes
  (v_inXmlDOM, msg_BO_root_url);
  FOR b IN 1 .. XDB.DBMS_XMLDOM.getlength
  (address_proplist) LOOP curnode1 :=
  XDB.DBMS_XMLDOM.item (address_proplist, b - 1);
  ....
```

7. Go through the DOM tree and map nodes to record elements using simple `XPATH` expressions (process elements). Also, we advise you to use the `NULL` value substitution and remove line break functions, as shown in the following code. Implementation is skipped for brevity. We believe that you have a good understanding about the illegal XML characters and how to handle line breaks:

```
...
v_tst_address_rec.addressID :=
  NVL(XXCU_COMMON_UTIL_TXT.removestrbrakes
  ((XXCU_COMMON_UTIL_XML.valueof(curnode1,'addressID'))),
  v_null_value);
...
```

8. After parsing, we can call the business API using populated record(s) as input parameters. The Business API should always return two output parameters: `v_status_code` and `v_error_message`, where the first is code (`0`: status OK, `1`: API warning, `2`: critical API error, or any other coding at your choice), and the second is a textual description of the API code. Based on the status code, the framework will decide how to process a faulty message, shown as follows:

   ```
   apps.xxcu_om_import_creditinfo_pkg.import_creditinfo(
     p_data_parties         => v_parties_int_rec
     , p_data_addresses       => v_addresses_int_rec
     , p_data_contactpts      => v_contactpts_int_rec
     ...
     , p_data_creditratings   => v_creditrtngs_int_rec
     , p_operating_unit_name => null
     , x_status_code          => v_status_code
     , x_error_message        => v_error_message );
   ```

9. Do not forget to close all temporary CLOBs and parsers, as shown in the following code:

   ```
   XXCU_COMMON_UTIL_XML.freedocument (v_inXmlDOM);
     XXCU_COMMON_UTIL.closetmpclob (olob);
   ```

So, the task was done in nine simple steps. Logically, the number of parsers will be equal to the number of inbound messages, but as long as all of them (we assume) are in canonical form, their number will be quite moderate. Again, after message detection and extraction from the queue, we can apply the transformation to any custom message, but this approach should be avoided. In general, these exercises for establishing parsers and constructors have a very positive effect on internal App organization, expressed as follows:

- The statements-duplicates provided by parallel teams were consolidated and adjusted according to the recommended canonical models. Canonical models are finalized and approved at last.
- All the remaining SQL statements were carefully turned for optimal performance, and the usage of all clauses such as `DISTINCT` and `UNION` were redesigned.
- All artifacts, required for rule-based processing and any kind of dynamic invocations were registered in the Service Repository.

- Canonical models, related parsers/constructors, Business Events, and all other public artifacts from Service Repository were also published on Corporate WIKI, a "How-To Cookbook" was composed and provided to every developers.
- Checklists from the previously enforced formal delivery acceptance process.

The example will be not complete without a few words about canonical endpoint handling.

Endpoint handling

RRD Canonical Protocol is JMS/AQ and we expect that all communications will be queue-based, especially for inbound flows, because that will considerably simplify message recognition and parsing, as demonstrated previously. However, as we mentioned earlier, Oracle Fusion Apps and Oracle DB, in general, are very capable of supporting practically all common protocols, without the Adapter framework, directly to the Service Bus, which is service collaboration and decoupling layer. Oracle DB Listener is a HTTP listener as well, we can easily do the HTTP `POST`/`GET` and construct SOAP messages, either manually or using OEBS XML Gateway. File operations never were a problem. We are not sure that doing FTP directly from DB is a good idea from a security standpoint, but we can do that as well. Practically, as we can call any Java function and wrap it in PL/SQL, we can do anything. The question is—should we?

The answer is that the RRD SOA Implementation board officially prohibited all protocols for internal developers except JMS/AQ and that was the right thing to do. Still, the proper solution will introduce the separation of a message's construction and delivery, so after gluing the Message Header (constructed as a separate CLOB) and Message payload in one container (`msh_olob`), we call the message dispatcher, which will select the correct delivery MEP (JMS/AQ, all other methods are commented). MEPs and protocols are associated with TP messages and TP endpoints according to the presented service taxonomy, and therefore, dispatching is simple, flexible, and architecturally identical to the Java Deliverer, as discussed in *Chapter 4, From Traditional Integration to Composition – Enterprise Business Services*. A dispatcher call with all associated parameters is presented as follows:

```
msgtext := 'Invoking deliverer for Message ID= ' || v_msgid;
  XXCU_EHS_ENDPOINTHANDLER.msgdispatcher
  (olob            => msh_olob,
   msgid           => v_msgid,
   msgjobid        => msgjobid,
   db_action       => v_dbaction,
   ret_err_status  => ret_err_status
  )
```

Enqueue

Here is a small and simple enqueuing procedure, based on the `DBMS_AQ` package:

```
procedure enqueue_msg(p_queue    in varchar2,
                      p_xml      in varchar2,
                      p_priority in number default 50,
                      p_corrid   in varchar2 default null
                      )
is
  msgsrc VARCHAR2 (4000):= package_name||'.enqueue_msg(VC)';

  l_stmt               varchar2(250 CHAR) := 'declare';
  l_jms_message        sys.aq$_jms_text_message;
  l_message_properties dbms_aq.message_properties_t;
  l_enqueue_options    dbms_aq.enqueue_options_t;
  l_msgid              raw(16);
begin
  -- blocked in XE
  l_jms_message := sys.aq$_jms_text_message.construct;
  l_jms_message.set_text(p_xml);

  l_message_properties.priority := p_priority;
  l_message_properties.correlation := p_corrid;
  dbms_aq.enqueue(p_queue,
                  l_enqueue_options,
                  l_message_properties,
                  l_jms_message,
                  l_msgid
                  );
  commit;
 EXCEPTION
    WHEN OTHERS THEN
         msgtext := 'EQUEUING FAILED: ';
         msgcode := SQLCODE;
         msgtext := msgtext ||'; '||SQLERRM (msgcode);
    XXCU_COMMON_LOG_RT.msglog('ERR',msgtext, msgcode,usermsg,
      msgsrc, msgjobid);
end enqueue_msg;
```

Dequeue

The dequeuing functionality is also based on the `DBMS_AQ` package and is fairly simple, as shown in the following code snippet:

```
DBMS_AQ.DEQUEUE(
  queue_name         => l_queue_name,
  dequeue_options    => l_dequeue_options,
  message_properties => l_message_properties,
  payload            => l_message,
  msgid              => l_message_handle);
```

Some specific functionality exists for inbound flows, as we mentioned previously. The `EndpointHandler` function will invoke only the parser associated with the received message. This invocation will be performed dynamically and we do not have to code it! That's one benefit of this framework. However, message-parser association is a developer's responsibility and basically it's done through an ESR configuration.

Association can be based on the rootnode element's name. It will be recognized after parsing of the incoming message. As long as every message has a unique root (and that's true for all canonical messages), the message ID will be recognized by a simple lookup on ESR's message definition table, where every message (ID) is connected to the parsing routine, registered in the ESR Service/Task table as parser. The developer is responsible for the proper declaration of the construction and parsing tasks in the Service/Task entity table and linking them to the canonical message (or any, in that matter). Another, and probably more correct, way of message recognition would be the parsing of the `msg_ID` element in the message header. Naturally, XPath is static for this element. You can see one of the possible realizations in the following code snippet:

```
v_rootElementName:=XDB.DBMS_XMLDOM.getLocalName(XDB.DBMS_XMLDOM
  .getDocumentElement(v_eventDOM));
  msgtext := 'Root node detected :'||v_rootElementName;
XXCU_COMMON_LOG_RT.msglog(msgtype, msgtext, msgcode, usermsg,
  msgsrc, v_parsejob_id);
  v_msgid  :=   XXCU_INTF_MESSAGE.get_msgidbyroot
  (ltrim(rtrim(v_rootElementName)));
v_parser := XXCU_INTF_MESSAGE.get_msgparser(v_msgid);
```

When a parser is recognized, it can be dynamically invoked using the NDS functionality, as shown in the following code:

```
//invoking parser
 execute immediate
                  'BEGIN '||v_parser||'(:1, :2, :3, :4 ); END;'
                  USING IN ip_lob, IN v_msgid,
                  OUT v_status,
                  OUT v_status_text;
```

Actually this type of dynamism would not be needed if we establish an individual queue handler for every TPs inbound queue. In this case, we will do inline message parsing together with dequeueing and all business API as a last step.

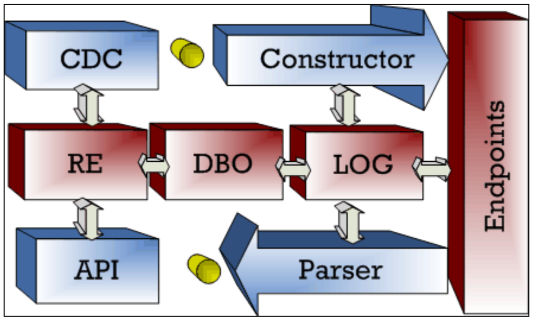
Some notes regarding universal endpoints handling are as follows:

- A few words about the `DBMS_AQ` package would be in order. We mentioned that the functionality is fairly simple, and it is actually, for the Oracle DB Standard/Enterprise editions, at least. A complete solution will work perfectly on XE as well, but AQ functionality is a bit tricky (Java is required), so we advise you to look for the very elegant solution on the AMIS Technology Blog (actually, you can find it in many places). Thus, you will have several AQ handling functions in your endpoint dispatcher.
- File object handling is also rather simple in implementation, but again we would advise you to use Java for that and wrap it into the PL/SQL package. We also advise you to avoid handling remote file objects from your DB.

Conclusion

Although being very simple and lightweight, the entire solution is about 40 PL/SQL packages (only core, excluding individual CDMs constructor/parser pairs), 10 separate and embedded Java modules, 4 separate FSO/FTP scripts, and a complete ESR DB schema as presented in the previous chapter. From the operations' perspective, it includes about 100 individual installation scripts, assembled in one bundle for jobs, triggers, index/table optimization, and so on. We simply do not have enough space to show it to you in detail (source code for core packages will be provided together with the ESR DB schema), but the general idea has been conveyed quite clearly, covering all aspects of solution's lifecycle, starting from the development's perspective, as follows:

- It is quite possible to implement a business agnostic solution in any silo-based Oracle application, converting it to be completely SOA-oriented. Following the SOA paradigm, with a clear understanding of design principles and relations between service artifacts, all these packages were delivered into production in 40 working days.
- Design proved to be so native and logical that after a couple of hours of training, all developers from the main RRD vendors become familiar with the concept and were capable of carrying on. The reason for that can be seen through the following figure—the red architectural blocks will be received by every team directly from the framework and the team will have to do only minor changes in the blue ones adapting them to specific EBM/Event. The DBO part is actually our Service Repository, as shown in the following figure:



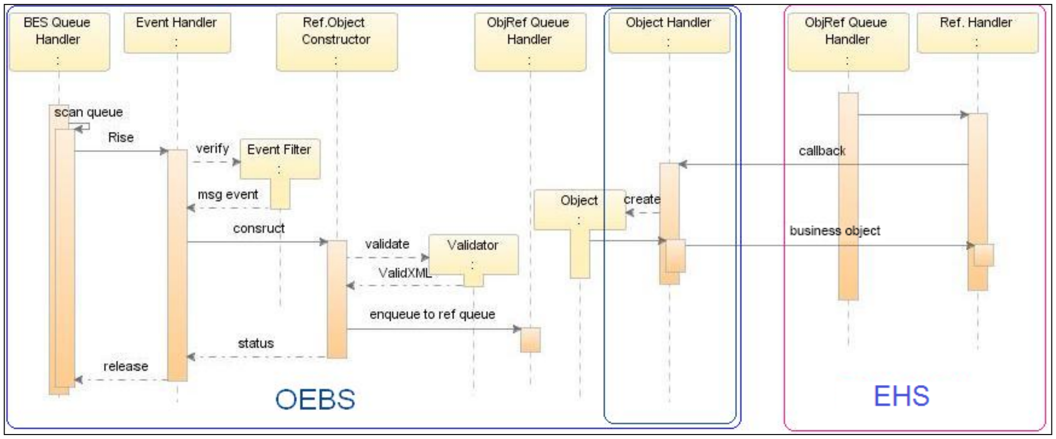
- As a result of the previous point, the development cycle reduced considerably. Implementation of a complex message with all related events reduced to two working days (Purchase Order EBM). The entire Order Management program was implemented in two weeks, plus two weeks for thorough testing (from the very beginning we agreed that every message handling module will be supplied with testing packages, that is, the actual parser will be delivered with the constructor for unit test and vice versa). Interestingly, the dynamic execution approach (based on the service Repository taxonomy and NDS) became so popular that we had to restrain developers from using it for purposes other than event filtering, message parsing, and construction.
- Finally, now you can clearly see the resemblance between the RTD pattern, implemented in *Chapter 4*, *From Traditional Integration to Composition – Enterprise Business Services* in custom Service Broker realization and this solution. We used the same methods, patterns, and semantics. Obviously, according to the Law of Indestructibility of Matter, the Adapter framework cannot just disappear by a flick of the SOA magic wand. We have to move some components to another layers, also changing the surroundings (Canonicalization first).

In addition to the development perspective, we would like to explore several points from a pure vendor-neutral architectural standpoint, as follows:

- Without any revolutions or the Big Bang, we refactored the solution making it more composable. We followed the reusability principle, employing existing cursors for message construction. We engaged the Loose Coupling principle for the separation of event detection, message construction, and message delivery. We observed the parallelism, maintaining desirable throughput and performance. And most importantly, we implemented Canonical Schema and Canonical Protocol; the main enablers of Composability for this solution.

- We enforced the Hollywood principle, presenting a completely decoupled agnostic solution. Now, middleware does not need to know anything about OEBS APIs, internal table structure, concurrent programs, and so on to pull the data after receiving an event message. The complete (Canonical EBM) message is promptly delivered with minimal delay.
- No protocol bridging, no transformations, both model or format. The number of EBM/EBOs has been reduced to the optimal level.
- Finally we considerably reduced the Adapter framework around the Oracle E-Business Suite. That was our ultimate goal and we reached it in a very short time. Mind you, we did not say "eliminated", as some ABCS elements remain dedicated to really weird things. Actually, that's what the Adapter framework is for.
- And one more thing—RRD's **Event Handling System** (**EHS**) solution is completely portable. It can be implemented on any Oracle DB (including XE). Although it was not our first (or even second) goal, this option is quite positive from a budget planning perspective.

With decoupled message construction queue and the native BES queue handler, the message construction sequence will be as presented as follows:



A vendor-neutral architectural perspective must be extended with aspects that are specific to Oracle-based implementation, and specifically, with benefits emerged from tools standardization:

- One of the cornerstones of this approach of eliminating ABCS layer is Event Detection. Although TIBCO is quite prominent for its Event-driven Architecture, Oracle also has a good history of Event recognition solutions, naturally based on DB products. Starting from 9*i*, Oracle provide CDC in an almost identical way as described previously, based on triggers, change tables, and mostly suitable for the ETL pattern. Remember, ETL is what we are trying to avoid here. From *10g*, the approach was enhanced by asynchronous feeds from changes detected into the redo logs. The advantage is obvious—the CDC is not part of the transaction anymore. Together with Oracle Streams, the CDC is capable to detect and propagate the changes to subscribers. In general, tables change history trace functionality was possible because of the new `DBMS_FLASHBACK` package. In *11g*, synchronous change data capture was introduced. Still, to understand the meaning of the basic event, the events data should be transported to the **decision support system** (**DSS**) or **online analytical processing** (**OLAP**) applications. You can employ these options for CDC, but you still need to do an analysis for a proper decision and also bear in mind that some of these options are available in the Enterprise Edition only.
- The XML message construction can be easily done by XDB constructs if you do not have legacy of existing cursors and will build everything from scratch. Otherwise, a custom construction utility would be more suitable; at least you will avoid a lot of mistakes by simply following the existing query logic in cursors.
- Message delivery is also one of the benefits. Actually, standardization options here can be seen purely from an SOA perspective. We need application/message agnostic protocol and MEP. JMS and JMS/AQ is an optimal choice as we could avoid implementation application-specific adapters and minimize the ABCS footprint in general. Of course, if your SOA infrastructure is completely based on the data model, provided by the OEBS XML Gateway or Application adapter by default, and all consumers are happy with that, then you should go for this solution. However, evaluate the complete Oracle AIA implementation as it could be the best choice, especially with all the industry PIPs.

> We mentioned this before and we would like to repeat it again—SOA doesn't have to be expensive. On the contrary, based on so many technologies and standards, it allows us to choose the optimal combination of tools and methods exactly according to our specifications and budget. We are not selling Oracle products, so we have no interest in offering AIA, CDC, ODI, and GG when you do not really need them. Actually, we strive to share the lessons we learned from our practical SOA implementation, based on Oracle tools.

Finally, what operations will we get after replacing the Adapter framework with this event driven solution? Are there any significant benefits of this radical simplification? Let's start from core NFRs:

- That's a really interesting point. Yes, it's inexpensive and easy to use, but what about the performance? Although the company's name is fake, the following numbers are real:
    - 350,000 to 450,000 outbound messages monthly for all OEBS domains. The number of inbound messages is roughly the same.
    - Depending on the financial period, the daily rate can be up to 100,000 messages per day.
    - Throughput is about 10 msg/sec for a 150 KB Order message. This parameter was never a problem as we can increase operational concurrency by implementing parallel queues/handlers.
    - The error rate is approximately 0.01 percent faulty message per week. Practically, all errors are related to XML validation for inbound messages.
- One of the declared goals was to put OEBS BES back into production. After an initial trial period, EHS implementation was considered so satisfactory that it stayed in production for three years (since 2011).
- Because of considerable simplification of the Adapter framework, migration from OFM 10*g* to 11*g* was gradual and relatively smooth.

Of course, there are always walls to hit. We had plenty of them, as already mentioned, including AQ implementation in XE, the `DBMS_LOB` functions for CLOB handling below and above 4 KB, and so on. We can explore the following two major factors that helped implement this approach:

- Strong support from local RRD Architecture team. Because of a very strong discipline established by system owners and wide authority finally granted to the chief architect (truly, *Necessity is the best Advisor*), implementation of this concept became possible in such a competitive environment.

- When developing in Java or SOA Suite, we have plenty of tools for maintaining concurrent development, implementation, and components interdependency (Maven is the natural choice, Hudson/Jenkins, classic Ant, and so on). Unfortunately, we do not have so many options for team collaboration in PL/SQL. Code rollup is a tricky business in centralized DB environments, such as OEBS, when a dozen teams work on the same modules (Account Payable, for instance) and we need some framework in addition to the strong discipline. Luckily, such frameworks were established, thanks a lot to the architect who managed to build a Maven-like tool, which allowed us to do complete or partial installations, code validation, and assembly on running environments practically without downtimes.

We hope that this example demonstrated how to optimize the Adapter framework (for DB adapters) by refactoring core applications, making them more SOA-oriented. Now, we return to the middleware layer and see how the ABCS optimization can be achieved there.

Establishing the Adapter Framework

The Adapter framework optimization, or, in fact, the minimization of all protocols/endpoint types aiming the realization of the Official Endpoint SOA pattern presented previously, depends on many factors (we have indicated them in the *Conclusion* section), but decisive is only one. As we already mentioned, this factor is not technical, unfortunately. At the end, the main question is the level of ownership we as architects have on refactored application endpoints. When the approach is invasive (we have to inject unified triggers and establish event handling and ESR schemas), we totally depend on the level of cooperation with the application owner/vendor. If this approach is not approved, then noninvasive methods shall be considered and some of them, as we explained earlier, could be based on disaster recovery data replication and here we could face even bigger resistance from DBAs/Operations.

Who can blame them? The approaches we discussed in the first part of this chapter have been proved many times and you can completely rely on them, but at the end, everything will rest on your ability to convey the rational of discussed SOA Patterns to the managers and application owners. We already gave you enough arguments to win hearts and minds (including the real operational figures, which in complete stress test were 35 percent higher), so we will put this discussion aside for now, but the possible outcome of this discussion could be "Build the integration layer to get our data" and that means—establish full-fledged ABCS. It doesn't mean that our optimization failed; it's just another confirmation that the SOA approach has plenty of room for traditional integration methods, which we will apply consciously. Thus, we move back to the CTU Telecom Enterprise, as we left it in *Chapter 4*, *From Traditional Integration to Composition – Enterprise Business Services*.

One small note before we continue with the CTU Adapter framework. Application Adapters, Technology Adapters, and Protocol Adapters are really a strong (and probably the strongest) side of OFM. The list of available adapters is enormous (we mentioned it in *Chapter 2*, *Oracle Fusion Introduction – A Solid Foundation for Service Inventory*) and the methods of their implementation are wizard-based and very native. This fact also reflects Oracle's transition from traditional integration to the composition-oriented approach. You can find a lot of books and web tutorials full of step-by-step demos, overloaded by screenshots, and we would like to spare you from that here. Still, some rules of thumb will be mentioned, which are as follows:

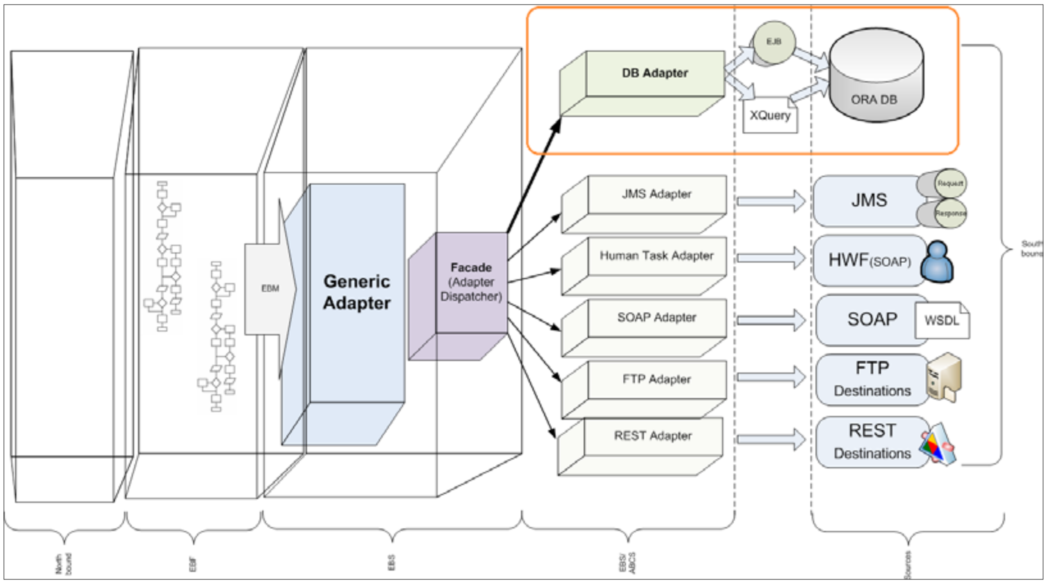
- The primary rule of adapters design can be expressed as "sacrifice the Reusability for better Loose Coupling and Contract Standardization". Indeed, for better modularity and performance, each adapter should be tailored to the application's endpoints. In order to keep all applications hot-pluggable, we should avoid the reuse of adapters (well, not exactly; the following examples will show why).
- An adapter should not disturb the underlying layers (ESB and Orchestration) by endpoint-specific errors and should not propagate them when the source endpoint cannot provide the necessary data or a target cannot accept the correct message. Retry mechanisms, close integration with audit, and error handling are a must. Discussing the EHS framework earlier, we omitted this part (which is quite substantial and based on AQ and business rules), but we will definitely return to this in *Chapter 8*, *Taking Care – Error Handling*.
- In order to support the first design rule, Adapter (Northbound) must provide a Canonical message and accept it before transformation to the ABM (Southbound). Thus, both types of transformation patterns and bridging should be implemented in Adapter. Remember that they are parts of the Service Broker pattern, which is in turn an essential part of ESB.

Following this approach, you can just use BPEL straight away. Just beware of hybrid service implementation, and you do not need to read further. And the hybrid services are quite a significant risk, as we demonstrated during the analysis of CTU's initial solution. That's why Oracle recommends isolating ABCS from EBF by EBS layers. So, the rules are meant to be broken, isn't it? Not quite. Again, we are just following common sense, that is, applying the pattern to a "meaningful extent".

What if (like in CTU) all your main ERP applications are scheduled for decommissioning in the next decade, that is, will stay for quite a while? What if the number of your applications, which you have to wrap by adapters, is so big that the quantity of adapters will just make your SOA infrastructure cumbersome?

Even more, what if all these applications are very similar in technology and in business purpose (CTU example, regional installations of the same application with minor changes)? Thus, the Reusability principle can be applied in the Adapter framework as well and we are going to demonstrate how.

In *Chapter 4, From Traditional Integration to Composition – Enterprise Business Services*, discussing OSB, we established the ESB layer with **Generic Adapter** (**GA**), fulfilling the VETRO pattern in an endpoint-agnostic way (which is an absolutely pure solution according to the SOA Pattern catalog) and Service Façade, dynamically dispatching the message (or service call) to the concrete Protocol Adapter. In addition to the classic ESB SOA patterns altogether, they are followed to the Business Delegate and Adapter Factory JEE pattern implementation rules; most of the Transport Adapters are JCA-based and all necessary parameters were transported to the GA and Facade in a message payload in the form of Adapter Message. Of course, the structure of this message is adapter bound and these elements are extracted from the Service Repository during the EP construction phase in the Composition Controller. We also kept the possibility to do the lookup in GA for EP extraction simplification, as these attributes are not necessary for the entire composition. You can see the list of Transport adapters in the following figure:

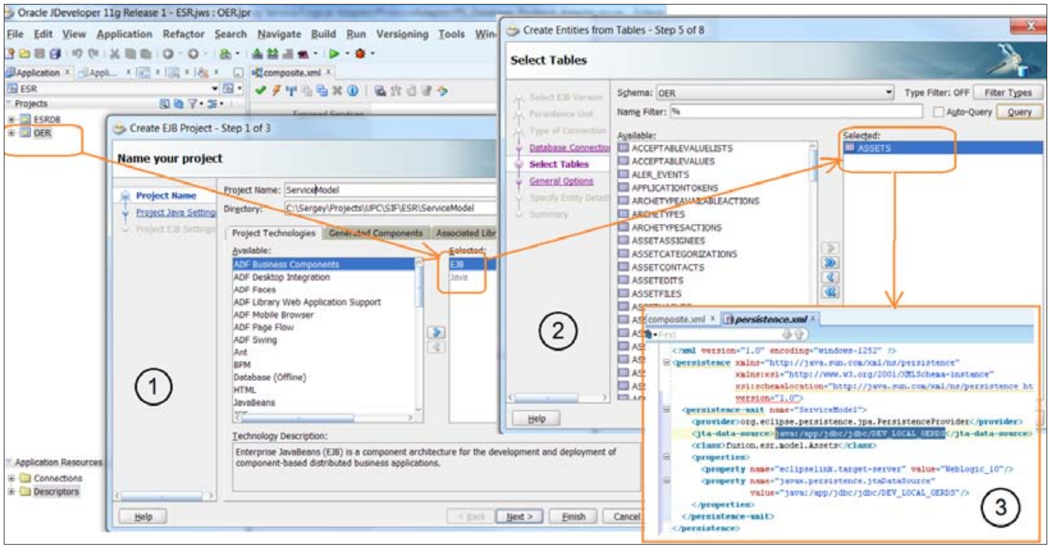


In the scope of one chapter, it is simply impossible to cover dynamic implementation of all adapters, so we will focus on the DB adapter first (as we did earlier during the RRD example), as probably the most common and interesting and will show two different methods of technical implementation. Before we do that, we will demonstrate how a traditional DB SCA adapter can be exposed on OSB and how we can decouple the EJB implementation using a proxy service on service bus. These two examples can be used directly in your projects as well, but they are essential for understanding the dynamic approach.

Exposing EJB through OSB

For the demonstration of the concept, we will create simple EJB using JDeveloper as an Entity bean and expose it as a Stateless Session bean on OSB. We will not go into the details as you can find all necessary details in the Oracle documentation or a dozen cookbooks, full of step-by-step instructions and screenshots for every mouse click. Entity beans are most often based on entities from DB tables, so we can easily use the DBs and DB connections we established in the previous chapter.

Create a new project in the same application we used for offline OER DB analysis **(1)**. If you didn't look at the internal OER structure, you can choose any other DB source you have for test purposes; just one entity table will be enough. Our choice is naturally the Asset table in the OER schema. While configuring the EJB settings, please select EJB 3.0 (what else?) and invoke the wizard constructing Entity from tables. You have several options for the DB connection type (`online`, `offline`, and `from AppServer`); select the one most suitable for you. We will proceed with **online** and select the **ASSETS** table **(2)** after querying OER schema. In any case, go to the `persistence.xml` file **(3)** after EJB generation and change `jta-data-source` to the correct one you use on your server (in two places).



Create EJB Project - Step 1 of 3

Name your project

Project Name: ServiceModel

Directory: C:\Sergey\Projects\UPC\SIF\ESR\ServiceModel

Project Technologies — Selected: EJB

Technology Description: Enterprise JavaBeans (EJB) is a component architecture for the development and deployment of component-based distributed business applications.

Create Entities from Tables - Step 5 of 8

Select Tables

Schema: OER — Name Filter: %

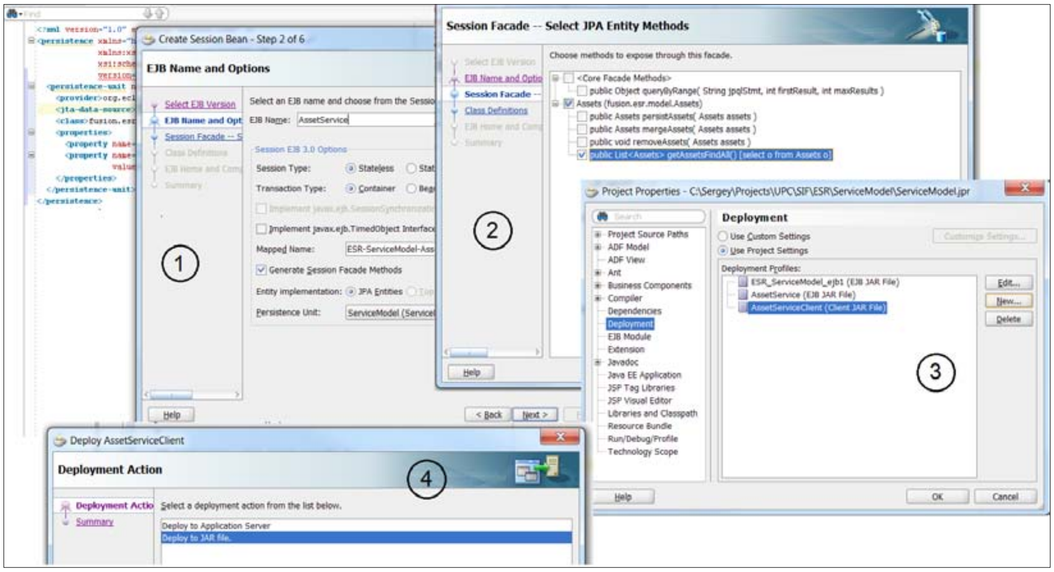
Selected: ASSETS

```
<?xml version="1.0" encoding="windows-1252" ?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://java.sun.com/xml/ns/persistence ht
             version="1.0">
  <persistence-unit name="ServiceModel">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <jta-data-source>java:/app/jdbc/jdbc/DEV_LOCAL_OERDS</jta-data-source>
    <class>fusion.esr.model.Assets</class>
    <properties>
      <property name="eclipselink.target-server" value="WebLogic_10"/>
      <property name="javax.persistence.jtaDataSource"
                value="java:/app/jdbc/jdbc/DEV_LOCAL_OERDS"/>
    </properties>
  </persistence-unit>
</persistence>
```

Oracle JDeveloper 11g Release 1 - ESR.jws : OER.jpr
File Edit View Application Refactor Search Navigate Build Run Versioning Tools
ESR
OER
ESRDB
composite.xml
persistence.xml
1
2
3
Help
< Back
Next >
Finish
Cancel

Building the Assets entity bean

Now we can create Session bean **(1)** and expose the operation(s), which we later will introduce using OSB Proxy. Start the **Session bean** wizard, create the **AssetService** bean and expose only one operation `getAssetFindAll()` for brevity **(2)**. Create local and remote interfaces for this bean (using default settings in the interfaces section). It will be also useful to create separate deployment profiles for EJB JAR and Client JAR (navigate to **[Project Root]** | **Project Properties** | **Deployment** | **New**) **(3)**.



Create Session Bean - Step 2 of 6

EJB Name and Options

EJB Name: AssetService

Session Type: ● Stateless

Transaction Type: ● Container

☑ Generate Session Facade Methods

Entity implementation: ● JPA Entities

Persistence Unit: ServiceModel

Session Facade -- Select JPA Entity Methods

Choose methods to expose through this facade.

☑ public List<Assets> getAssetsFindAll() [select o from Assets o]

Project Properties - C:\Sergey\Projects\UPC\SIF\ESR\ServiceModel\ServiceModel.jpr

Deployment — ● Use Project Settings

Deployment Profiles:
- ESR_ServiceModel_ejb1 (EJB JAR File)
- AssetService (EJB JAR File)
- AssetServiceClient (Client JAR File)

Deploy AssetServiceClient

Deployment Action

Select a deployment action from the list below.

Deploy to Application Server
Deploy to JAR file.

Building the Assets session bean

A single function in **AssetServiceBean**, **List<Assets> getAssetsFindAll()**, can be used directly by Client or OSB Callout. However, it would be better to create one more Java class in the EJB project, accepting output from this function, do the simple alteration in a list, and return it (we will do the filtering of **Assets**, **getRuleAssets()**, but you can use your own implementation). Also, please copy the EJB's JNDI name from the `Client` class as shown in the following code snippet; we will use it later in the OSB:

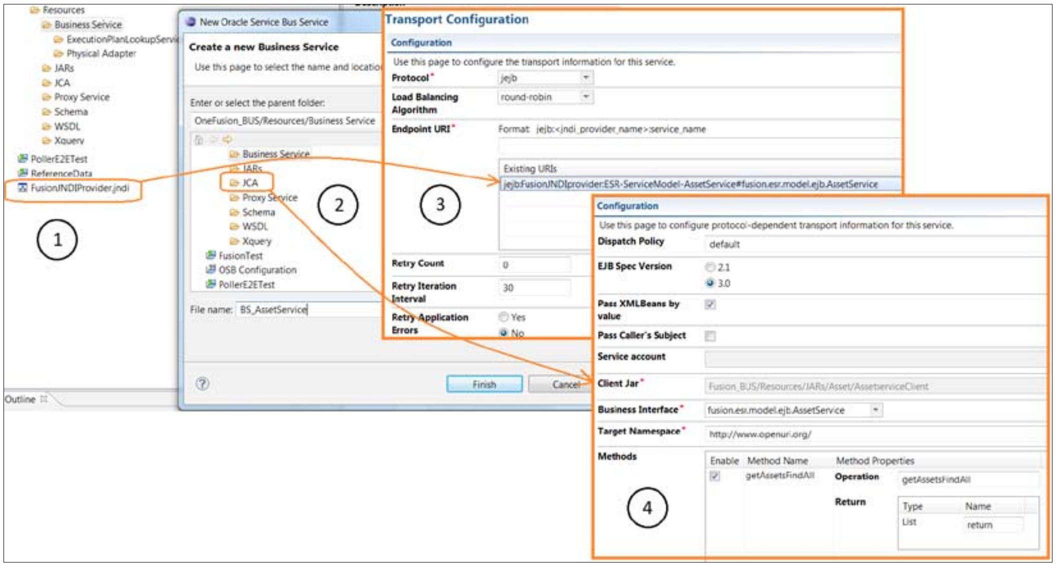
```
AssetService assetService =
  (AssetService)context.lookup("ESR-ServiceModel-
  AssetService#fusion.esr.model.ejb.AssetService");
```

Deploy the `Client` class as a jar. It will be used in the OSB Business Service (copy it to the JAR OEPE project folder).

We're done with JDeveloper and now we can proceed with the OSB part. We assume that you already have a JNDI provider configuration file in the root of your OSB project. If not, you should create one now, pointing to your development server (usually `http://localhost:7001`, for a more complex clustered JIT environment, you could have `t3://jitosbhost:<port>, jitosbhostmirror:<port>`) **(1)**. Now we are ready to create Business Service.

Go to the **Business Service** subfolder, select **Business Service Wizard** from the context menu, and name it accordingly **(2)**. Select **Transport Type Service** in the **Service** tab below and then go to the **Transport** tab. Set protocol as **JEJB** and set the Endpoint URI as a concatenation of your JNDI Provider's name and JNDI name from the `Client` class mentioned earlier **(3)**.

Click on **JEJB Transport type**, set **EJB Spec Version** as `3.0`, and click on **Browse** for the Client jar. Go to the Client's jar location, pick it, and validate the client's method in the **Methods** sections. For simplicity purposes, we have created it with only one method. You will see the EJB-based Business Service created in the **Operation** field **(4)**.



Create a new Business Service

Use this page to select the name and locatio

Enter or select the parent folder: OneFusion_BUS/Resources/Business Service

File name: BS_AssetService

Transport Configuration

Configuration

Use this page to configure the transport information for this service.

Protocol*	jejb
Load Balancing Algorithm	round-robin
Endpoint URI*	Format: jejb:<jndi_provider_name>:service_name Existing URIs jejb:FusionJNDIprovider:ESR-ServiceModel-AssetService#fusion.esr.model.ejb.AssetService
Retry Count	0
Retry Iteration Interval	30
Retry Application Errors	○ Yes ● No

Configuration

Use this page to configure protocol-dependent transport information for this service.

Dispatch Policy	default
EJB Spec Version	○ 2.1 ● 3.0
Pass XMLBeans by value	☑
Pass Caller's Subject	☐
Service account	
Client Jar*	Fusion_BUS/Resources/JARs/Asset/AssetserviceClient
Business Interface*	fusion.esr.model.ejb.AssetService
Target Namespace*	http://www.openuri.org/
Methods	

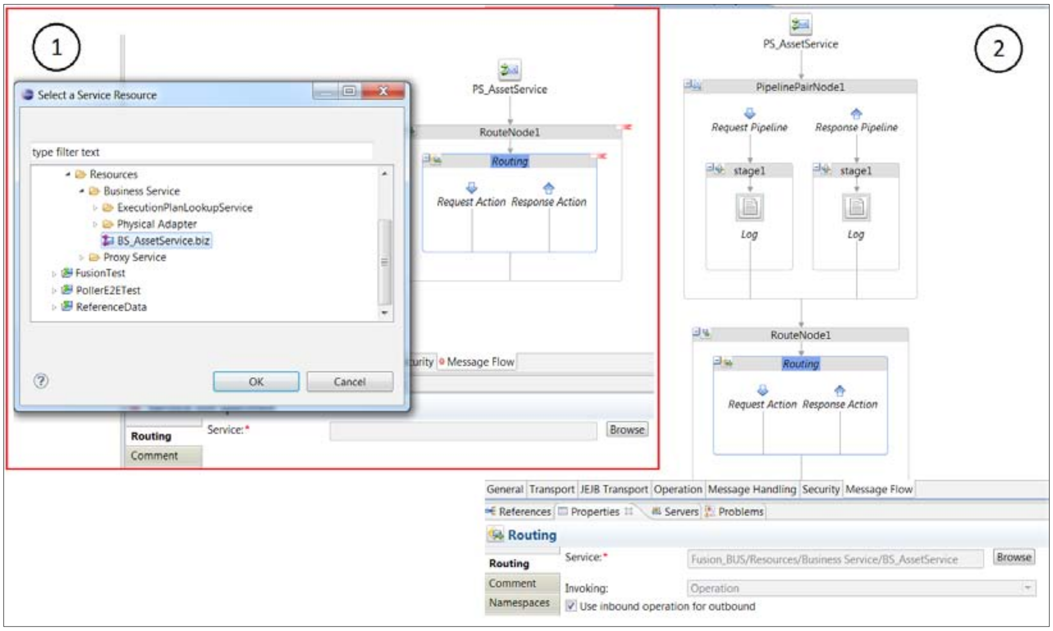
Enable	Method Name	Method Properties	
☑	getAssetsFindAll	Operation	getAssetsFindAll
		Return	Type: List Name: return

Deploying the EJB service on OSB

Now, with another few quick steps, we will proceed with the creation of Proxy Service from our EJB Business Service. Go to the **Proxy Services** subfolder and start the **Proxy Service** wizard.

By the way, you can have any folder's structure according to your preferences, but we advise you to look closely at the structure presented in step one in the previous figure. It seems to be a minor thing, but it has considerable impact on team collaboration, bundle deployment, and Governance in general. The presented structure is strongly conducted to the one old Oracle standard we omitted in *Chapter 2, An Introduction to Oracle Fusion – a Solid Foundation for Service Inventory* for brevity—the **Optimal Flexible Architecture** (**OFA**). The role of this standard is really hard to overestimate and DB architects and DBAs know it really well. For OSB/SCA, in terms of the Adapter framework, the purpose of this standard is to identify the location of OSB and JDeveloper artifacts. For some, as we mentioned earlier, two developer tools around one framework seems too many and you can find the examples where the structure of the JDev SCA/JCA projects is embedded into the Eclipse OSB folders structure. Although we can see the reason for that, we still advise you to keep these structures separate and use the Continuous Integration and delivery bundling tools for the final assembly. Also remember that OFA principles are extremely important for the FTP/FSO TP folders hierarchy.

Name your Proxy service suitably and set it as **Transport Type Service** (the **General** tab). Transport shall be JEJB, similar to the Business service. Set an endpoint URI as you want or leave it as proposed. JEJB properties are the same as for Business Service in step 4, please choose the same Clients jar. Leave all values in other tabs as proposed by default. Now we are ready to implement the Message Flow around this EJB. Refer to the following screenshot:



Service:* Fusion_BUS/Resources/Business Service/BS_AssetService

Invoking: Operation

☑ Use inbound operation for outbound

Implementing the EJB service message flow

Go to the **Message Flow** tab and right click on the **PS_AssetService** starting point. Add the Route node (as we want to route to our Business-Service-EJB wrapper) and create the routing activity. Select the business service created earlier using the **Browse** button **(1)**. Please use inbound operation for the outbound call, as presented in the preceding screenshot.

Now you can add pipeline pairs above and for the sake of appearances, add two stages for request and response, and establish logging activities in each. Surely, it is better to give meaningful names for every stage and operation, but that's not important in this example and therefore skipped. You can log the entire message body for the request operation and for the response. Now we are ready to publish this Proxy Service on our OSB **(2)**.

After publishing this service on OSB, you can find the JNDI name of the proxy in the WLS JNDI tree (**Environment | Servers-<your_Server_name> | View JNDI Tree (new window) | <Name_of _EJB_Proxy>**). Copy the value from the **Binding Name** field and paste it into the `context.lookup` operation in the `Client` test class. Now when you run the test, you will see that all requests are going through the Proxy; check the response in the OSB console.

One last thing left to complete this quick demo is to add another stage in the response pipeline, we call it FilterResponse, and add the Java Callout activity. In the **Method** field, browse the Client jar, where `getRuleAssets()` is implemented, and select it as we are going to invoke it. In the **Action** field, choose the Expression builder and as an inbound parameter, select the entire response, and set the output to the new variable in the **Result Value** field. Add **Replace** operation after Callout in this scope and select **Replace node contents** (body with XPath expression for Response return value) by variable from the previous step.

Redeploy it on OSB and run the Client again. You will see the modified results according your implementation.

This quick and basic demo just demonstrated how easily we can decouple any EJB invocation using OSB and although it's not really an adapter yet (just a proxy, as explained in *Chapter 4*, *From Traditional Integration to Composition – Enterprise Business Services*), we will further demonstrate how an EJB approach can be used for the dynamic Adapter framework. As you realize, functionality within the bean can be much more complex than just filtering the Java list of values. We can encapsulate the entire DB call in a DB-agnostic way.

Traditional DB Adapter implementation

Any ABCS elements should be abstracted through OSB as we have learned in the previous chapters. Let us repeat that again for better modularity and composability. We should avoid direct Adapters connection to the Enterprise Business Flows. In other words, our task-orchestration services must not become hybrid; otherwise, business logic will be polluted by TP API specific logic. The following example is based on an absolutely standard Oracle technique, so for brevity, we will not overburden you with screenshots, as they are quite obvious. Three tools will be required—JDeveloper and Eclipse with OEPE for development and SoapUI ideally for testing.

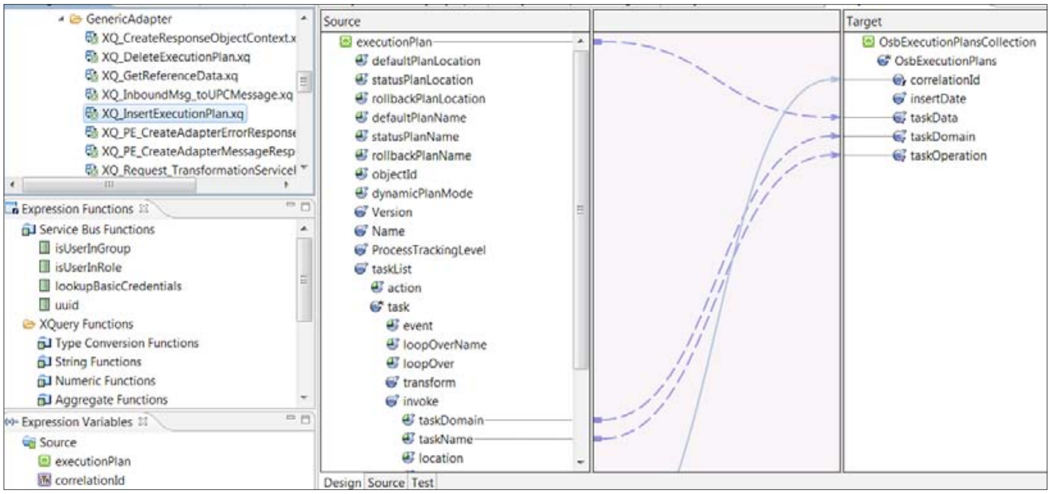
> Do not follow this example right away, please read it first.

This is primarily an architectural exercise (although every step can be useful for an OSB OEPE developer):

1. Let's start with the creation of DB Adapter in JDev. Create a new SOA Project and choose the **Empty Composite** template.
2. Drag **DB Adapter** from **Service Adapters** to the left swimming lane; it will start **Adapter Wizard**. Use a DB of your choice. We will use the **Service** table from a custom ESR realization from the previous chapter. Give a meaningful name to your adapter.
3. As all adapters are WLS resources, JDBC Application Modules shall be established for our application server. Technically, the JNDI is the lookup mechanism, allowing you to dynamically resolve the connection property of a JDBC Connection object by JNDI alias. The connection name shall be unique. Remember, WLS Connection Factories are deprecated.
4. Use a DB connection from the Offline DB ESR project or create a new one. Provide a JNDI name, which we will use on our Application server.
5. For simplicity, use only the **Select as a DB Operation** option on Service Table. We can maintain relations between Service and Service Engine tables. We also can apply the attribute filtering.
6. We can define select criteria, selecting only Rule functions, based on the DB Rule execution engine (NDS).
7. Finalize the wizard, generating WSDL, XSD, JCA configuration, and TopLink mappings. We will use these files in our OSB project.
8. Start OEPE and create a new OSB project and import JCA, XSD, and WSDL files (right click on a project and select **Import**) from the JDev project root. If the JCA file will be invalid after import, just open it and fix the path according to the new location.
9. Generate Business Service by a right-click from JCA. We will leave all generated parameters for transport, message handling, and policy as is.
10. Now we need to create proxy service in order to complete the decoupling. We already have WSDL from the SCA DB adapter, but we really would like to hide the DB-related implementation and present the true decoupling.
11. Start Proxy Service wizard and give it a suitable name. Add Routing node and browse to the Business Service created earlier, similar to the previous example.

12. By changing the ExecutionPlanLookup service from a File-based to DB-based implementation, we have created a complex view for consolidation of all relevant service metadata, stored in the **Enterprise Service Repository** (**ESR**) into complete **Execution Plan** (**EP**). As the ExecutionPlanLookup service is the utility service, we can add an adapter into composition without compromising SOA principles. We also preserved the file-based implementation (together with the Oracle Rule Engine EP's resolution) and can switch between File-based and DB-based implementations using configuration settings or the MH parameter (tracing level element). View, Query, and In/Out Transformations on Adapter are quite heavy (although not complex), so we will skip the detailed implementation and trust that you can implement it yourself, using the presented taxonomy and DB schema, if it will suit your needs. The point here is should we move the Transformations into OSB, leaving Adapter as a bare DB access point or keep transformations to the canonical EP (or any other Canonical Message in your case) inside adapter? Actually, we answered this question earlier, but let's look at the technical possibilities.

13. While working with OSB, you must have definitely noticed two things. Despite the wide range of message manipulation techniques, XQuery is the most common, and secondly, XQuery mapper is rather minimalistic compared to the XSLT mapper in JDeveloper. Still, this mapper is quite capable of doing the quite complex operations and you can access it in OEPE by navigating to **Window | Open Perspective | XQuery Transformation**. In the following screenshot, you can see the actual EP mapping example that we used in the Synchronous Service Broker realization (on OSB) to create simple task collection:



XQuery Transformation perspective

14. After changing the perspective, you can create two mappings for the Request and Response actions in your routing. First, we will map the MH Request elements to the inbound parameter used to select necessary rule functions from repository, and secondly, transform the DB Response to the canonical EP (remember, this is just an example).
15. Constructed XQueries will be used in the Expressions of Replace actions, located in the Request and Response pipelines respectively, by replacing the node content of `<soap-env:Body>`. We have to specify, in the parameter binding, the element that presents the Response from the adapter.
We are almost there.
16. We need to test it. Publish Proxy on OSB and start SoapUI. Get the URL to the Proxy WSDL (`http://<your_OSB_host>:<your_OSB_port>/<your_OSB_project_step8>/<ServiceName>?wsdl`) and paste it into the **Initial WSDL/WADL** field. Generate the Request, change the content of the query according to your DB data, and send the Request. This is it.

In a quick pace, but with all necessary details kept intact, we demonstrated the traditional approaches of the implementation and Adapters isolation using OSB. Although EJB wrapping into Proxy is not completely an adapter-related task, it's quite common and can give you a flavor of what you can achieve by the power of EJB methods, hiding the complexity of Java manipulations, and exposing them in one simple method.

Some more OSB aspects shall be mentioned in relation to the Adapter framework—the ability to support transactions. The JCA transport is transactional. If a JCA proxy service is invoked in an EIS transaction, or if a JCA business service is invoked in a transaction, the transport propagates the transaction. Sometime ago, Oracle had got in its arsenal, which is one of the most powerful Transaction Coordinators, Tuxedo (`http://www.oracle.com/technetwork/middleware/tuxedo/overview/tuxedo-and-soa-bwp-128150.pdf`), and if you need complete transactional support for your applications, this could be the perfect choice. Still, the Oracle Service Bus can provide adequate transactional support for our adapters. A transactional adapter has the potential to start or enlist in a global transaction context when processing a message. The following examples from Oracle documentation illustrate how transactional properties vary depending on the adapter:

- A JMS proxy service that uses the XA connection factory is a transactional endpoint. When the message is received, the container ensures that a transaction is started so that the message is processed in the context of a transaction.
- A Tuxedo proxy service may or may not be a transactional endpoint. A Tuxedo proxy service is only transactional if a transaction was started by the Tuxedo client application before the message is received.

- While an HTTP proxy service will not typically have an associated transaction when invoked by an HTTP client, you can set an option in the HTTP proxy service configuration that starts a transaction and executes the message flow in the context of that transaction.

You can read more on these use cases at `http://docs.oracle.com/cd/E23943_01/dev.1111/e15866/architecture.htm`.

Dynamic Adapters implementation and DB Transport Adapter

Now we have everything we need, we can go to the core skipping the implementation technique details explained earlier. What we do want is to execute any SQL statement on any SQL (Oracle) database in our infrastructure and we would like to do it in a dynamic way based on the information provided in the message container shown as follows:

1. Probably the simplest way to execute the dynamic SQL statement is to use the `execute-sql XQuery` function directly in the Proxy Message flow, in a request pipeline, presented as follows. You will need two input parameters; one for the query, and another for DB source name. Although lightweight and simple as is, this method has some limitations—we can execute only single statement and only `SELECT`. The last one is quite a considerable drawback as it would be quite difficult to execute complex transitions (although possible, depending on your SQL skills). We saved the following code as a standalone XQuery script and will use it further in our flow:

```
Declare variable $database  as xs:string external;
declare variable $query  as xs:string external;

declare function xf:xq_runQuery($database as xs:string,
$query  as xs:string)as element(*) {
 <out>
  {fn-bea:execute-sql($database, xs:QName('rows'), $query)}
 </out>
};

xf:xq_runQuery($database, $query)
```

2. Secondly, we can implement the Session bean with one main method, `executeSQL`, demonstrated as follows. The EJB implementation is pretty straightforward and EJB will be invoked using the EJB Transport, similar to what we discussed earlier. Again, it will accept a datasource and a SQL statement list. After establishing the connection, it will loop over the SQL statements in the same transaction. If one transaction fails, all of them will be rolled back:

```
@Override
@WebMethod
@WebResult(name = "sqlResponse")
public CTUFusionDBAdapterResponse executeSQL(
   @WebParam(name = "SQLStatements")
CTUFusionDBAdapterSQLStatements sqls,
   @WebParam(name = "datasource") String datasource) {
   int x = 0;
   CTUFusionDBAdapterResponse response = null;
   try {
        connection = getDynamicConnection(datasource);
        Statement statement = connection.createStatement();
        String resultStr ="";
        String[] sqlStrings = sqls.getSql();
        for(x=0;x<sqlStrings.length;x++){
            boolean result =
              statement.execute(sqlStrings[x]);
            if(!result){
                resultStr+=Integer.
                  toString(statement.getUpdateCount());
            }else{
                resultStr+="S";
            }
            resultStr=(x<sqlStrings.length-
              1)?resultStr+"|":resultStr;
        }
        response = new
          CTUFusionDBAdapterResponse("0","",resultStr);
        statement.close();
        connection.commit();
        connection.close();
    } catch (SQLException sqle) {
        sqle.printStackTrace();
        log.severe("SQLException,
          ["+sqle.getMessage()+"]");
```

```
            return new CTUFusionDBAdapterResponse("SQL-
              1","SQLException, SQL@position["+x+"]
              "+sqle.toString(),null);
        } catch (NamingException e) {
            e.printStackTrace();
            log.severe("NamingException resolviing datasource,
              ["+e.getMessage()+"]");
            return new CTUFusionDBAdapterResponse("Sys-1",
              e.toString(),null);
        }finally{
         try {
            if(!connection.isClosed()) {
                connection.rollback();
                connection.close();
            }
         } catch (SQLException e) {
                log.severe("SQLException clossing the
                  connection, ["+e.getMessage()+"]");
                e.printStackTrace();
         }
        }
      return response;
   }
```

3. Speaking of establishing the dynamic connection (`getDynamicConnection`), the implementation is presented as follows, just to complete the example:

```
InitialContext context = new InitialContext();
DataSource source = (javax.sql.DataSource) context.
lookup(datasource);
connection = source.getConnection();
connection.setAutoCommit(false);
return connection;
```

4. Finally, to cover all EJB related details, please see the following `CTUFusionDBAdapterResponse` class implementation; `setters` and `getters` are omitted here for brevity:

```
@XmlRootElement(namespace =
  "urn:com:telco:ctu:la:adapter:dbadapter:v01")
public class CTUFusionDBAdapterResponse implements
  Serializable  {

    private String errorCode;
    private String errorMessage;
    private String result;
...
```

Steps 2 to 4 are related to the standard EJB implementation and DB handling in Java. For all the necessary details for implementation, you will find in Oracle documentation and tutorials at `http://docs.oracle.com/javase/tutorial/jdbc/basics/sqldatasources.html`.

5. Now, as we have all executable modules, we need to address the inbound message structure as a carrier and provider of our inbound parameters. As already mentioned, we will use our standard message container CTU Message, and all Protocol Adapter details will be injected into the payload from the Execution Plan. Adapter-specific parameters, injected from EP are highlighted in the following example. As you can see from the ESR DB schema presented in the previous chapter, values for these parameters are taken from the MEP and Endpoint table, related to the application, action as a TP in our service composition:

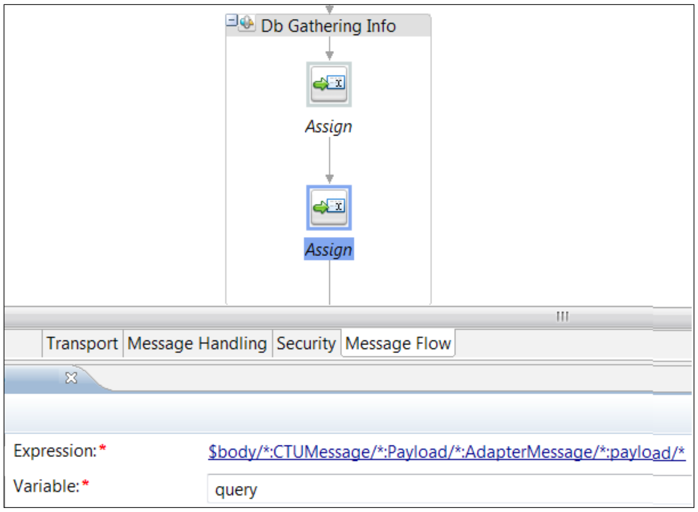
```
<AdapterMessage>
 <endpoint>{JNDI name of DB connection}</endpoint>
 <protocol>{DB}</protocol>
   <payload>
      <SQLStatements>
       <Sql>SQL Statement 1</Sql>
       <Sql>SQL Statement 2</Sql>
       <!— ....->
       <Sql>SQL Statement N</Sql>
      </SQLStatements>
    </payload>
</AdapterMessage>
```

Now, with actual workers (executable modules, XQuery, and Java), and input and output parameter structures, we can proceed with the assembly of our DB protocol Adapter as a Service Proxy.

6. Let's return to our Generic Adapter implementation from *Chapter 4*, *From Traditional Integration to Composition – Enterprise Business Services* and look again at Operate function. It invokes Facade (actual dispatcher) using dynamic XQuery- and DB-related part and is presented as seen in the following code snippet. Now you can clearly see the complete picture of how dynamic routing works and what parameters in message container we employed for that:

```
...
if (data($request//*:AdapterMessage/*:protocol)= 'DB')  then
  ( <ctx:route>
    <ctx:service isProxy='true'>CTUFusion_BUS/Resources/Proxy
  Service/Logical Adapter/
           ProtocolAdapter/PS_Database_Protocol_Adapter</
ctx:service>
    </ctx:route>)
...
```

7. OSB Project was created and a new Business Service wrapping SQL handling EJB was deployed in a similar manner as we explained in the *Exposing EJB through OSB* section previously. A new Proxy Service (as a Messaging Service) was created, based on this Business Service with XML Messaging based on the CTU Message container (Messaging tab).
8. The first step in a Proxy Message flow (right after the logging of an inbound message, of course) would be a variable creation and assignment for SQL query and datasource. We will use it almost immediately.



Expression:* $body/*:CTUMessage/*:Payload/*:AdapterMessage/*:payload/*

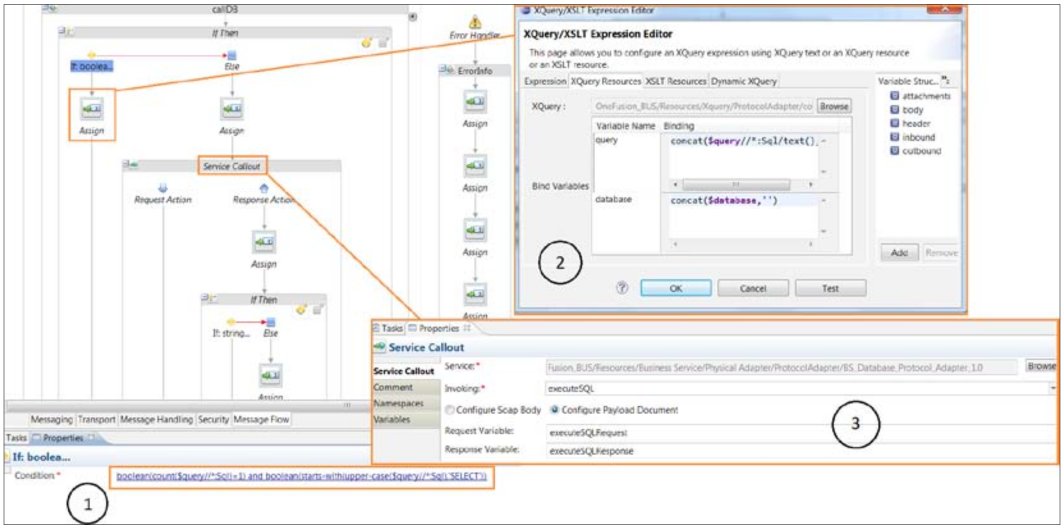
Variable:* query

Assigning a SQL query from message payload

9. The SQL execution part is quite similar to the flow we implemented for a simple EJB in the previous exercise, except that we have two more things. First, we have to implement IF branching, routing the simple queries to the XQuery function execution (step 1) and all others to the EJB Service Callout. The condition is as presented on the following screenshot, **(1)** (it should be only one statement and only `SELECT`). Secondly, for the complex queries, we have to prepare a SQLStatements List and datasource parameters for the Java invocation. See the following script:

```
<urn:executeSQL
  xmlns:urn="urn:com:telco:ctu:la:adapter:dbadapter:v01"
  xmlns:java="java:ctu.fusion.adapter">
   <urn:SQLStatements>
      {
          for $sql in $query//*:Sql return
            <java:Sql>{data($sql/text())}</java:Sql>
    }
        </urn:SQLStatements>
       <urn:datasource>{$database}</urn:datasource>
</urn:executeSQL>
```

The complete SQL execution part is presented in the following screenshot, where step 1 is the condition for our `If` branching, step 2 is the `Assign` operation, executing our single `SELECT` XQuery, and step 3 is the EJB Service Callout:



If: boolea...

Condition *: boolean(count($query//*:Sql)=1) and boolean(starts-with(upper-case($query//*:Sql),'SELECT'))

XQuery/XSLT Expression Editor

This page allows you to configure an XQuery expression using XQuery text or an XQuery resource or an XSLT resource.

XQuery : OneFusion_BUS/Resources/Xquery/ProtocolAdapter/co

Bind Variables

Variable Name	Binding
query	concat($query//*:Sql/text(),
database	concat($database,'')

Service Callout

Service:* Fusion_BUS/Resources/Business Service/Physical Adapter/ProtocolAdapter/BS_Database_Protocol_Adapter_1.0

Invoking:* executeSQL

○ Configure Soap Body ● Configure Payload Document

Request Variable: executeSQLRequest

Response Variable: executeSQLResponse

Service Callout
Comment
Namespaces
Variables

Maintaining the EJB service callout

10. Finally, in our Response flow, we generally have to replace CTUMessage/Payload (body) with result of the Query execution.

That's all folks. In ten relatively simple (and most importantly—standard) steps, we created a lightweight and very universal DB adapter, completely agnostic and suitable for any Composition Controller. The main enabler of this technique is *not* the OSB's XQuery engine or EJB, as they are absolutely standard and work exactly the same way on other ESBs (RedHat FUSE or ServiceMix/CXF, for instance, where we have a similar approach to Service Bean invocation from the XML blueprint, acting as an EP), but the Service Repository with taxonomy, adapted for runtime discovery and invocation of any Service / Service Artifact.

Summary

We are in the middle of the book, but journey is far from over yet. So far we covered the three main SOA Frameworks: Service Collaboration (ESB), Service Orchestration (EBF), and Application Business Connector Services (ABCS, or Adapters) and all SOA Patterns associated with them. We also described the role of Enterprise Service Repository (yet another framework, heart of the SOA Governance) in all of these frameworks. This chapter, entirely dedicated to the ABCS layer had no purpose to cover all aspects of Adapters implementation though, but rather how to optimize this layer in order to make all our service-oriented applications in our inventory more intrinsicly interoperable (one of the major SOA characteristics, you remember).

Why? Well, why does a classic construction brick have pretty much a standard size of 3 5/8". 2 1/4". 8"? Maybe because none of the construction architects in the entire world want to have their masons waste their time adjusting boulders, rocks, and stones instead of building houses and bridges? And further on, maybe that's the way the Pattern-based standardization approach, proposed by Christopher Alexander, becomes so increasingly successful among the software architects as well?

The Extensive Adapter framework is not only a waste of time (and money) during design time and development, but also the constant pain for Operations and Support depts and the reason for more than 80 percent of outages and breakdowns in our operational environments. All middleware specialists know that they are the first people to blame when the message from source A didn't reach destinations B and C. The root cause of that—application A initially was not able to collaborate with applications B and C, and the adapter was simply unable to fix this problem for a complete 100 percent. Even if it could, it's an extra layer, adding complexity to our landscape and therefore making it susceptible to faults.

Thus, our primary architectural task would be to make our core applications more service-oriented for eliminating the requirements for adapters. This approach was demonstrated in the first part of this chapter, when we optimized the OEBS APIs and Event Handling System for the Scandinavian Logistic operator. You were supplied with enough information and code samples for your own implementation of this approach.

Composability and Composition Controllers are the primary goals of this book as this principle and these building blocks are endorsing reuse and modularity (read—cost savings and operational reliability). Therefore, the second part of this chapter demonstrated how we could promote reusability on the Adapter level, balancing adapter-related SOA patterns between OSB and ABCS. Remember, although perfectly operational, this technique should be applied with caution, depending on the number of legacy applications in your infrastructure and the level of their similarity. Sorry, it would be quite irresponsible to give you numbers for selecting any of these approaches, but the CTU example can give you an understanding of what could be really achieved.

Finally, if first two approaches are not possible, you can employ the standard adapter technique, quite well-known from version 10*g* and even earlier, build the individual adapters for all applications, and decouple them using OSB. Some examples for that have been provided as well.

Once again, just in case if someone has an idea—we have nothing against Adapters, we love them. Frankly, it's a good way of making money (although we believe that composing yet another book, full of screenshots from BPEL Creating Partner Link for all type of adapters wizard would be quite useless for you as you probably have enough already). Adapters are inevitable and play essential roles in many solutions, presenting Data Integration, Virtualization, Federation, and Master Data Management. Most of them are based on classic SOA ESB patterns and can be quite effectively implemented on the Oracle OFM (surely, most salesmen of these tools will strongly disagree with us). Anyway, Oracle has a single product called ODI to cover most of these requirements, but that's the subject of a completely different book.

Only the journey is written not the destination. Our next stop will be the Security Patterns and how they can be applied at all levels, from a single Java component up to Security Gateway. We will look at possible threats, attacks types, and the methods of risk mitigation.

7
Gotcha! Implementing Security Layers

Nothing is more vulnerable to any kind of attack than the compositions of different components. In fact, better perimeter protection is one of the tactical advantages of the old silo approach, and no one can deny this. You could protect your service compositions made from the same service domain because you can control it in the same way as a silo; however, if there is a single participant (composition member) outside of the domain's premises, all security concerns will multiply drastically.

In this chapter, we will be faced with quite a few challenges, some of which we have already mentioned. Firstly, native-born security architects have completely different mindsets than solution architects. We cannot ask you to forget all that you have already learned from previous chapters, but we will try to introduce you to another way of thinking using our knowledge of patterns and frameworks.

In about 40 pages, we will do our best to systematically cover all the common techniques and approaches used in SOA security patterns, with references to best practices and publications. Most (if not all) books dedicated to SOA security that you find on Amazon will be dated 2008, 2005, or even 2003, and do not cover the latest standards development (OAuth, SAML, and PKI) and recent tools. Probably the best (in our opinion) paperback, *Securing Web Services with WS-Security: Demystifying WS-Security, WS-Policy, SAML, XML Signature, and XML Encryption*, was published in 2004. Thus, in this chapter, we will not capture the immensity of the SOA security topic and instead focus on layered protection, realized by standard SOA patterns. This layered protection cannot be covered without touching upon SOA-specific attacks aimed at SOA-specific vulnerabilities. Patterns will demonstrate how to mitigate security risks common to SOA implementations.

Where are we now?

We would like to start with a quote from a security report based on research of 110 companies from industries including financial services, the government, and IT. The quote is quite long but really interesting:

- More than two-thirds of IT security resources remain allocated to protecting the network layer, while less than one-third of the staff and budget resources were allocated to protecting core infrastructure such as databases and applications.
- When comparing the potential damage caused by breaches, most enterprises believed that a database breach would be the most severe.
- Nearly 66 percent of respondents said they apply a security inside out strategy, whereas 35 percent base their strategy on endpoint protection.
- Even with this fundamental belief in strategy, spending does not truly align as more than 67 percent of IT security resources—including budget and staff time—remain allocated to protecting the network layer and less than 23 percent of resources were allocated to protecting core systems like servers, applications, and databases.
- 44 percent believed that databases were safe because they were installed deep inside the perimeter.

How old do you think this report is? Twenty years, maybe ten? Not at all. The results of this survey (`http://www.oracle.com/us/corporate/press/1972875`) were published in mid 2013. Take a look at the *Oracle SOA development roadmap* table (*Chapter 2*, *An Introduction to Oracle Fusion – a Solid Foundation for Service Inventory*). *Basic Security Profile Version 1.1* was published in 2009 and this profile de facto has finalized all security standards developed for more than 10 years. You do not have to be a pentester to understand that something is wrong here. The question is, how bad?

No, this is not bad, because the word *bad* is not capable enough to describe how horrible the actual situation is! It simply means that in at least 66 percent of cases, vital information about your clients, financial transactions, planned merges/acquisitions, employees' private data, and strategic development/products is already in the caring hands of your diligently watchful competitor(s). It also means that this information in two-thirds of cases can be acquired within two days without significant investments into complex sniffing equipment. Yet again, in most cases, it's in the best interest of intruders to keep your data intact and hide all evidence of the security breach.

> Without a doubt, the human factor is crucial in any security system; however, discussing this is beyond the scope of this book. Nevertheless, you have to contemplate the fact that about 90 percent of all information leakages are carried out or initiated internally. Yes, sometimes unintentionally, but even good intentions (or the absence of bad ones) provide sufficient basis for anecdotic stories about lost or forgotten notebooks or CDs by agents of one kingdom with allegedly the most proficient secret service in the world. As a result of this contemplation, a clear understanding must be firmly implanted in your mind; in SOA environments, starting from moderate complexity and higher, with more than two service domains and the presence of intermediary, the reliance on TLS/SSL alone is no better than publishing your connection strings with a username/password on your corporate front page.

In this chapter, we will start with an analysis of the situation in order to formalize essential solutions patterns capable of reducing security risks.

Initial analysis

We would like to begin our analysis with an old saying: a chain is no stronger than its weakest link. Banal, isn't it? Sorry about that, but indeed with four architectural SOA levels, everything comes down to the resiliency of the single-service implementation, which is in the lowest architectural service layer. As long as any particular service is a composition of services (Task or Task-orchestrated service), the three basic building blocks of SOA infrastructure can be identified. The first two are Utility and Entity service models, usually employed to compose the Task service(s). The third is a Service engine (which is not a small thing, but you will probably rely on an existing one instead of inventing your own for commercial realization). An Entity service's internal architecture is usually more complex than a Utility service, as it commonly involves DB as the entity's persistence storage. Therefore, from the static implementation standpoint, all four enterprise SOA layers will be protected proportionally to the level of security resilience of the Entity service anatomy, and you certainly remember its every single block:

- The core logic is encapsulated into single or several components. How are you handling exceptions, including `NullPointerException`? Is your code thread-safe? How about memory utilization and stack control, global variables, and so on? If you do not know this, how can you be sure that your main architectural block is safe? A stack/buffer overflow attack is one of the most popular ways of breaking into your system and is commonly focused on components' implementation.

- We already mentioned DB's presence in the service anatomy. The ways of service persistence implementation are utterly crucial for the entire SOA implementation. The absolute champion among all types of attacks is SQL injection, and it has been for years. Combined with improper error handling in the component (see the previous point), it will present grave danger for your business. Another quick check is of the DB account that you assigned for this service. What privilege options are available? DBA, XDB, or both? Another obvious question is: does anyone else access the Service data bypassing the Service Contract (just because it is faster and someone decided to cut some corners)?
- The Service Contract presents quite a substantial set of internal service Facades and Agents, performing service data serialization/deserialization, exposing a component's functions, and enforcing the internal policies. Usually, most of this functionality is easily available from libraries or IDEs, but its openness is not exactly a good thing. Attack spearheads will be pointing to XML/JSON parsers and marshallers, exploiting existing or possible vulnerabilities in standard libraries and XSD syntax.

Here we mentioned only three internal services elements susceptible to attacks, but these attacks are the most common and truly devastating. Common to these types of vulnerabilities and attacks is that their target is static core service logic, encapsulated in component or groups of components. However, this doesn't make them similar to the classic security issues of the silo approach because we have the second native part of SOA—the service interactions.

Services or components have to communicate with each other in order to carry out their business tasks. What's worse for security personnel is that they have to perform it dynamically, depending on numerous business conditions that involve a considerable amount of external resources in an agnostic manner (see the CTU example, discussed in previous chapters). We do not have strict domain security boundaries anymore because one message can carry information about different (even business-opposing) parties. Different parts of the message can be transformed or enriched separately by independent intermediaries and so on. Information Confidentiality, Integrity, Non-Repudiation, and Origin are constantly at risk when we have something in transit, and we constantly do. Thus, certain measures (in the form of Patterns) shall be applied in order to protect the aforementioned information properties. Luckily, these measures, covered by WS-* specs as Encryption, Digital Signature, and Portable Trust, are quite mature and far older than the SOA concept itself.

Before we go into the risk analysis, vulnerabilities, and SOA-related attacks, we would like to jump ahead to some generic conclusions:

- It is not possible to have poorly designed services at the beginning (usually designed as a PL/SQL Web Service or any web service by right-clicking in JDev) and then convert them into something secure by hiding behind some magical "Secure Gateway" or "Defense Perimeter". At best, the performance of such a service, after applying all security restrictions, will degrade ten times or more (Oracle estimates). This is because a Service Gateway (SG) will have to meticulously screen every single call and validate every single response in order to shield the holes in the service design and intentionally throttle the traffic, as the service simply cannot keep up with the incoming requests. Surely, such a situation will feed the common stories about how the "naughty" security killed our good performance.
- The logical outcome from the preceding point is that you as an architect should ensure that the service design is safe and sound in every single detail and not just by drawing blue boxes in PowerPoint and connecting them by red lines. As mentioned earlier, developing entirely in Java would probably be too much (although this is definitely a positive thing), but you should perform peer reviews and participate in testing at all levels. Frankly, this is not news; please refer to Thomas Erl's book, *Service-Oriented Architecture (SOA): Concepts, Technology, and Design*, where you find SOA architect's role laid across a whole project's lifespan.
- With no magic pill available to mend all security issues at the end of the project, an SOA architect should work hand in hand with a security specialist and be familiar with the current trends in risks, vulnerabilities, and attack types. OWASP (`http://www.owasp.org`) is definitely one of the best places to go, and all our further analysis will be based on the classification proposed by this project.
- We have already mentioned one common security design rule generally associated with encryption and digital signature—algorithms are widely open, keys (private of course) are utterly protected. This statement not only stresses the necessity of rigorous testing of a security's crucial elements but also denotes the considerable risk associated with having something custom-built (in-house) as the central part of your security infrastructure. The security is probably the one (very conservative) area of IT where having your own private opinion could be an expensive luxury indeed.

You don't have to build everything from scratch. There are plenty of appropriate tools and libraries and you, as an architect, should just put them (or apply the patterns) in the right place. Again, a good starting point could be to maintain OWASP's terms and terminology within your team—the common understanding of spoofing, surreptitious forwarding, stack smashing, and so on. For the same reason, we will just follow the already proposed classification, avoiding unnecessary reinvention.

> To illustrate the risks of having a poor understanding of security design, a highly respectable IT company, and pioneer in event processing and BPMN, participated in CTU's RFI process. Lacking the COTS market-proven security solution, this company proposed a custom package, developed for other customers over several years. The proprietary Security Perimeter was proposed to the completely stunned architects where scans for the threatening content was executed after authentication. Further still, the scanner itself was based on the standard XML parser. Having said that, in this chapter, we will not present you with the custom solution as we did before for ESB and adapters. Instead, we will talk about the API Gateway, a relatively new Oracle strategic product capable of covering five out of eight common SOA security patterns.

From the proven SOA Pattern Catalog, we have eight security patterns: four for service implementation (Group I for static service implementation, starting from **Exception Shielding**; `http://soapatterns.org/design_patterns/exception_shielding`) and four to protect service interactions (Group II for service messages in transit). Some of the patterns such as **Data Confidentiality** and **Data Origin Authentication** (group II, see `http://soapatterns.org/design_patterns/data_confidentiality`) are in fact the direct realization of the WS-Security standard, WSS (see `https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss`), covering two main W3C specifications: **XML Encryption** (`http://www.w3.org/TR/xmlenc-core/`) and **XML Signature** (`http://www.w3.org/TR/xmldsig-core/`). Since they are of the utmost importance, they are well covered in all the books we have already mentioned, so we will not focus on them much.

The Trusted Subsytem in its turn (group I) is a security-related extension of the Contract Centralization pattern, designed to prevent access to service resources, bypassing the standard service contract and associated security policies. Thus, there are five patterns left: two related to Authentication and Authorization and three that can be implemented (centralized) by the so-called Security Perimeter. To understand their roles and importance, we will proceed with the most common vulnerabilities first.

Common SOA vulnerabilities

Think of military operations in close encounters. Your service domain has a certain edge that is in one way or another exposed to the world outside. (If it's not exposed, you are the lucky one! Why are you reading this?) This edge is actually the skirmish point, where the attacker methodically shoots and observes, trying different weapons. A publicly exposed contract (WSDL/REST) gives enough information for the imposter to devise the initial weapon of choice, for example, service operations or an XSD structure—everything is there, so an attacker has an undisputed tactical advantage. Vaguely defined XSD (types `any` or `string` for all elements) just makes more room for experiments (in case your attacker is bored) and intercepted valid messages can give the attacker a quite a good understanding of the possible data ranges.

One option is to not expose the WSDL definition (hide the API documentation or at least remove all the comments from WSDL). Well, the secluded contract is not really inline with the Composability principle. However, what about your UDDI? It is similar to the DNS server for IP networks and susceptible to similar attacks. UDDI-crawling attacks can turn your own versioning strategy against you if you keep old, less secure contracts available for backward compatibility.

In any case, the attacker will be looking for a response from your service—the more elaborative the better. Ideally, a complete error stack trace is what the attacker is dreaming of (an entire unhandled SOAP error with `faultString` is good enough), but in fact, any piece of information is welcome. Needless to say that the biggest portion of this information will be provided by your Error Handlers (EH) at all levels; therefore, vulnerabilities in EH design shall be discussed first.

It would be a mistake to think that the standard HTTP response or no response at all will considerably improve the overall security. Blind-type injection is one of the most difficult injections for attackers, but it is still quite able to deliver results and standard responses can be mapped to attack types and they are informative enough.

Surely our goal is to make the attacker's life hard, but our developers and operation will be proportionally affected as the Discoverability principle is sacrificed.

Error handling vulnerability analysis

Here, we will combine the most common error handling vulnerabilities, allowing attackers to explore your line of defense and collect all the necessary technical information about your services for further steps, aiming at authentication/authorization weaknesses. The methods are quite obvious:

- Study exposed contracts and/or intercept valid messages (being a passive intermediary or eavesdropping).

- Check the message for potential cryptographic nonce, constructed as the concatenation of random string and timestamp and used once per message. If the message contains additional information regarding the valid time range (for instance, `return_acknowledge_till<…>`), try to resend the message (as an active intermediary) within this interval. Interestingly enough, it's not that rare that for an `add<something>` operation, the attacker gets a message acknowledging the error, with `faultString` containing the primary key violation along with the constraint name, table name, and some additional information about DB itself.
- The presence of a nonce technically means that the message is signed (HMAC-SHA or older MD5); otherwise, this composite nonce doesn't make any sense. Here the attacker has two options: it's quite possible that a small clock synchronization interval can be maintained between the involved systems, potentially allowing reply attacks. If the weak hash algorithm is used, the attacker can exploit it by using brute force or a collision technique (MD5, `evilize` library; for an example by Peter Selinger, see `http://www.mathstat.dal.ca/~selinger/md5collision/`). Anyway, these attacks will be undertaken after studying the message structure and the response.
- It's also not as rare as you might think (yes, it sounds unbelievable) that the XML digital signature cannot be strictly enforced by a contract's WS-Policy. This could happen during the transition period, when some migrating consumers are not ready to be fully compliant with the declared policy. So, all the XML `ds:nodes` containing the following code can be easily stripped by the attacker and a message will still be accepted:

  ```
  <SignatureValue>
  WkZUJAJ/0QNqzQvwne2vvy8U5Pck8ZZ5UTa6pIwR7GE+OoGi6A1kyw==
  </SignatureValue>
  ```

The important thing is that whatever hacking technique is employed by the attacker, the initial step is always the same: gather as much information from your service response message as possible. Your Error Handler is the major supplier of this information.

Regarding the following vulnerability list, feel free to use your own labels (alphanumerical codes) for all kinds of vulnerabilities. As you go further, you will need them as flags to mark potential weaknesses on the technical infrastructure map.

Information leakage

The vulnerability code for information leakage is EH01. Take a look at the common catch block:

```
catch (Exception ex){
ex.printStackTrace();
System.out.println(ex);
}
```

What is good for the logfile in a Dev or JIT environment is a disaster in production and definitely should be avoided in a SOAP error response.

Missing error handling

The vulnerability code for missing error handling is EH02.

Standard HTTP response codes 4XX (Unauthorized, Bad Request, Forbidden, Not Found, Method Not Allowed, and so on) are quite often employed in REST-based APIs. Firstly, they are already quite informative for the attacker. Secondly, using standard handlers instead of fine-tuned handlers in your services could not only potentially reveal service technical information (presence of resources, class/resource hierarchy, and methods availability) or service business logic, but can also make the life of your developers difficult if mapping is too generic.

We also have to mention that redirection of everything to a single generic error page or attempts to map 403 to 404 and so on (with the intention of deceiving the attacker) will primarily affect service consumers and ops, not the culprit.

Empty catch block/uncaught exception

The vulnerability code is EH03. The following code is another quite common block:

```
try {
          provideService();
}
catch (SomeUnusualException ex){
       // our system is quite resilient and we do not care about
low-probability cases
}
```

Note two things here: practical—if this unusual exception ever occurs, you will have no information about it, and philosophical—you better believe it will happen. A developer's laziness is the attacker's best friend. Peer review is an essential part of an architect's tasks. Some tools from the following section can help you.

Catching NullPointerException

The vulnerability code for catching `NullPointerException` is `EH04`.

Catching `NullPointerException` doesn't make much sense. Generally, this is not a runtime exception we should handle in a `catch` section. In general, it's the lack of coding culture (note, we are not accusing anyone) and if you have filthy code, none of the existing perimeter protection systems or IDS can protect you. Sorry, that is just the way it is. Here are the tools you can use for static and dynamic analysis of your code:

- Static and dynamic (with heap walking) analysis:
    - Oracle JRockit Mission Control: `http://www.oracle.com/technetwork/middleware/jrockit/overview/index-090630.html`
    - JProfiler: `http://www.ej-technologies.com/products/jprofiler/overview.html`
- Static analysis:
    - FindBugs: `http://findbugs.sourceforge.net/`

Return inside the finally block

The vulnerability code for the `return` statement inside the `finally` block is `EH05`.

All errors that might occur or are thrown in the `try` block will be ignored by the `return` statement in the `finally` block:

```
Object ObjectHandlingMethod() {
        Object o = null;
         ....
        try {
      o = MethodErrorThrower();
        }
        finally {
      CleanUpRoutines();
      return o;
  }
     }
  Object HandlerErrorThrower(){
          ...
          if (size == 0) {
        throw new EmptyStackException();
         }
        catch{
          logerror()
       }
    ...
  }
```

An exception is thrown in the other method, called from the core class. An error was caught and even properly logged, but it wasn't propagated back to the caller. The `return` statement in the `finally` block is choking any exceptions, making the code not only unsafe, but also very hard to maintain.

Inappropriate cleaning

The vulnerability code for inappropriate cleaning is `EH06`.

As you can see from the preceding code, we have `CleanUpRoutines()` in the `finally` block. This is a mandatory part for any final block—all connections must be closed, file pointer released, threads unlocked, and memory cleaned from the loop-related variables. This is highly important as some attacks are not only aimed at revealing technical implementation details but also at depleting the service resources.

Handling dissimilar exceptions in the same block

The vulnerability code for handling dissimilar exceptions in the same block is `EH07`.

This situation is quite close to the one described for vulnerability with code `EH02` and can also be presented by using the same exception handler for differed named exceptions. An indiscriminate handler will not only open the door to the mishandling of complex situations simulated by an attacker, but also will leave an unclear trace record in the system's logs, blinding the response/ops team.

Authentication and authorization vulnerabilities

Here we mostly talk about API authentication (System-to-System) and not just about human and web page interactions (although it's also based on SOAP and REST services, serving direct and brokered authentication). In this case, arguably, you can assume the main authentication weakness of all time—the password strength can be finally tamed as the API is not affected by typical human factors, that is, the ability to remember passwords (or leaving yellow sticky notes on the monitor). Indeed, just from a password strength calculator (you will find a lot of them on the Web), we can see that a four-character password, which is common for humans, can be brute forced in a couple of minutes, while a 20-character API key will take ages (with adequate OS (Linux/Unix/Windows) file protection in place, of course).

Well, our optimism should be very cautious about that though. Not too long ago, we were able to use Amazon S3 and EC2 cloud services (actually, AMIs with all our assets) to log in with our regular Amazon shopping account. So naturally, as mentioned earlier, using signature-wrapping (stripping) and cross-site scripting (XSS) attacks, pentesters were able to compromise these Shopping and EC2 credentials and gain total control over the victim's account with virtual machines containing stored code and data. Should we mention the victim's shopping cart? Should we also mention that EC2 is probably the most popular sandbox for developers from several leading companies?

We believe that this vulnerability is already mended by Amazon, but we can add several good recommendations to our list of security design rules from the preceding case:

- Using the same password could be convenient, but that's not the idea behind SSO, especially if cross-system authentication is based on such weak possession or knowledge. Do not mix human- and system-based AA approaches.
- Single-factor authentication should be avoided; two-factor (by the way, Amazon claims to have multifactor authentication, MFA) with one easily compromised factor will give you a false sense of protection. In fact, any systems based on something you know (password) should be enforced with something you possess (token), something you are (for example, fingerprint), or better, password transmission should be avoided completely.

Generally, gaining direct access by exploiting or resending SOAP/REST messages is a kind of gamble and will require not only inefficiency in security design, but also a little luck as well, and usually attackers do not count on that. After the identification of the backend system's types by studying response messages, the attacker will perform the following tasks:

- Try to obtain credential information from service resources (DB and config files) directly.
- If the credentials for this service are obtained, check the system privileges associated with them.
- Explore the possibility to bypass the service contract and directly get to the service resources. Use the known weaknesses of the technical platform (missing latest security patches for Oracle DB, MSSQL, and so on).

For some reason (for instance, license policy), you can decide to use an MS SQL database as the backend for certain critical products such as the Oracle Enterprise Repository. That's perfectly fine, but you (your DBA) forgot to disable `xp_cmdshell` (or better, delete `xpsql70.dll`, `drop EXEC sp_dropextendedproc @functname='xp_cmdshell'`, and forget about its existence). What's worse is that the backend resources are shared (which is common) and some services are used as a Trusted Subsystem with SA privileges. This is the perfect recipe for a successful SQL injection attack as well as an invitation to gain full control over all your resources. These types of attacks are well documented and included in OWASP. Countermeasures that have already been mentioned include not giving your service account more privileges than is really necessary. (Frankly, we all know where it comes from—on a Dev server, our developers focus on functionality first and security granularity later. Please do peer review for the component's code and installation scripts including DB.) In addition to the already mentioned vulnerabilities, the most common authentication and authorization (AA) vulnerabilities are gathered in the following sections.

Simple authentication protocol

The vulnerability code for a simple authentication protocol is `AU01`.

Utilize the Diffie-Hellman scheme to create a session random hash value. This type of authentication is quite susceptible to reflection attacks, that is, when an attacker creates the second handshake session using the challenge obtained during the first (incomplete) one.

Password system exploits

The vulnerability code for password system exploits is `AU02`:

```
String plainText = new String(plainTextIn)
   MessageDigestencer = MessageDigest.getInstance("SHA");
   encer.update(plainTextIn);
   byte[] digest = password.digest();
    if (digest==secret_password()){
//log me in
     }
```

The failure of a password authentication mechanism will almost always result in attackers being authorized as valid users.

Authentication decision based on the Referer field

The vulnerability code is AU03.

The HTTP header element as defined by W3.org is `Referer = "Referer" ":" ( absoluteURI | relativeURI)`. The J2EE code for extracting a fields value for further authentication is `HttpServletRequest.getHeader("referer")`. In fact, the `Referer` field in HTML requests can be simply modified by malicious users.

Authentication decision based on the DNS name resolution

The vulnerability code for the authentication decision based on the DNS name resolution is AU04:

```
import java.net.InetAddress;
public class Authenticator {
public boolean authByHostname (String clientIP) throws Exception {
      booleansafe = false;
     InetAddress address =InetAddress.getByName(clientIP);
      String hostname=address.getHostName();
String canonicalhostname = address.getCanonicalHostName();
                        If (canonicalhostname.endsWith("trustedsite.
com")) {
                             safe = true;
                        }
                        return safe;
  }
}
```

You cannot control external DNS servers. All that is in there (IP/name mapping, cache, and registering APIs) one day can be poisoned and compromised. Traffic can be routed to the ghosts controlled by culprits, where IP addresses, names, and host attributes will be mocked as trusted. Simply put, don't trust anything coming from outside, especially from DNS, and do not base your authentication on these attributes.

Single-factor authentication

The vulnerability code for single-factor authentication is AU05.

Consider a dual-factor authentication as significantly more secure. Increasing the factor will certainly increase authentication resilience, but it could have a negative effect on performance (depending on the number of authentications per second, and it must be evaluated case by case).

Least Privilege Violation

The vulnerability code for Least Privilege Violation is `AZ01`.

The elevated privilege level required to perform operations such as `chroot()` should be dropped immediately after the operation is performed.

Most commonly, this vulnerability is exploited when your service, acting as a Trusted Subsystem, uses the elevated privilege level, accessing the common resources.

File Access Race Condition

The vulnerability code for File Access Race Condition is `AZ02`.

The program checks a property of a file, referencing the file by name.
It later performs an FSO operation using the same filename and assumes
that the previously checked property still holds.

Common SOA risks

Vulnerabilities represent risks materialized (or maybe not, depending on how good we are) through various attacks. Any project (at least in practice) has a risk assessment. Security risks are the main part, especially for services with external exposure. This section is critical and must stay tuned with the current security trends. OWASP is one of the primary sources of trend information.

We took the OWASP top 10 data for 2013 (`https://www.owasp.org/index.php/Top_10_2013`) and mapped it to the standard SOA security patterns, capable of mitigating these threats.

Injection

Anything that can be inserted, implanted, or simply added to the command line or DB query string could not only break data consistency or add unwanted data portion to the dataset, but also execute some DB or OS command, allowing the attacker at the end to gain complete control of the victim's system.

At the very least, the system will respond with an error message, and it is the architect's responsibility to balance the SOA security, abstraction, and discoverability requirements.

Several factors make this risk number one in the top 10: difficulties in mitigation, number of attack-automation tools, and potential devastation.

The following are the suggested patterns to apply:

- **Message Screening for inbound messages**: Implementation of this pattern is based on establishing the Service Perimeter Guard (yet another pattern) in front of the Service Perimeter and firewall. Additionally, it can be applied to a regular ESB.
- **Exception Shielding for responses:** This is for an individual service at the time of design, combined with the Service Broker (composition controller). In addition, it can be applied to the Service Perimeter Guard.

Broken authentication and session management

The factors that can lead to insufficient authentication are reliance on weak passwords, single-factor authentications, unreliable/compromised protocols, algorithms (such as MD5), the usage of digital signature without encryption (and vice versa), too short or repeatedly reused session keys, a simplified nonce, the possibility to compare protected and unprotected messages, revealing/loss of private keys and certificates, storing passwords as clear text in AIM DB, and misconfiguration of trusted subsystems that allow the bypassing of the service contract and accessing the service resources directly.

The following are the suggested patterns to apply:

- Brokered and Direct Authentication:
    - Identity DB/LDAP
    - Policy Enforcement Point
    - Policy Definition Point
    - SOAP Header
    - HTTP Header
    - SAML elements
    - PKI elements (such as CRL)

Cross-site scripting (XSS)

The distributed way of handling a service request is quite common. What's more is a single message can be addressed to separate services in parallel or sequentially. Intercepted (the man-in-the-middle attack) and forged in certain parts, impersonating trusted parties (see Trusted Subsystem pattern, `http://soapatterns.org/design_patterns/trusted_subsystem`), a message can cause session hijacking, malicious redirections, and the exposure of sensitive data. As we mentioned earlier, an attack on Amazon was successful because the application signature verification and XML interpretation were handled separately. Yet again, Amazon is quite protected now against XSS. Are you?

The following are the suggested patterns to apply:

- **Data Origin Authentication (digital signature)**: In this, Service Perimeter Guard can be used as a pattern
- **Data Confidentiality (encryption)**: In this, a service message prompts you to make sure that every individual part of the SOAP message is supplied with the signed digest; the nonce is crypto resilient

Insecure direct object references

The usage of FSO configuration elements is inevitable (for example, XML, INI, or property files), especially close to the skirmish point. The developer's logic is simple: "I cannot use full-fledged DB in DMZ; it's too heavy and has many weaknesses. The amount of config data I will use in my utility service (or agent) dynamically is insignificant and I can use simple XML instead, staying flexible and configurable at the same time." The logic is flawless, but do we have file consistency checks every time we access it? Do we have adequate file protection from the OS side? What if the attacker, by executing a buffer overflow attack, causes a segmentation fault, halts the program execution (exits abnormally), gains control over the program's resources (not exactly root!), substitutes/modifies the file, and lets the system restart the process? This means that the service agent infrastructure (part of the SOA architecture) is equally vulnerable to attacks as are common entity services because agents are common event-driven programs utilized in all service interactions (imagine the agent checking the elements of the `<string>` type for the acceptable length) and, sometimes, their log footprint is so small that you will have a hard time finding the real problem.

The following are the suggested patterns to apply:

- **Trusted Subsystem**

  This pattern will be applied to every service and agent in the SOA infrastructure. This is a joint task for an architect, OS administrator, and security specialist, and must be performed during a peer review. Every single call to a resource shall be validated and tested. As most common attacks here will be related to buffer overflows, you have to decide on want type of code (that is, language) you want to implement your protection, especially close to DMZ—managed (Java) or unmanaged (C).

Security misconfiguration

Speaking of mandatory configuration routines, please do the basic sanity check by asking yourself the following questions: has LDAP synchronization managed with open protocol (not SSL)? Have you applied security patches or are you afraid of breaking something in production (alternatively, have you applied the wrong patch)? Do you encrypt HD with sensitive data? Have you forgotten to update the Certificate Revocation List? Have you run a service under the root privileges? Do you keep your firewall ports configured by default (it's still not a problem to find which are opened, but we do not want to give the bad guys a chance to slack)? Still believe that WEP is unbreakable (in addition, allowing guest WEP-based Wi-Fi access to corporate data)?

This sanity checklist is not complete, but we have no intention to publish a corporate red book with all security do's and don'ts.

Another noteworthy point is that you can play and have a lot of fun with honeypots and honeynets, but please make sure that they are completely (better still, physically) separated from any of your actual environments.

There is no pattern called diligence or vigilance; that's the state of mind of security ops. You, as an architect, must assist with the proper security configuration of the following elements of the SOA infrastructure:

- Composition controllers and subcontrollers
- Concurrent contacts
- ESB / SG engines / agents
- Orchestration engines

In fact, every single element of your SOA infrastructure must not go amiss. An obvious thing to say is that highly reusable components, service engines, and most common service agents must be checked first and on a regular basis.

You also have to include into your Ops red book (security response plan) and orange book (backup/recovery plan), a drill schedule, usually performed on your honeynet.

Sensitive data exposure

From a perspective of common sense, this is not a vulnerability, but primarily a negligence similar to the security misconfiguration discussed earlier and quite common to web applications (remember, we are not accusing anyone). However, for SOA, it has a broader context. Data is not only exposed on static web via AJAX/ REST API. More often than not, it is insufficient crypto-strength or the lack of crypto-protection on sensitive elements of a SOAP message.

Reliance on TLS when intermediaries are on the message path is another example of such exposure, especially if the intermediary is active, that is, involved in message transformation. Even behind the Secure Gateway, in IPC-certified organizations, message data should be encrypted all the way to the ultimate receiver.

The following are the suggested patterns to apply:

- Data Origin Authentication (digital signature)
- Data Confidentiality (encryption)
    - Service Contract
    - Service Messaging
- Resource Data Storage or similar resources

Missing function-level access control

Missing function-level access control vulnerability denotes insufficient authorization. It is not enough to just check whether the user is valid; a system must guarantee that this user will be allowed to call only permitted operations.

The Entitlement Server is an essential part of identity management, and for API management, Oracle Enterprise Repository with Registry (UDDI) synchronization is highly important. In terms of authorization, all your policy definition points should be supplied with connectivity to the IAM Rights/Entitlement store. Needless to say, any client-based validations for authorization do not make sense.

The following are the suggested patterns to apply:

- Service Perimeter Guard
- Service Contract (concurrent contract)
- Service Messaging
- MDM for all identity sources

Cross-site request forgery (CSRF)

This is a kind of manipulation, that is, when a legitimate client is forced to send a request to the service on behalf of a forger. A request can include session-related information including cookies.

Distributing session cookies is a common practice, for instance, in a multiscreen IP TV, when a valid user wants to transmit active session data from the big screen to or her tablet (or vice versa). This situation can be emulated by an attacker in order to catch and analyze the session data.

The following is the suggested pattern to apply:

- **Service Perimeter Guard**

  Actually, perimeter protection is the last resort. Services and especially services-composition controllers must be designed with caution based on all the previously mentioned recommendations for the components design and security configuration.

Using components with known vulnerabilities

Similar to XSS, this kind of design flaw is quite harmful for distributed service activities (although, OWASP considers it as a moderate threat). The dynamic nature of service compositions makes SOA architecture quite susceptible to these kinds of attacks.

Going further, the implementation of the absolutely valid Endpoint Redirection SOA pattern (`http://soapatterns.org/design_patterns/endpoint_redirection`), used for version control and service load balancing, can open the door for such attacks if redirection is done by simple mapping on LB without any perimeter protection with message scanning.

The following are the suggested patterns to apply:

- Service Perimeter Guard
- Composition Controller (subcontroller)
- Enterprise Service Repository

Make sure that all redirects originated and orchestrated by your composition controller (destination endpoints taken from the service repository, the external API URL) are validated separately by SG. Do not accept any redirect parameters from the response message belonging to previous invocations.

If you maintain an invocation list in a message header (message tracking data), make sure that it is assembled from trusted sources.

Attack types

The following are the different types of identity/authentication manipulation attacks.

Reflection attacks

The attack code for the reflection attack is `AT01`. A typical attack sequence is listed in the following steps:

1. The attacker starts a new session and sends a request/challenge.
2. The server responds with an encrypted challenge and its own challenge.
3. In a new session opened in parallel, an attacker sends the server's challenge from the first session (still opened).
4. Naturally, the server understands the second session as new and responds with a new encrypted challenge.
5. The attacker uses the server's response from the second session as its own response for the initial session. With a nonzero probability, the server will accept its own response from the second session as a valid handshake and open the connection.

Identity spoofing

The attack code for identity spoofing is `AT02`.

The identity associated with the message or resource must be removable or modifiable in an undetectable way for the attacker to perform this attack.

For example, after authentication, the valid REST service user will have their token as a part of the URL query string for an extended session. Resource access permission is validated using this token. The user with less privileges from the parallel valid session will obtain this query string (man-in-the-middle, `https://www.owasp.org/index.php/Man-in-the-middle_attack` and eavesdropping, `https://www.owasp.org/index.php/Network_Eavesdropping` and modify the session using the obtained token (or user ID if it is open). They will have time until the session expires to illegally access the resources.

In the query string, the entire static section of it must be signed. The signed digest can be in the HTTP header and enforced by a contracts policy (in the security gateway). In the case of SOAP messages, WS-Security elements for the digital signature and encryption must be applied.

Replay attack

The attack code for the replay attack is `AT03`.

As we demonstrated earlier, replay attack is the attacker's bread and butter. Having constructed using your XSD or intercepted message, the attacker modifies the time range in the message (if applicable), and the sequence numbers (if necessary), and resends it, sometimes stripping the signatures (if the policy allows).

This attack has many variations and is not always intended to be successful at the very beginning; gathering response info is the initial target, including session data (as almost any SOA message exists in a certain session context). An attacker will explore the predictability and randomness of the session ID in order to repeat this attack with a more accurate IDs. Needless to say that the sequential IDs and reliance on date or time ranges is not a really good idea.

This attack can be combined with buffer overflow attacks in order to crash the service completely. An attacker can assume that after the service restarts, cache will be nullified as well, so the new session ID (possibly starting with zero) will be accepted for the same old message. Oracle's distributed cache and clustered environment with many OFM nodes can prevent this, but an attacker could try to shutdown all nodes at at once, or try to get to the Node Manager (especially if it's in a single-node mode).

If the distribution of cookies is involved, cookies reverse engineering can be employed in order to make the reply attack successful.

SQL injection

The attack code for SQL injection is `AT04`.

The true and, therefore, bitter irony here is that due to a lot of DB abstraction layers (including X/O mapping, persistence layers, and even migration to NoSQL DB types) in a service's internal architecture, some experts openly proclaimed a couple of years ago the Death of SQL Injection. As you can see, it's a present-day top risk and all the old techniques we used for old ASP pages are quite powerful for REST and SOAP.

> This is the most common attack, yet it is devastating and difficult to repel (it ranks first in the top 10 previously mentioned). We will try to focus on it in a greater detail, but if you want to know more, you have to study the resources particularly dedicated to it.

Even if the replay attack fails the information about the backend DB is gathered, such as the version, patches, platform, and possibly constraints, DB name and tables name. It is quite a good start to look for a violation of your Contract Centralization SOA pattern implementation (`http://soapatterns.org/design_patterns/contract_centralization`), look for open DB connections available at the service location, and carry on with the injections.

Interestingly, SOAP messages can be a good carrier of SQL injection attacks and an overenthusiastic Error Handler can provide perfect assistance in it. A simplified SOAP request will look like the following code:

```
<soapenv:Body>
           <pci:getCreditCard   soapenv:encodingStyle="http://schemas.
xmlsoap.org/soap/encoding/">
                          <id xsi:type="xsd:string">1 or 1=1</id>
           </pci:getCreditCard>
</soapenv:Body>
```

Yes, the same old `1 or 1=1`, exactly as in old HTML/ASP times. The `pci` namespace is pointing to the `webgoat.owasp.org` test application, which is open to such a direct approach. You might think that your application is far better protected than OWASP WebGoat application (which is in fact deliberately unsecured). We hope so, but let's not jump to conclusions right away. All we know (from the preceding code) is that the XML SOAP message can be used for the SQL injection directly. There are three classes of SQL injections: Inband, Out-of-band, and Inferential. The difference is how you get the response (if you ever get it). If you see the response immediately, that's Inband, and in the SOA world, it's the most common. What if you shut down the error handler, block all responses and return nothing, and redirect everything back to a 404 page? Can you be safe? Sorry fellow architect, the answer is no. Jumping ahead, we can say that an Inferential or Blind SQL injection technique is the hardest one, but it can still do the trick. The following is a simplified modification of the previous code example:

```
<id xsi:type="xsd:string">id=1; if not(select system_user)<>'sa'
waitfor delay '0:0:10' </id>
```

So, what we are actually asking is what privileges your service account has on the underlying resources. If it's a MS SQL system admin, please return your "unbreakable" 404 page after 10 seconds. One way or another, we will get the answer, but it will take just a little longer.

All these attacks are so common and effective that they have a lots of tools to support most of the attack types. Firstly, to find the victim (entity service, potentially with DB), public Seekda Web Service search engine or WSindex can be used. For official UDDI crawling of public services, `http://www.soapclient.com/uddisearch.html` is useful.

To probe and test the targeted service, SoapUI is amazingly good (you can try it together with WebGoat first). However, if someone wants command-line tools to fire constructed SOAP messages with injections, then SOAPClient4XG (Java), CURL, and SOAP::Lite (Perl) prove to be handy. If you need something for all occasions, Burp Suite is an obvious choice and you will learn a lot about injection attack patterns and much more. Talking about injections, specifically with proxy assistance, look at the following list:

- paros
- w3af
- sqlmap
- wpoison

We are sure that you will find more, but at the time of writing this, these were active and quite helpful. Still, if you are going to use them as a verification tool, remember that they do not cover all three types of injections and you have to do a lot of manual work to identify service weaknesses (we will not go into the basics of injection as it's not the subject of this chapter).

So an attacker gets the error message (or whatever—system silence is also a message). The injection point is identified. The next step is to identify what type of data is behind. That's simple, because your XSD will clearly say that. If not, it is not that difficult to learn after a series of reply attacks. Why is it important? Because attackers will learn, should they use a single quote for `1=1`, and the type of evasion technique will be necessary to bypass signature scanners at the Security Gateway:

```
…>id=1 having 1=1... or   …>id=X' having 1=1 …
```

An attacker wants your data. Thus, union-based constructs will be used along the way. If your service stays passive, the last resort is blind injection.

What's important here is that this technique is almost identical for REST, SOAP, or command-line (direct) attacks in terms of construction of the injection syntax. In REST, you (or the attacker) can use the following:

```
http://[victim_site]/[victim_resource_page]?id=1%20or%20
1=convert(int,(CREDITCARD_NUM))
```

If error messages are on, the system will respond with an error saying that the conversion to `int` failed, but it will give you (or the attacker) the actual column name. So, you can collect all the column names from the REST service DB (guesswork here is exceedingly easy).

If that doesn't work, you can identify the number of columns first using a union-based technique, such as `UNION`, `SELECT ALL`, `1`, `2`, `3`...`N`, where `N` is the number of columns and you have to try it `N` times (just some guess work). After that, you can start replacing the numbers with possible column names and watch the response. A lot of the previously mentioned hacking tools can do this for you.

Blind injection is the hardest because all that you can simulate usually is how quick the blank error page (404, or whatever) will be returned. So, it's basically just "Yes" or "No"; however, with some persistence, that would be enough. The `sqlmap.org` site uses blind injection, so it will help you in the hardest cases.

For an attacker (or you, if you pentest your security perimeter), life is a bit more complex than breaking into a WebGoat site and the actual REST injection query string could contain something like the following code (depending on DB and its version) as well as 200 more characters, presenting conditional branching, redirecting, and calls for recreation of the earlier deleted `xp_cmdshell` stored procedure. (Yes! It can be restored by SQL injection in the REST service query!):

```
...OPENROWSET('SQLOLEDB',";'sa';'<password>','select 1; DECLARE@
resultint,@OLEResultint, .... EXECUTE @OLEResult=sp_OACreate "WScipt.
Shell", ...... "CreateObject%0X",...
```

An attacker can change the privileges, call standard packages, or stored procedures (Oracle or MS SQL). Even using a blind injection, it is possible to extract a sys password and the current service account into the DBA group.

We will touch upon countermeasures in the following section, but while we are in the attacker's shoes, we will briefly explain how they can be dodged:

1. If the REST/SOA API is used in a web app and the user data entered is sanitized by JavaScript before the API ... no, we are not going to discuss it, it's just too easy!
2. If the first point is clear, then the defender will establish the already mentioned Service Perimeter Guard with the secure perimeter's SOA Message Screening pattern. The first move is to maintain the black list and to look at the `1=1` signatures. Where is the enforcement point? Equal sign? So, is this the numeric filtering with `"<,>,!="`, yes? How about `like` instead? What about `2=2;` or `3>1;` or `/**/2/**/=/**2`? We can go for miles with such tricks, but we will save you time and head straight to the sad truth: the signature-based IDS will catch only school kids playing with a freshly downloaded Burp Suite or automated tools configured by default. Does the restrictive list have comments? Replace them with `%2D`:

   ```
   http://[victim_site]/[victim_resource_page]/[resource_
   operation]?id=2/**/or/**2/**/like/**/2%2D%2D
   ```

3. Looking for more complex regular expressions in IDS such as `SELECT`, `UNION ALL`, `LIKE`, and so on? Go to `asciitohex.com` and read what's written further: `%55%4e%49%4f%4e%20%41%4c%4c`. Go on, add it to the REST query string with some of your trimmings.
4. Now IDS can identify Hex. Good! Does it check for UnIoNaLl letter case mutilation in Hex? And why should it always be the same encoding type? An attacker has plenty encodings to play with: UTF7 or UTF8 (here some XML developers will start getting the idea why declaring encoding in the XML message root is so important).
5. For the automation of detection evasion, go to `http://phpids.org/`. Try your own injection strings and see what patterns/IDS regular expression will be triggered for free!

The last thing to mention is that SQL injection is still the most popular, but don't count on it alone. The attacker will unleash a complex multivector attacks the injection's support. One of the possible combinations could be as follows:

- Crash restart your service (buffer overflow) in order to reset the nonce sequencing.
- Detect the IDS type and see which logs it's controlling (Apache, IIS). Then attack the HTTP servers in order to get to the logs and clean them.
- Detect internal IDS weaknesses including the black lists.

By the way, if you think that SQL injections are possible because someone is concatenating the SQL string before executing the query, be aware that the SQL procedure injection is quite possible as well. Prepared statements are more resilient though.

The conclusion is that the attacker has an advantage, not you. Your task is to prepare a multilayered defense.

XPATH injection

The attack code for XPATH injection is `AT05`.

The underlying idea behind XPath (and XQuery) is to see XML as a kind of systematic storage (similar DB) and have similar ways of accessing data.

So, as in SQL, user supplied information can be used for the construction of the XPath query string. The hacking methods are almost identical to those previously explained.

The standard path to employees' XML data, `/employees/employee[@id='EMPLOYEE_ID']`, can be easily supplied with the classic `'%20or%20'1'='1` and all the employee data (including CEO) will be dumped to the browser. You can add any XPath-related function you like, such as:

```
'%20or%20fn:contains(fn:lower-case(@lastname),'your_CEO_lastname')%20
or%20'
```

The counter-countermeasures are similar to an SQL injection. The most hard to break are the prepared statements for XPath expressions.

JSON injection/JavaScript injections

The attack code for JSON and JavaScript injections is `AT06`.

Imagine your service accepts the JSON documents (or constructs them based on user input) and stores them for public use. For big and bulky JSONs, you would dedicate a NoSQL database, hoping that NoSQL means NoSQL injections.

JSON itself is brilliant, as it's simple, and it can be injected (in a positive way) in any software/site capable of handling JavaScript. The flexibility is immense, thanks to AJAX and shared DOM. We can manipulate web page elements almost effortlessly using, for instance, the Chrome extension API (`chrome.extension.getURL()`). JSON content will be included directly into the `<script>` tag of the targeted page. You will find all the necessary instructions on the jQuery site, including `manifest.json` and a sample of the `<injected.js>` files.

Fellow architects, the preceding paragraphs are from the Chrome Extension API and jQuery documentation (`https://developer.chrome.com/extensions/extension#method-getURL and http://jquery.com/`), that's not a joke. You will even find the following examples:

```
document.head.appendChild(script);
document.body.setAttribute("onLoad", "injected_main();");
```

At first glance, nothing is wrong, except the potential JSON parser/deserializer's strength. Also, we know that the standard `eval()` function, used for conversion JSON into a JS object, is full of holes. The `eval()` function in many cases (five conditions that can prevent this from happening, some of which are based on filtering) can execute the embedded JavaScript code. Also JSON's array vulnerability has been exploited many times. The most well-known victims are Gmail and Twitter.

> JSON is no less secure than XML. It's all about how you use data (encryption, digital signature, screening, validating against standards, use of reliable parsers instead of `eval()`). The preceding example with the injecting of potentially insecure JSON (which also could be the subject of injection) is something you must strongly reconsider.

Schema poisoning

The attack code for schema poisoning is `AT07`.

Poisoning, that is, injecting malicious code into a single document, XML or JSON, is bad enough, but the corruption of your entire service data blueprint is a disaster, one that attackers will try to hide from you as long as they can, sneaking in and out unchallenged. It could be the result of a long-planned attack, when an attacker watched your dev repository site, got into it, modified schema or its include elements, and then waited for the opportunity in production.

A bit complex isn't it? Most common for the complex schemas is having different XSDs at different locations, sometimes public for common QDT/namespaces; an attacker can get into the weakest location and change the element type of the XSD instruction. Sometimes, just changing the level of data granularity could be enough—an unlimited string in a key element can open enough room for injections.

Another way of poisoning is having entire schema as `<any>`. Yes, we used this type of declaration for the demonstration of the universal message container for the agnostic controller, but we did it behind the perimeter protection. We all know many SOAP-like endpoints that accept `<any message>` because there are so many message/schemas, types of data, and vendor's versions, so it seems to be easier to parse messages on the backend without initial validation. So many opportunities for the attacker!

Forced browsing

The attack code for forced browsing is `AT08`.

As described by OWASP, forced browsing is an attack where the aim is to enumerate and access resources that are not referenced by the application but are still accessible.

This is quite a nasty thing and really dangerous. You have SOAP/WSDL, or more commonly, the REST service infrastructure with plenty of handy services available. It's so easy to build the REST service, as demonstrated earlier. When you start, it's almost impossible to stop. You have them for the internal purposes of accounting/finance, delivery and warehouse, procurement and provisioning, and those `OrderRequest` and `getInvoice` services that you decided to make public.
Or do you just think only two of these are public?

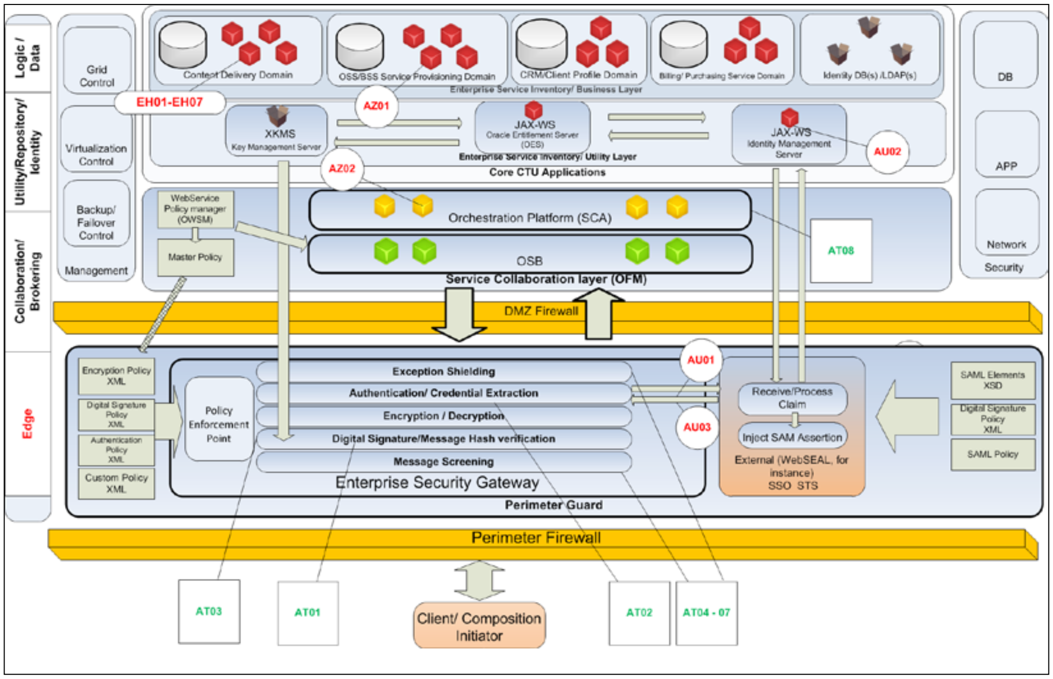
Some of the scanners mentioned earlier are capable of traversing the victims' server in order to find those that are not declared but still exposed resources with a much less secure model. Brute force guessing could work as well.

Risk mitigation design rules

We didn't set a goal of covering all possible vulnerabilities and attacks. You can study them at OWASP and other resources, but the ones mentioned are sufficient to devise your battle plan and gather all the critical requirements for your SOA protection. Clearly, there is no single tool that can help us a 100 percent, but from from the risk table, you can gather that Service Perimeter Guard is the top pattern to address most of the SOA-related risks. Together with proper service design, an identity management system, security token services, and Policy Studio, this pattern, materialized as Secure Gateway, will be our first line of defense.

Using code for vulnerabilities and attack types from the from the previously discussed classification we will compose our security battle map—a security-related heat map in a form of SOA components and technical infrastructure blueprint, where we link the existing service domains with predefined codes. The goal is to identify critical nodes in our infrastructure and assess the feasibility of the core SOA patterns application. We will use the CTU telecom example from the previous chapters, but its SOA infrastructure is quite common to any enterprise. The level of details on this block diagram will ultimately define the precision of the security protection assessment. So it's in your own interest to make it as complete and comprehensive as possible. What is presented in the next figure is just a starting point and you should expand it for every layer with a clear definition of overlapping sections (areas that an attacker will try to address first).

We already placed Oracle strategic products in appropriate places, including Service Gateway, Entitlement, and Identity Management Suite. Bear in mind that most vulnerabilities are inherited from poor service design and this fact is quite hard to visualize on the heat map; you should refer to peer review reports for this.


Logic / Data
Utility/Repository/ Identity
Collaboration/ Brokering
Edge
Grid Control
Virtualization Control
Backup/ Failover Control
Management
Content Delivery Domain
OSS/BSS Service Provisioning Domain
CRM/Client Profile Domain
Billing/ Purchasing Service Domain
Identity DB(s) /LDAP(s)
Enterprise Service Inventory/ Business Layer
EH01-EH07
AZ01
AZ02
XKMS
Key Management Server
JAX-WS
Oracle Entitlement Server (OES)
Enterprise Service Inventory/ Utility Layer
JAX-WS
Identity Management Server
AU02
Core CTU Applications
DB
APP
Network
Security
WebService Policy manager (OWSM)
Master Policy
Orchestration Platform (SCA)
OSB
Service Collaboration layer (OFM)
AT08
DMZ Firewall
AU01
AU03
Encryption Policy XML
Digital Signature Policy XML
Authentication Policy XML
Custom Policy XML
Policy Enforcement Point
Exception Shielding
Authentication/ Credential Extraction
Encryption / Decryption
Digital Signature/Message Hash verification
Message Screening
Enterprise Security Gateway
Receive/Process Claim
Inject SAM Assertion
External (WebSEAL, for instance) SSO STS
SAML Elements XSD
Digital Signature Policy XML
SAML Policy
Perimeter Guard
Perimeter Firewall
AT03
AT01
Client/ Composition Initiator
AT02
AT04 - 07

SOA technical infrastructure from security perspective

Identity management – defending credentials verification systems

Gartner defines Identity Management as follows: "Identity management is the set of business processes, and a supporting infrastructure for the creation, maintenance, and use of digital identities." Direct or Brokered identification services are the most critical resources in our service inventory, not only because they hold clients and corporate sensitive information, but also because of their highest level of reuse.

The most common rules around identity management protection are as follows:

- Use a zero-knowledge password protocol (ZKPP) such as SRP.

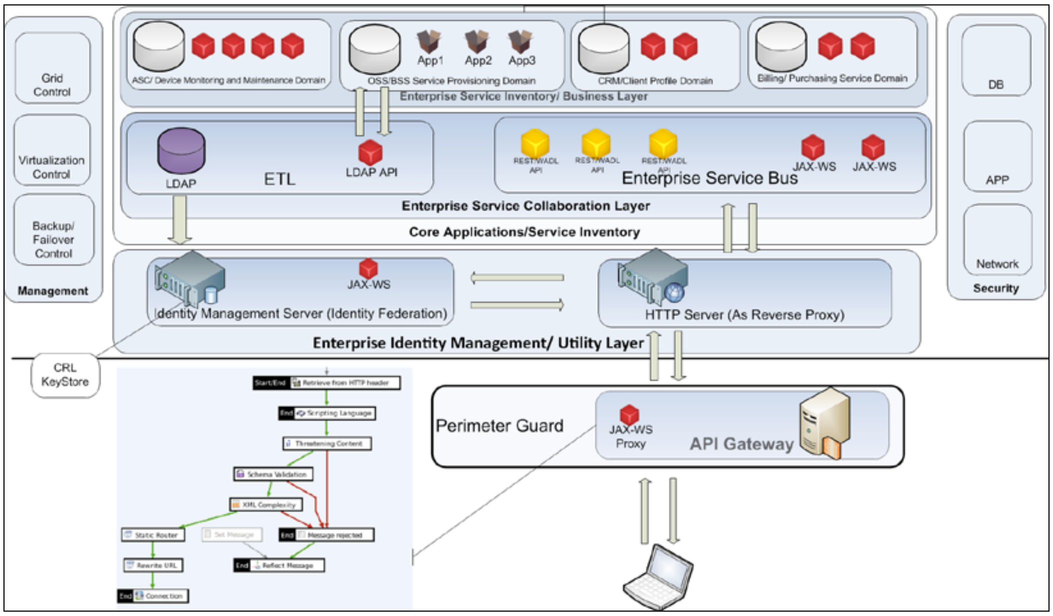
- Passwords should be stored safely to prevent insider attacks and to ensure that if a system is compromised, the passwords are not retrievable. Due to the reuse of a password, this information might be useful in the compromise of other systems these users or services work with. In order to protect these passwords, they should be stored in an encrypted way, in a nonreversible state, so that the original text password cannot be extracted from the stored value.
- Password aging should be strictly enforced to ensure that passwords do not remain unchanged for long periods of time. The longer a password remains in use, the higher the probability that it has been compromised. For this reason, passwords should require periodic refreshing, and users should be informed of the risk of passwords that remain in use for too long.

The common tasks around identity management can be described as gathering and storing all credential information in a secure way (passwords as hash, storage media encrypted, and DB row-level security). We have to remember that identity data is quite often scattered around several applications in different business domains, so MDM could be the essential part in Identity Data Maintenance and protection. Once collected and synchronized, credentials will be exposed through a secure API to a Security Token Service, which will provide/renew/revoke tokens, and to the authentication services (usually in scope of Perimeter Guard) and making authentication decisions based on the information in the incoming message. Following the separation of the concerns principle, ID storage (ID management), ID validation (access management), and ID extraction/injection (on SG) are three different components of the AAA security subsystem.

As you can see in the following figure, the starting point is always Perimeter Guard; its main purposes will be to scan the message before AA's operations and isolate IDM systems from the client. The additional level of isolation can provide the reverse proxy pattern, shielding internal Web/REST resources from direct calls.

IDM modularity opens the possibility for the parallel implementation of Direct and Brokered Authentication. The first one is simple and described by the name itself. Brokered Authentication (the synonym is SSO) requires an Authentication Broker, usually to act as a Security Token Provider, which returns the entry validation ticket (token) to the service-requestor for a predefined amount of time with a list of the allowed resources. As this operation requires redirection (see OWASP risks), SG shall be involved in scanning redirects, signing the tokens, and minimizing redirects (the initial client request can be immediately redirected to STS without it reflecting back to the client).

From the following figure, we see that Oracle Identity suite and API Gateway can be combined with other products such as Tivoli Federated Identity Manager (as STS) and WebSEAL reverse proxy.



SOA technical infrastructure from identity management perspective

Authorization data can be stored separately, but controlled by the same Oracle Identity Manager.

The list of Oracle Identity Management and Access Management products at the time of writing this are as follows (some of these could be rebranded, renamed, or merged, so check the Oracle site regularly).

The following are the Identity Governance products:

- **Oracle Identity Manager (OIM)**: This is an identity provisioning product. OIM includes features for self-service password management, access request forms, delegated administration, approval routing workflows, and entitlement management across any number of connected systems.
- **Oracle Identity Analytics (OIA)**: This collects logs from IDM products and other systems to report on usage, builds effective IT roles, and detects account-related audit issues such as orphaned accounts.

- **Oracle Privileged Account Manager (OPAM)**: This secures accounts with elevated access, such as root accounts on Unix systems and databases, by implementing a password checkout system.

The following are the Access Management products:

- **Oracle Access Manager (OAM)**: This is a **Web Access Management (WAM)** product that enables SSO across an organization's web presence.
- **Oracle Adaptive Access Manager (OAAM)**: This enables organizations to apply stronger, risk-based, and multifactor access control to an organization's web presence.
- **Oracle Identity Federation (OIF)**: This provides standards-based identity federation capabilities to enable SSO across websites.
- **Oracle Security Token Service (OSTS**): This is a WS-Trust compliant STS implementation. An STS converts security tokens of various types, enabling compatibility and trust across federation boundaries.
- **Oracle Entitlements Server (OES)**: This is a fine-grained entitlements service that supports various externalized authorization mechanisms including XACML 3.0.
- **Oracle Enterprise Single Sign-On (OeSSO)**: This is a client-based SSO product that enables users to access web, client-server, and legacy applications through a single, strong authentication wallet for authentication.

Directory services products

Indisputably, Oracle is one of the leaders in directory product offerings (LDAP directories). **The Oracle Internet Directory (OID)** was the first product in this group and now, we have a highly efficient **Oracle Unified Directory (OUD)**, which includes both a highly scalable LDAP directory service based on Java and a **Oracle Virtual Directory (OVD)** product. OUD comes with the following three main components:

- Directory Server
- Proxy Server
- Replication Server

The Directory Server essentially is a highly scalable and top-performing LDAP. The Proxy Server contributes to LDAP's high-performance proxy requests and responses and the Replication Server is responsible for the data replication from one OUD to another.

This list of products is just an indication that Oracle has everything necessary for the proper implementation of all eight SOA security patterns. Now it is your responsibility to enforce the security of your services by addressing error handling vulnerabilities (these are what makes your services leak and opens the door for injection-type attacks).

Exception shielding – preventing an information leakage

We have put together short responses to problems in Error Handling (EH[nn]) identified during error handling vulnerability analysis.

EH05

As an option, `javac` will warn you about the return in the `finally` block if a compiler's argument is set as `-Xlint:finally`. Use of the Ant `<compilerarg>` element as follows:

```
<javacsrcdir="${src.dir}" destdir="${classes.dir}"
classpathref="libraries">
<compilerarg value="-Xlint"/>
</javac>
```

EH06

Correct cleaning is always important, but you should be extra careful with threads. Please look at the following code samples. They can help you to mitigate at least two types of attacks, bases of buffer overflow and information leakages:

```
private static final long SLEEP_INTERVAL = 100;
private static void removeGarbage() {
 try {
     System.gc();
    //give a thread chance if you can
    Thread.sleep(SLEEP_INTERVAL);
     System.runFinalization();
  }
  catch (InterruptedExceptionie){
    //handle threads properly, log exception clearly,
     //DO NOT JUST print stack trace !
     // Try to exit neatly, do not just kill it by  .stop()  method
      Thread.interrupt();
    }
//Other errors
   catch (Exception ix){
        //same as above, DO NOT JUST print stack trace !
        // use Runtime.getRuntime().totalMemory(), maxMemory() and
freeMemory()
        // in logging procedure for recording the JVM memory state at
the moment of
        //error.  Use simple equation  usedJVMMemory = totalMemory() -
        // freeMemory()  for  calculating amount of used memory.
    }
}
```

> Do not force garbage cleaning by calling `System.gc()` too often to release the memory. You should trust the intelligence of modern JVM (Oracle JRockit in particular, the true core of OFM) and rest assured that JVM does everything possible to optimize memory utilization. If your code is far from optimal, and contains any vulnerabilities mentioned in vulnerability analysis at the beginning of this chapter, even the best garbage cleaner from JRockit won't be able to turn the tide. Furthermore, even the paramount perimeter protection around your Service Inventory will just die trying to defend service compositions.

Although samples in this paragraph are Java-related (strategic Oracle language), make sure that you have the proper PL/SQL exception handlers as well. All data handling code should be on PL/SQL. (This is for a relational DB, of course; for NoSQL, it depends on realization but is close to data.) Use PL/SQL packages for better modularity and EH centralization. The statement in packages must be prepared. That's it. These are the most effective measures to make injection attacks as hard as possible. Oracle provides complete guidance on how to write injection-proof PL/SQL code. You can find the documentation at `http://www.oracle.com/technetwork/database/features/plsql/overview/how-to-write-injection-proof-plsql-1-129572.pdf`.

Message screening – preventing injection attacks

While discussing the attack types, we spent most of the time talking about injections, because this is the best way to get to your precious data. That's what the attackers want, not just to crash your system. Some say that **DoS** (`http://www.cert.org/historical/tech_tips/denial_of_service.cfm`) repelling is the hardest security task. No, it isn't. It's just most expensive, but not the hardest. Injections are much more tricky. Why? Because most of the protection techniques are based on scanning input signature and we demonstrated (very briefly though) that IDS methods can be disabled and bypassed. The deceptive complexity of data abstraction levels should not hoodwink you—most serializers/marshallers are designed to transport data from the XML to the SQL column without changes (that's their sole purpose) and security is not their responsibility.

Does this mean that message screening is futile? No, even 50 percent positive catches is a positive thing, and we really could reach higher numbers. Just remember, defense is complex. How? Secure Gateway (Oracle and most of others) is an ESB by design. So, we basically have two ESBs and you can add an additional XSD check on OSB as well. Make sure that the only option your attackers have is a blind injection. Let's make their life harder. What else can help us? The following are the most effective measures for consideration:

- As already mentioned, use prepared statements. But remember, all data in the statement binding must be parameters. If in addition to that, you construct your statement as a concatenation of a string, you're just invite trouble.
- Oracle SG (inherited from Vordel) uses nonstandard XSD parsers. Surely, nonstandardization is not a guarantee of protection, but at least, it will give attackers a hard time. By the way, the same algorithm is used in Intel ESG, which is also quite secure, so it's double-checked.
- Simple and restricted inputs are easier to scan. That is, the input of 10 digits is much easier to protect than a car's VIN. Yes, that's not always the case, but you can work on Canonical Data in order to minimize the impact on security.
- Generic Adapter from *Chapter 6, Finding the Compromise – the Adapter Framework*, which is capable of dynamically executing any SQL statement, should be considered insecure and you should never contemplate using it for externally exposed compositions. Actually, we mentioned this when we discussed its design and clearly stated that, ideally, every adapter will be individually tailored to the wrapped application. Generic Adapter (Adapter Factory) is a pattern aimed at the reduction of similar adapters in front of ESB. Still, it can be secured—a statement can be prepared (and presented as a procedure call), and input parameters can be thoroughly sanitized (that's how it works in CTU—lengths and types are strongly restricted). Still, the main rule is to use it only for internal services and hide all composition behind Secure Perimeter.

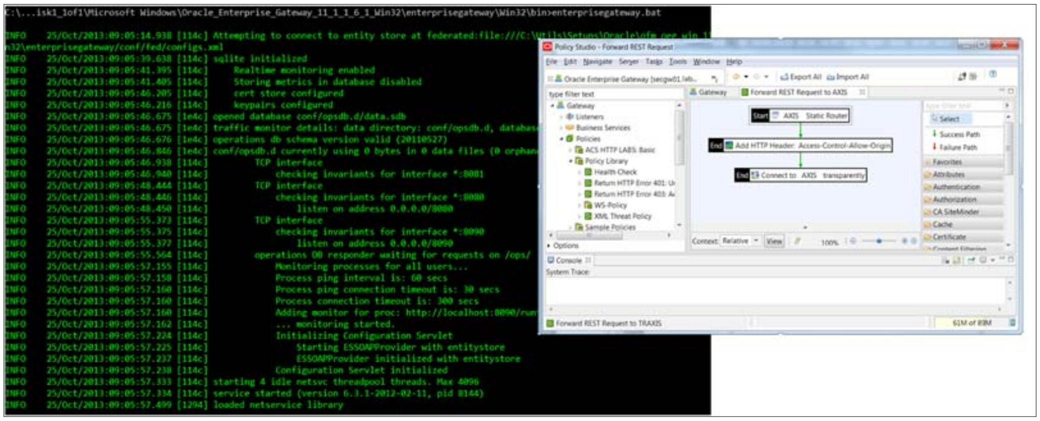
> Combined together, the preceding measures can give quite a level of protection. Amazingly, there are still a lot of developers who produce tons of JAVA code such as `Statement stmt = conn.createStatement("INSERT INTO customer VALUES('" + user + "')"); stmt.execute();`. Thus, Oracle API Gateway must be in your arsenal.

Oracle Enterprise (API) Gateway

From the diagram in the *Risk mitigation design rules* section, you can see that OEG is the product with highest concentration of core SOA security patterns. What are the common requirements for such a tool to be trusted?

Vendor-neutral (generic) requirements

Some information about the requirements can be found on OWASP (search for `OWASP XML Security Gateway Evaluation Criteria Project`), but a list of twenty or forty criteria is too small. Our actual list has almost 200 positions and comprises technical requirements of three different commercial SGs. Oracle is one of them. We encourage you to read some technical whitepapers about its capabilities as it is beyond the scope of this book.



```
C:\...isk1_1of1\Microsoft Windows\Oracle_Enterprise_Gateway_11_1_1_6_1_Win32\enterprisegateway\Win32\bin>enterprisegateway.bat

INFO    25/Oct/2013:09:05:14.938 [114c] Attempting to connect to entity store at federated:file:///C:\Utils\Setups\Oracle\ofm_oeg_win_1
n32\enterprisegateway/conf/fed/configs.xml
INFO    25/Oct/2013:09:05:39.638 [114c] sqlite initialized
INFO    25/Oct/2013:09:05:41.395 [114c]     Realtime monitoring enabled
INFO    25/Oct/2013:09:05:41.405 [114c]     Storing metrics in database disabled
INFO    25/Oct/2013:09:05:46.205 [114c]     cert store configured
INFO    25/Oct/2013:09:05:46.216 [114c]     keypairs configured
INFO    25/Oct/2013:09:05:46.675 [1e4c] opened database conf/opsdb.d/data.sdb
INFO    25/Oct/2013:09:05:46.675 [1e4c] traffic monitor details: data directory: conf/opsdb.d, databas
INFO    25/Oct/2013:09:05:46.676 [1e4c] operations db schema version valid (20110527)
INFO    25/Oct/2013:09:05:46.846 [1e4c] conf/opsdb.d currently using 0 bytes in 0 data files (0 orphan
INFO    25/Oct/2013:09:05:46.938 [114c]         TCP interface
INFO    25/Oct/2013:09:05:46.940 [114c]             checking invariants for interface *:8081
INFO    25/Oct/2013:09:05:48.444 [114c]         TCP interface
INFO    25/Oct/2013:09:05:48.446 [114c]             checking invariants for interface *:8080
INFO    25/Oct/2013:09:05:48.450 [114c]                 listen on address 0.0.0.0/8080
INFO    25/Oct/2013:09:05:55.373 [114c]         TCP interface
INFO    25/Oct/2013:09:05:55.375 [114c]             checking invariants for interface *:8090
INFO    25/Oct/2013:09:05:55.377 [114c]                 listen on address 0.0.0.0/8090
INFO    25/Oct/2013:09:05:55.564 [114c]         operations DB responder waiting for requests on /ops/
INFO    25/Oct/2013:09:05:57.155 [114c]             Monitoring processes for all users...
INFO    25/Oct/2013:09:05:57.158 [114c]             Process ping interval is: 60 secs
INFO    25/Oct/2013:09:05:57.160 [114c]             Process ping connection timeout is: 30 secs
INFO    25/Oct/2013:09:05:57.160 [114c]             Process connection timeout is: 300 secs
INFO    25/Oct/2013:09:05:57.160 [114c]             Adding monitor for proc: http://localhost:8090/run
INFO    25/Oct/2013:09:05:57.162 [114c]             ... monitoring started.
INFO    25/Oct/2013:09:05:57.224 [114c]             Initializing Configuration Servlet
INFO    25/Oct/2013:09:05:57.225 [114c]                 Starting ESSOAPProvider with entitystore
INFO    25/Oct/2013:09:05:57.237 [114c]                 ESSOAPProvider initialized with entitystore
INFO    25/Oct/2013:09:05:57.238 [114c]             Configuration Servlet initialized
INFO    25/Oct/2013:09:05:57.333 [114c] starting 4 idle netsvc threadpool threads. Max 4096
INFO    25/Oct/2013:09:05:57.334 [114c] service started (version 6.3.1-2012-02-11, pid 8144)
INFO    25/Oct/2013:09:05:57.499 [1294] loaded netservice library
```

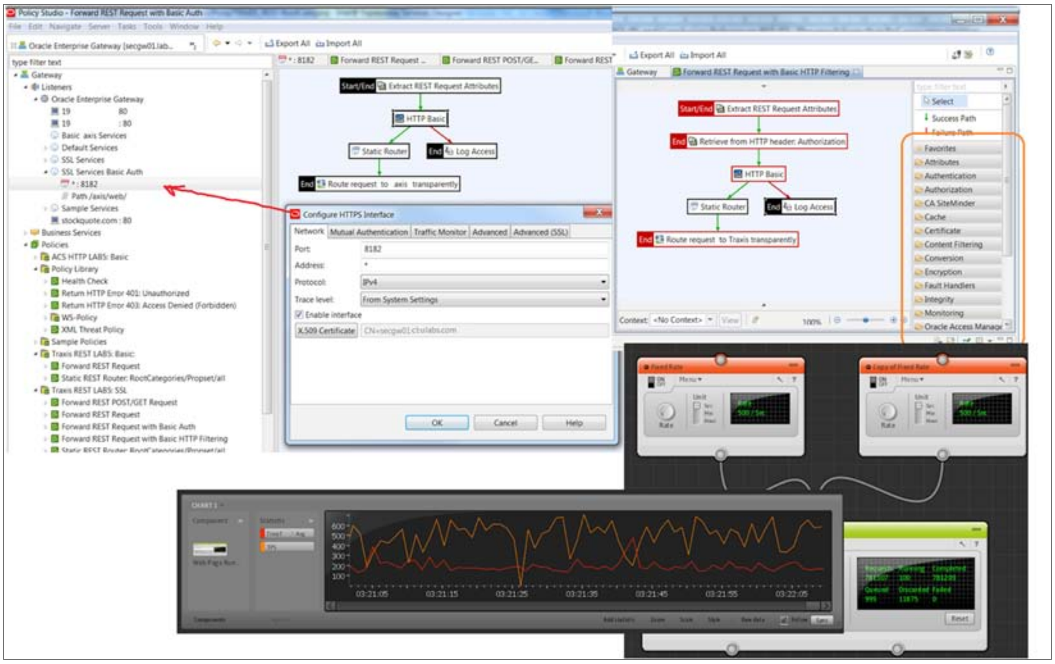
Pass-through proxy with HTTP header verification

Number 5 in the OWASP top 10 is directly related to the complexity of security tools and mechanisms. In our experience, OEG proved to be amazingly simple in installation, maintenance, and development (see the previous screenshot). Right after unpacking, you will get a resilient and simple environment and an easy to start/stop and monitor. Policy Studio (main development tool) is not exactly Eclipse-style, so it will take time some time to adapt after OSB with its traditional request-response pipelines; however, in an hour, you will be able to build reasonably complex flows for SOAP/REST services with an HTTP or SOAP attribute verification and validation and so on. Development is policy-based, that is, you can define the message flows and different policies for every step separately, maintain nested policies (implementing a Policy Centralization SOA pattern), and apply a common policy to the different flow elements.

Using drag-and-drop development, you can assign a scan for inbound messages, connect to different identity providers, extract or inject SAML tokens, and protect them from spoofing/alteration (see the message processing flow in the next screenshot). The full set of OEG development categories is seen on the right-hand side of the next figure, in the orange box . It is a truly complete set of functions, essential for perimeter protection, inbound/outbound message screening, authentication, authorization, and runtime audit.

Performance requirements

One of the critical requirements for a business is obviously concerned with incongruity between security and performance. A simple dummy REST proxy service (not a real case!), assembled as shown in the following screenshot, was even unable to stress dual-core 8GB RAM VM, handling 500 REST transactions per second (response 3K JSON). Actually, LoadUI from a single machine was unable to produce enough stress.



REST service stress test setup

> The preceding example demonstrates how easily the REST service flow can be assembled. For your own protection, please forget that Basic HTTP authentication exists. Using HTTP Header for authorization is not bad though, as it's encrypted and signed.

For real OEG performance capabilities, please see figures from our real perftest report.

We have several tests that run the 50 KB message that contain the following aspects:

- Signing of the `<EmployeeID>..</EmployeeID >` element
- Encryption of the `<Salary>..</Salary >` element
- Signing of the SOAP body
- Using HTTPS between test tool and Oracle Enterprise Gateway

The main test was executed from a terminal session running on the Linux server:

```
use HTTPS sessions
parallel (threads): 64
test duration: 3600 (somehow the actual test run for a bit more then 2 hours)
time taken: 7785.540000 secs.
bytes sent: 22362.978370MB (23449282407 octets)
bytes received: 24503.956365MB (25694260549 octets)
transactions: 445609
connections: 445629
sslConnections: 445629
sslSessionsReused: 0
bytes sent/sec: 2.872373MB (3011901.859987 octets)
bytes received/sec: 3.147368MB (3300254.131248 octets)
transactions/sec: 57.235465
```

As a conclusion from these figures, you can deduce that having 10K TPS on a single VM with two cores and an 8 GB RAM for a 50K SOAP message, completely protected by TLS and MLS, is an achievable target.

Summary

Following the book's paradigm, "identify the problem first and propose the adequate solution (pattern)", we spent a lot of time discussing security threats, risks, and attack types. We only regret that, in this format, we cannot elaborate more on hackers' techniques. Nevertheless, forewarned is forearmed. You are now equipped with some of the most common and dangerous tricks. You have a list of essential Oracle tools and fundamentally, you know how to combine them in defensive and preventive patterns.

Practically, any attacks start from "reconnaissance missions" to obtain the critical information from regular or fault responses, error logs, dumps, and so on. Therefore, the next chapter will be dedicated to fault handling frameworks and will logically continue the security frameworks requirements in regard to error catching, shielding, prevention, and recovery.

8
Taking Care – Error Handling

While discussing security patterns, we described handling (or, in fact, mishandling) faults as one of the major contributors to the vulnerabilities of SOA, and the inadequately designed Fault/Errors Handling (EH) framework is apparently the main provider of all information to the error/event logs and the "grateful" attacker. We mentioned some simple rules for exception handling inside a single service (Entity or Utility service models) as the sole building block of the entire SOA infrastructure. Implementation of this rule would be enough for an infrastructure that contains only these models of SOAP services or simple REST services without compositions of any complexity. When we have something more complex (such as Task Orchestrated service models) or in fact any external exposure along with associated security risks or (usually) both, something that is a lot more substantial is required. This *something* in addition to a proper EH's service design will require events logging and log mining/analyzing; it also requires you to add compensation handlers, build compensation policies, bind policies and handlers, and establish manual recovery routines in a worklist as the last line of defense. More often than not, the complexity of the steps we just mentioned is frustratingly high (composition with three sequential invocations can easily span compensation activities with five invocation steps). It is so complex that after several workshops and incomplete prototypes, architects decide to put all of the handling into a work list to come up with a manual resolution with all the associated human-related problems (responsiveness, accuracy, and consistency).

The design of our agnostic composition controller will be incomplete and the whole idea of dynamic composition assembly compromised if we do not demonstrate how to automate error recovery using SOA patterns. Traditional Oracle OFM/SCA realization covers both Rollback and Compensation patterns (where Rollback is part of Atomic Transaction Coordination and Compensation is BPEL/SCA); the automated recovery functionality is usually consolidated in the policy-based Error Hospital OFM facility.

It would be useful to look at the term *policy-based* and understand what the policy is, how it can be enforced, how many of them there are, and is it really possible to centralize all policies in one center. The role of the Service Repository, whose taxonomy we discussed earlier, has to be observed from one more side.

In this chapter, we are going to discuss standard tools first, explaining what kind of centralizations you have to maintain to achieve Policy Centralization (for recovery, compensation, and composition protection) as well as basic patterns such as Compensative Service Transaction, Service Repository, and Service Instance Routing. However, the main purpose here is to present **Automated Recovery Tool (ART)**, which is capable of automating service recovery and transaction compensation.

Associating SOA patterns with OFM standard tools

Compensation is not exactly proportional to the complexity of your composition. With no rollback option available, returning to a condition that is equivalent to the composition's initial state (before the composition was initiated, or at a certain composition condition where the state of affairs was consistent) is directly related to the duration of the composition. Think of it this way: if you break your wife's favorite cup, a bunch of flowers and another cup would be enough. However, if you forget your wedding anniversary, even a good diamond ring could barely settle this down (although, one positive thing is for certain: you will never forget it again). Talking seriously about services' data consistency, the further we are from the composition's initiation point, the more difficult it will be to remove all relations that occur around the data implanted by service activities. As we mentioned, long running compositions hosted by SCA in general can be presented as a chain of ATCs that are handled by OSB. Actually, this is the shortest description of the agnostic composition controller, which is available in *Chapter 3*, *Building the Core – Enterprise Business Flows*, to *Chapter 6*, *Finding the Compromise – the Adapter Framework*; it signifies that the centralized exception handling facility should equally cover Rollback and Compensations.

> Regardless of the terminology you have established for your domain, the process of handling errors, exceptions, and system messages, in general, is the same as that of detecting the event, assigning an appropriate action for it, and executing this action in a controlled manner. In the scope of this chapter, the event we understand is not just any event (change of state), but those that are not in the range of our "happy execution plan". A forced event such as `Audit` is something different, although its data can be stored together with exceptions in the common log.

As we mentioned, a number of corrective actions can be quite substantial, and going further, not all of them can be really corrective. There are some preparations and post-completion steps required; also, the service composition should adhere to certain design rules in order to be more controllable in situations when recovery is needed. We will start with gathering common requirements first and then see which of the Oracle tools can cover them in a generic and reusable way. Our first traditional move is the analysis of a typical SOA landscape.

Initial analysis

Firstly, we have to bear in mind that for one single composition, we have six runtime-related SOA frameworks involved from end to end. Therefore, the occurrence of an event requiring extra care must be delivered to the composition controller that is currently involved in coordinating services' collaboration. This delivery process is conditional and depends on the composition phase (initiation, registering participants, requesting participants, invocation, accepting votes, rejecting votes, data assembly/transformation, and final delivery). Thus, an events' origins (the producer that the so-called event produces) and its time of occurrence will define how it should be handled/recorded.

At least four different Oracle products make service interactions possible: OSB, SCA/SOA Suite, ESG, and OER. This is the bare minimum, so the list of products could be much broader. These service engines, buses and orchestrators, as separate products, have their own error-handling facilities. That's quite understandable for separating API gateway error information from the service domain behind it, but it doesn't help with harmonization and unification of exception handling. The master composition controller will get accurate and consistent information about an exception in order to take proper corrective actions. As we could have at least two types of controllers' realizations (demonstrated earlier), handlers' collaboration is not a simple task.

Remember that events can be basic or complex. From the Exception Handler's standpoint, events can be of two types:

- The first type is where there is immediate reaction of an abnormal situation on service, service engine, an element of service infrastructure, or a remote endpoint, provided as an error message (for instance, a SOAP error message for WS-services, or error stack trace for an internal service resource)
- The second type of event talks about the result of statistics gathering and/or patterns' analysis of basic events that is obtained during a certain amount of time (with relation to a certain threshold of time)

By considering all of this together with the previous point, we realize the importance of consolidation of all Error Handlers under one controller; going further, we'll talk about Log Centralization as the source for **complex event processing (CEP)**. Oracle has a number of instruments for data analysis and monitoring; BAM is one of them and is sometimes considered a service transaction monitoring tool. However, some limitations make it unsuitable for runtime composition error handling. BAM is the topic of the next chapter; there, we will talk about it a bit more. Right now, we just need to mention that lack of runtime SOA capabilities makes architects and developers look at log mining and infrastructure monitoring tools such as Nagios (`http://www.nagios.org/`) closely. In your service infrastructure, you could have your own homemade monitoring/logging tools, and quite often, developers have their own opinions about the usage of log4j and its structure (scattered everywhere). Thus, Log Centralization is one of the primary prerequisites of a successful EH implementation, although, it's not considered a standard SOA pattern (it's just good common sense). By the way, Oracle is putting in some efforts in this direction by presenting different management packs for products' OEMs. Alerts from OEMs can be very useful as an input for EH.

You will also need to consolidate all logs—when we say all, we mean *ALL*. Data misses from hidden or stray logs, growing uncontrollably, are not only a perfect recipe for service resources' depletion and system crashes, but also a juicy target for attackers—and integration of all Error Handlers are the two main requirements for establishing Policy Centralization. Again, here we are talking about fault policies mostly, setting aside all other policies. This task is harder than the previous type of centralization for several reasons:

- Every tool has its own policy representation and ways of binding it to the event resolution, and different components within SCA have different policies: Mediator, BPL, and HumanTask.
- The WS-Policy specification along with related WS-SecurityPolicy and WS-PolicyAttachment are not related to most internal policies in OFM. They share some common principles, but do not expect them to be compatible or easily transformable. OEG (API Gateway) Policy Studio will not cover all diversities of policies despite its name, and it would be better to not use it for a purpose that is different from security.

Obviously, the complexity of Policy Centralization as a task is proportional to the complexity of underlying individual policies and their alternatives for every Policy Subject within its scope. These terms will be explained shortly (as you will need them for SOA exams), but stating it plainly, centralization of EH policies will be positive only if you anticipate all error scenarios for your compositions and present them in a clear hierarchy. Otherwise, it will be the centralization of *false positives*, thereby mishandling most of the error situations.

Individual service exception handlers and resource exception handlers could be a good start for building this hierarchy. The process of speculation on possible compositions in which this service might participate in would be the second step, and results of this process should be properly correlated with established Service Layers (also SOA patterns, which are discussed in *Chapter 1*, *SOA Ecosystem – Interconnected Principles, Patterns, and Frameworks*), vertically and horizontally. Simply put, this kind of anticipation and countermeasure planning must be commenced right from the service design stage, even before the first line of code is written. Your first WSDL is a good basis for this work. However, what if we cannot anticipate all the possible error situations? Well, manual recovery using SOA Suite Human Workflow is still acceptable for cases that cannot be identified, but the number of these cases must be kept to a minimum. This will ensure that manual recovery operations will be performed in minutes.

For instance, in the article *Protecting IDPs from Malformed SAML Requests* (Steffo Webber, Oracle), the discussion of OEG policies for securing SAML tokens clearly state that for mitigating SSL flaws, the manual reaction of the vulnerabilities of an SAML token on these threats is acceptable as long as the policy allows you to catch the event and send the alert.

We also have to take into account that any centralization will require storage for policy assertions, and as long as we deal with different policy formats, these assertions have to be expressed in a form that is suitable for the following:

- Transformation and quantification
- Should be understood by humans (ops, rule designers, and business analysts), with possible alterations only by an authorized personnel

Logically, Service Repository would be our first choice with the Service Repository endpoint available.

While talking about storages and logs, we have to make one distinction between Audit and Exceptions. This difference is clear in service components' design where we handle errors in the `catch{}` block, and Audit any data using `log.info()` or any other command/library you want. At the OFM tools' level, it's also obvious; for instance, OSB has log activity. The situation is not always clear when we are dealing with orchestration engines where dehydration storage can also be the source of Audit; for custom packages, it can be a common logic. In general, the level of Audit and error reporting is not the same, especially in Identity Management and Perimeter Protection. Also, regarding the Oracle Fusion Audit Framework, you should be aware that applications will not stop operating if the Audit is malfunctioning. Standalone applications can be included into the OFM Audit Framework through the configuration of the `jps-config.xml` file.

Now, we will consolidate the results of our analysis into common but detailed requirements, which are suitable for extending Oracle's Exception Handling facilities.

Common requirements

Let's get back to *Chapter 1*, *SOA Ecosystem – Interconnected Principles, Patterns, and Frameworks*, and look at the figure under the SOA Technology Concept one more time. In the first chapter, we tried to consolidate `WS-*` specifications in one logical roadmap. Generally speaking, we do not have standards that are specially dedicated to Error Handling, and activities in this area are governed by policies (as we mentioned, not exactly WS-Policies and differently for all vendors); this is due to the importance of the subject we put on the right-hand side of SOA patterns (generic) under the roof of the consolidated Error Hospital (which is Oracle-specific). So, we must have and maintain the following:

- The ultimate importance of the Exception Shielding pattern for SOA security has been explained in the previous chapter. There, we stated that doing shielding and error message sanitation on the gateway is not a good choice because unclean messages will travel across the whole infrastructure until the front door before it is cleansed/blocked. However, what if you do not have a Gateway? Even if you have, think about the processing pressure you put on the secure perimeter, which is already loaded with Message Screening tasks, or the possibility to read an error's theatrical information within your network. Anyway, excessive messaging could unnecessarily stress your network. Therefore, cleansing must be on service at first hand, at least for proper distribution of the workload.
- Another task associated with Exception Shielding is the translation of an error's message/code. Unfortunately, this task cannot be performed on a service; that's the responsibility of the Composition Controller (within Error Handler as well). When the remote application returns a completely valid error message with an error code that is different from your service domain notation, it must be filtered/normalized according to your standards. As a typical composition controller is either an SCA- or OSB-based service, it can be done on these inner layers. Alternatively, it can be done by an adapter in the ABCS Framework using Adapter Factory (which is also OSB-based in our realization). In our design, we have the Lookup functionality, but its main purpose is to extract from the ER transformation descriptor (XQuery function or XSLT) or endpoint's URI. In this case, translation can be performed by transformation, but everything depends on the transformations' complexity. Transformation of an acknowledged message from an external service would be requited anyway, as we should not expect that its format will be the same across different domains; this would be too ideal.

If you decide to handle it in SCA, traditional Domain Value Maps will do the job with the handy `dvm:lookupValue` function. Later, we will touch upon how it can be done in a traditional way, but you surely remember that DVMs are traditionally stored in OFM MDS, and transformations could be scattered across adapter projects. So, the maintenance of any form of centralization is a question.

- After cleansing/normalization, an error's technical information is properly logged. Obvious things such as squeezing the entire stack trace dump into the 2K `VARCHAR2` field are solved during the development stage, but the question is a bit broader than the completeness of stored information. According to generic SOA principles, the core requirements for Abstraction and Discoverability (which are not exactly the best of friends) are as follows:
    - Interpretability of logged data. Data elements that are necessary for making a decision must be easily discoverable and understandable, as many policies will trust their meanings. Just think of this: not all reactions on an event should be immediate; it is desirable to have different handling policies for the same error, and it should depend on the time window and the number of event occurrences in a certain time interval. This means that some abnormal events which happen between business hours (09:00-17:00) should be handled in 5 minutes (retried, passed to compensation, send to human tasklist, or canceled), whereas 02:00-04:00 batch operations' mishaps of the same type must be handled after one hour (the question of why someone could decide to run batch jobs using SOA compositions is not critical for this example; it can be done anyway). Conditions for this kind of discriminant handling with sliding time windows could be really complex, so the data for making decisions must be precise and easily extractable.
    - Interpretability also requires proper error classification, and for immediate exceptions, this can be done on an acting service. Even simple segregation of Technical and Functional errors will be helpful for event pattern recognition, but actually, we can do a lot during the initial service design phase. It is our responsibility to identify the framework that will be hosting our service (we will identify the closest neighbours of our service) and types of compositions it will participate in. Thus, for runtime errors, it's quite possible to compose a list of errors and their causes. Right from the start, we can say that most technical errors will be related to utility services, whereas functional types belong to task-orchestrated services. This classification must be stored in the Service Repository, but we should warn you about making it too formal. Example of this registering and classification exercise will be demonstrated further.

    - The preceding requirement is the main prerequisite for a low EH footprint. The exception handling system should not consume vital server resources, essential for the business operation environment. The EH component must not only be lightweight, but the physical components should be separated as well to reduce the additional burden on EH. Obviously, the `Retry/Continue/Cancel` default resolutions are properties of OFM service engines (OSB/SCA), and the basic compensation mechanisms are attached to BPEL. However, log consolidation jobs, complex processing of events using patterns, instances of composition controllers, and running EH-related compositions can be moved to the separate environment.

- Next and probably the main EH requirement is the Error Handling automation. In the previous bullet point, we already mentioned the basic resolution operations based on primitive policies, but what we really are looking for is the automation of complex compensations, triggered by irregularities in dynamic compositions. Logically, compensation of such compositions should also be dynamic, which makes this task quite difficult. If service composition error classifications are covered diligently with all the necessary details, dynamic compensation is achievable in various ways: through the standard OFM Error Hospital and/or custom compensation execution plans similar to what we discussed in *Chapter 3*, *Building the Core – Enterprise Business Flows*, and *Chapter 4*, *From Traditional Integration to Composition – Enterprise Business Services*, which are dedicated to synchronous and asynchronous composition controllers.
- Finally, if dynamic resolution is impossible (too late or too complex), manual resolution is the only option.

Generic Audit requirements in OFM are covered by Audit Framework for any application, included in the Audit policy (yes, another policy), using two-phase event propagation with optional event filtering. Any Audit-enabled application (Java running on WLS or a web app on an HTTP server) can dump Audit data using an Audit API to local storages called bus-stops. In the second phase, an Audit loader agent will upload dumped data to the centralized DB for Log Centralization. In some cases, dispersed bus-stops will suffice, but then you will miss the analytical capabilities that are provided by the Oracle BI Publisher connected to the central DB.

These five major requirements all together present the complete scope of EH that is capable of resolving most SOA errors with reasonably reduced pressure on Ops task forces. Naturally, the complexity of the recovery operations will not disappear only with the realization of what has to be done; to make it happen, quite a considerable amount of work must be done during the service design phase and during the initial exception testing phase in the dev environment.

There must be a strict rule indicating that all draft service versions must be delivered into a JIT environment with a complete set for all possible "rainy day" scenarios. This approach is common for all types of EH resolutions; either you maintain them on a standard Error Hospital using OFM policies or you delegate them to more complex EH composition controllers, linked to the Service Repository with the required EPs. In this case, fulfilling the requirements of service composition error classifications should be the first step, and we will look at it now.

Maintaining Exception Discoverability

Although service error definition and classification is utterly important for the reasons we just saw, we only have enough room to provide directions for building and maintaining such a list in the Service Repository. This exercise is part of building a Service Profile and a Service-level profile structure (see *Chapter 15*, *SOA Principles of Service Design*, *Thomas Erl, Prentice Hall*). The basis for the exercise is as follows:

- The number of frameworks that will participate in service deployment and interactions.

  We mentioned six, but commonly, we should focus on three: EBF, EBS, and ABCS (see the next figure, where the color red indicates errors and orange indicates Audit). For the presence of the ESR framework, we should consider Inventory Endpoint as an abstraction point within EBS.

- Technical layering (vertical) and positioning of the service in core technical layers.

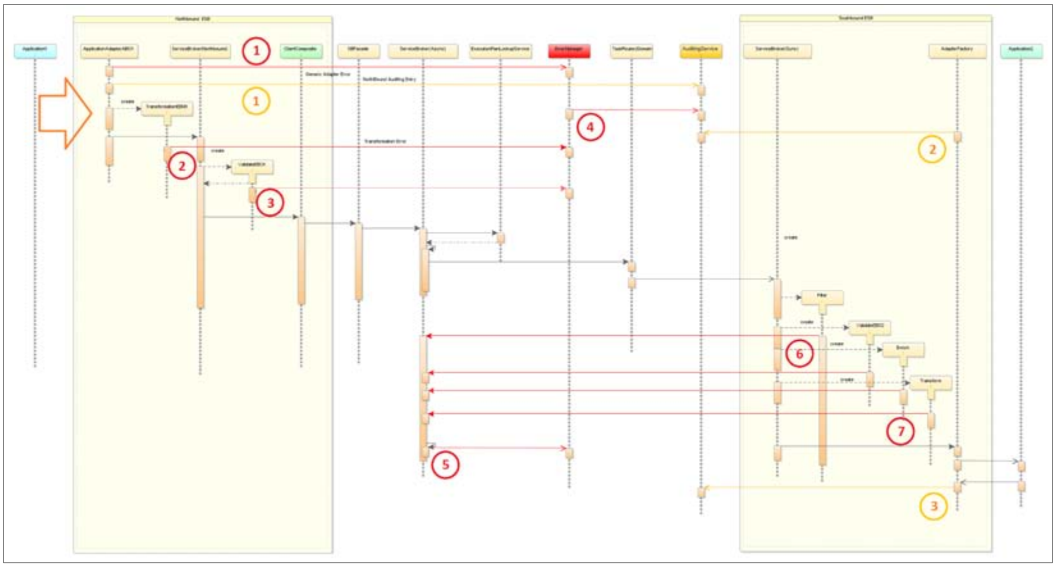
  Naturally, we have EBF as only a single layer. Bus and Adapters (not always physically, as some components/patterns are common, such as Adapter Factory) can logically be Northbound and Southbound, so the service operations will also be location-specific.

- The nature of underlying service technical resources.

  The resource acquisition error from entity services sharing the same DB should be treated with elevated priority compared to a service with a dedicated configuration file, such as FSO. Continuing on that, it seems to be that Functional type warning from task-orchestration services about gradual performance degradation must be correlated with dehydration DB runtime metrics, and so on.

- Structure of the SOAP fault and other acknowledge and error messages.

The structure of these messages must be associated with logs' structures in order to minimize mappings and transformations on logging APIs and EJBs.



Fault handling sequence diagram in Agnostic Composition Controller

In the preceding sequence diagram, which is the consolidated view of all our SOA layers and frameworks around the agnostic composition controller, starting from the left (northbound), we can identify generic EH rules as follows:

1. First, we start with the error in Application Adapter 1 (Active Poller), stating `Service-Legacy wrapper cannot extract data`. The reason why an active service adapter cannot connect to the endpoint data source is because DB is offline and Queue is unavailable. Information will be provided to the Error Handler when the common resolution operation "Retry predefined number of times" (configured on ABCS/ OSB, depending on adapter realization), failed last time. **Error Manager (ErM)** will initiate the process of sending an acknowledgment message to the Legacy Application (if it can accept it, either human or machine readable) and activate data extraction using Dual Protocol again if it's available. Technically, ErM cannot do much about this error on the north side, even if poller is a complex BPEL process that assembles data from various data sources over a prolonged amount of time into a consolidated ABO. This scenario just signifies that the source is not available and business composition cannot be initiated, so the individual legacy adapter will just send a clear message to Audit (the orange line).

At the same time, when Adapter-Aggregator is based on an agnostic composition controller, then exception handling should also be agnostic with extraction of resolution action from Service Repository. We should do this with caution, not overloading the SR endpoint with requests that involve trivial responses (OSB can perfectly handle retries with predefined intervals). The advantage of this approach is that you can monitor a controller's state (active or down) more proactively, without the need of having an additional heartbeat and external monitoring (although, it is inevitable for a passive adapter and some other components in the underlying frameworks).
Other (additional) reasons could be that the ABCS layer is down, there is a network/firewall issue, file location is detached, the queue is unavailable, load balancer is (mis)configured, and the server (node) is down.

2. The next error situation: `data extracted and assembled on the north side, but transformation from ABO1 to EBO failed`. The potential reasons for this can be that the XQuery statement is not available, the execution of XQuery returns a business fault, and business data is inconsistent. Here, we cannot blame the application-composition initiator for all these errors; if the transformation descriptor is not available, we might have severe problems with the Enterprise Repository. Recovery options are still limited, but the severity of this problem is very high. If an adapter supports asynchronous communications, the recovery operation will not involve data retransmission from the source. Technically, this error could be related to misconfiguration or erroneous installation, so it can be recognized at earlier stages.
3. When business data is invalid, we should stop processing immediately and store the inbound message in an Audit log for further investigations. The application source should be informed accordingly.
4. What if we get an exception in ErM? In this case, the ErM composition controller will not be able to invoke compensation flows or return names/code of resolution actions to individual components. This problem is quite similar to the situation where the master composition controller fails. (To avoid possible misunderstandings, the main composition controller and the EH composition controller are different instances of the same agnostic composition controller; both of them are based on the SOA pattern and promote Reuse and Composability. In order to reduce the EH footprint, you could decide to physically separate them on different WLS nodes.)
Most commonly, this situation is related to the missing/inaccessible routing slip (execution plan). This is a highly critical situation, and a warning will be issued together with a proper Audit.

This is one of the reasons why in your design, you should separate Error Management and Audit. The same is true for errors that occur in Audit, but they are definitely not life threatening and business can continue as usual. Only Error Handler can inform you about the possible causes: invalid Audit message input, JMS queue as an Audit service is unavailable, Audit MDB has malfunctioned, Audit service is down, and Audit DB is down.

> The OFM Audit Framework by default is undisruptive for business processes/applications because of the chain of bus-stops and Audit loaders. The situation could not be so simple in OFM if you decide to include `COMPOSITE_INSTANCE`, `CUBE_INSTANCE`, `CI_INDEXES`, and `DLV_MESSAGE` tables into your Audit schema to trace a runtime trail of a message flow, identified by an execution context ID (ECID) outside of the OEM console. Actually, you should do this, as all together these tables can provide you with about 100 status codes regarding the composite instance's state. We do not have space to present them here; you will easily find them in the Oracle documentation. Thus, if your Audit, based on information from these tables, hangs, it's a very bad sign of the health of your SCAs. So, do not build SQL Audit statement killers that could slow down the SCAs' database; on the other hand, missing inconsistent Audit data from these tables indicates the critical situation in SCA. Talking about the status code in general, only `STATE_CLOSED_COMPLETED (5)` should make you happy. Be extra careful about `STATE_CLOSED_STALE` and `STATE_OPEN_FAULTED`. Code 12 (running with faults, recovery required, and suspended) must have the highest priority in your recovery routines. We will talk about SCA Mediator code from `MEDIATOR_INSTANCE` a little further.

5. The next error situation (**5** on the previous figure) is similar to the previous error, but occurred on the master controller. Generally speaking, there are two main reasons that are associated with this: either we do not have something to execute or we are unable to invoke something. The first one cannot be handled at runtime, as it's a result of a design-time or deployment mistake. In the controller operation, you do not even have to stop it for fixing this situation. You can disable the failing service first in ESR, then deploy the corrected EP, and enable the service again. Situations where we cannot invoke something or receive an erroneous response from the composition member during runtime are the main reasons why we should have Error Handler with dynamic error resolutions. Exceptions will be handled automatically when: the Service Broker SCA is down; the queue for Async SB is unavailable; BPEL, DB is overloaded or down; threads stuck; called SCA is down; messages cannot be correlated; the message structure invalid; an unknown Correlation ID is provided in the response message.

6. Exceptions (parts **6** and **7** in the previous figure) in southbound ESB/ABCS layers, occurred during the execution of EP and/or VETRO pattern operations, will be propagated back to the master controller where they will be handled as mentioned in the previous paragraph. The exception causes are the same as for northbound, but the southbound layer is completely our responsibility; here, we will do everything possible to deliver the service message to the composition member. Basic reties of individual adapters will be handled in OSB locally within the business interval and only then will the error be transmitted back to SB. There are some common reasons for this: the JMS queue is unavailable for async, HTTP Servlet is down, network problems, general OSB technical problems (proxy service not found or disabled), the application server is down, application data source is unavailable, and an error situation on the adapter factory (OSB) together with responses from southbound applications/services will be properly audited (parts **2** and **3** in the previous figure).

In the following table, we will consolidate core exception handlings' properties with regards to a particular framework, a component in a framework, its role, operations performed, possible errors, error causes with relation to the operation, possible resolution actions, and areas where these recovery operations will be performed. Again, we are not going to perform the full mapping operation or present the complete list of values for the operation and possible reasons for an error to occur. This is simply because there is no space for it, and our role is to give you directions and basis for your team exercise. Anyway, your own SOA Framework layering cannot be much different from what is presented in the previous figure. So, after two to three workshops with developers, you will compose the complete spreadsheet, which will be ready to be deployed on ESR and exposed via the Inventory Endpoint for ErM lookup and fire. Regarding error code, you must map the external code to your own coding system and vice-versa if necessary. This mapping will be implemented in DVM or (better) in ESR.

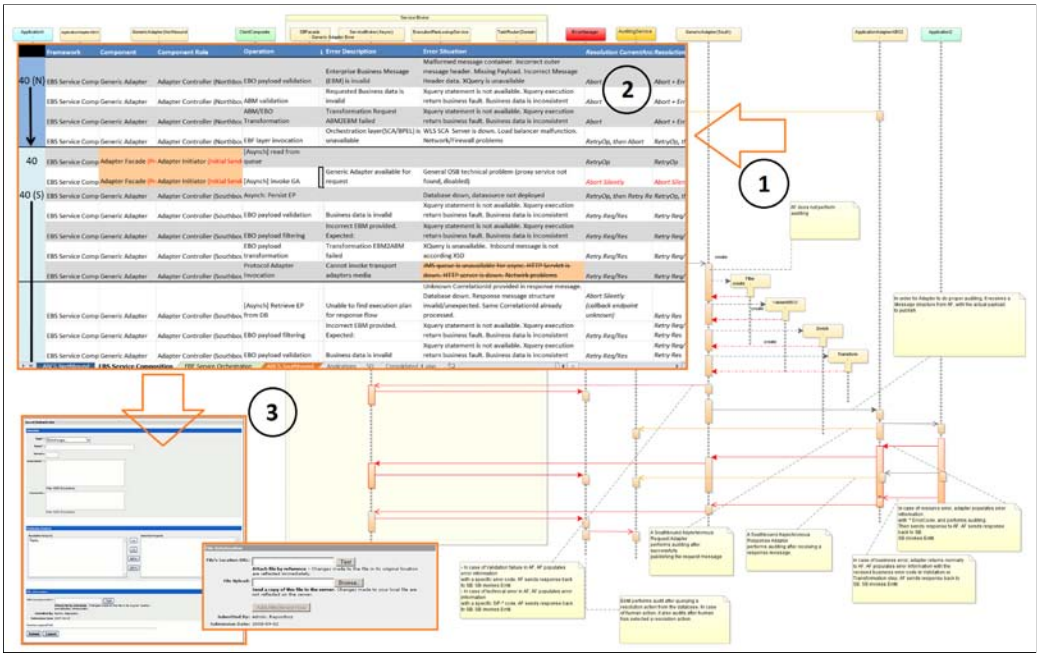
Property	Common values (the list is incomplete for brevity; extend it with your values, but avoid redundancy)
Framework	This list is complete. It includes South [ABCS1, EBS1], EBF, and North [ABCS2, EBS2]
Service Role	Some more roles can be assigned in addition to the following: Designated controller Composition controller Composition member Composition initiator Ultimate receiver

Property	Common values (the list is incomplete for brevity; extend it with your values, but avoid redundancy)
Service Type	Some more types can be found in addition to the following: Configurable Poller Generic Poller Generic Adapter Service Broker Custom Composition Member Application Adapter Protocol Adapter
Operation	Some more types can be found in addition to the following: EBM payload validation ABM validation ABM/EBO transformation EBF layer invocation Asynchronous read from queue Asynchronous invoke GA EBM payload filtering EBM payload transformation Protocol Adapter invocation SB invocation Asynchronous instantiation MDS read EP parse EP traverse Composition member invocation
Logging/ Audit requirements for functional monitoring	Put your own requirements here. We suggest (and actually have it implemented in our design) that we perform an unconditional audit on the edges south and north. The BPEL Audit level is set to Prod. Logging for errors is mandatory. The External Audit log DB is implemented for data consolidation and maintaining vendor neutrality; it is available for external monitoring tools as well as for BAM. Common Audit sources include `COMPOSITE_INSTANCE`, `CUBE_INSTANCE`, and `DLV_MESSAGE`. Add your own sources. The External Audit facility is probably the most common homebrew application. The main rule is the consistency of unified log records, for instance, services-initiators on north should provide log records containing "START ..", not "starting" or null; similarly, for ultimate receiver: "END", not "stopped" or "exit."

Property	Common values (the list is incomplete for brevity; extend it with your values, but avoid redundancy)
Error code	Canonical expression can be applied here. As an error occurs in a framework when a certain operation is performed by a component that assumes a certain role, the combination of these three codes could be quite logical. Mapping to/from external code will require an extra column.
Error description	The list is not complete. Add your own descriptors with extra care for services with external exposure. Descriptions based on the application Canonical Expression pattern could be as follows: • Transformation EBM2ABM failed • Requested business data is invalid • Unable to find XML routing slip (EP) for response flow
Possible reasons	The list is not complete. This information is for the Audit log, not for the `SOAPFault` message. Make it clear that this will be the basis for your ops teams' actions. Every new reason discovered during the exploitation must be diligently added to the following list: Cannot correlate message, message structure is invalid, execution plan XML is not according to XSD or missing XDK error, Invalid Audit message input, JMS queue as an Audit service is unavailable, Audit MDB is malfunctioned, Audit service is down, Audit DB is down, and so on.
Resolutions	The following list is not complete: Abort, ErM Notification, AbortSilently, RetryOperation, and Recover
Resolution scope	List is not complete: OSB PS, SCA BPEL, Custom Java, Network Infrastructure
Severity	This position together with the next one will set requirements for the response time: Low, Moderate, High, or Critical
Probability	Low, Moderate, or High

The structure of the spreadsheet with the initial data we mentioned in the preceding table is a balanced scorecard (see the next figure, step **2**), which is divided into several tabs for every framework. It contains the logical outcome for a combination: service + performed operation + position in technical infrastructure + process step (this is the service role). The completeness of the spreadsheet will depend on the granularity of the sequence diagrams for all use cases (step **1**).

You do not have to use Excel as presented in the following figure. Any tool will do, but in the end, all the details must be stored in ESR (Oracle or custom, step **3**) and the runtime part exposed via Service Endpoint for ErM.



	Framework	Component	Component Role	Operation	Error Description	Error Situation	Resolution Current	Arc.Resolution
40 (N)	EBS Service Comp	Generic Adapter	Adapter Controller (Northbou	EBO payload validation	Enterprise Business Message (EBM) is invalid	Malformed message container. Incorrect outer message header. Missing Payload. Incorrect Message Header data. XQuery is unavailable	Abort	Abort + Er
	EBS Service Comp	Generic Adapter	Adapter Controller (Northbou	ABM validation	Requested Business data is invalid	Xquery statement is not available. Xquery execution return business fault. Business data is inconsistent	Abort	Abort + Er
	EBS Service Comp	Generic Adapter	Adapter Controller (Northbou	ABM/EBO Transformation	Transformation Request ABM2EBM failed	Xquery statement is not available. Xquery execution return business fault. Business data is inconsistent	Abort	Abort + Er
	EBS Service Comp	Generic Adapter	Adapter Controller (Northbou	EBF layer invocation	Orchestration layer(SCA/BPEL) is unavailable	WLS SCA Server is down. Load balancer malfunction. Network/Firewall problems	RetryOp, then Abort	RetryOp, t
40	EBS Service Comp	Adapter Facade (N-	Adapter Initiator (Initial Send	[Asynch] read from queue			RetryOp	RetryOp
	EBS Service Comp	Adapter Facade (N-	Adapter Initiator (Initial Send	[Asynch] invoke GA	Generic Adapter available for request	General OSB technical problem (proxy service not found, disabled)	Abort Silently	Abort Silen
40 (S)	EBS Service Comp	Generic Adapter	Adapter Controller (Southbou	Asynch: Persist EP		Database down, datasource not deployed	RetryOp, then Retry Re	RetryOp, t
	EBS Service Comp	Generic Adapter	Adapter Controller (Southbou	EBO payload validation	Business data is invalid	Xquery statement is not available. Xquery execution return business fault. Business data is inconsistent	Retry Req/Res	Retry Req/
	EBS Service Comp	Generic Adapter	Adapter Controller (Southbou	EBO payload filtering	Incorrect EBM provided. Expected:	Xquery statement is not available. Xquery execution return business fault. Business data is inconsistent	Retry Req/Res	Retry Req/
	EBS Service Comp	Generic Adapter	Adapter Controller (Southbou	EBO payload transformation	Transformation EBM2ABM failed	XQuery is unavailable. Inbound message is not according XSD	Retry Req/Res	Retry Req/
	EBS Service Comp	Generic Adapter	Adapter Controller (Southbou	Protocol Adapter Invocation	Cannot invoke transport adapters media	JMS queue is unavailable for async. HTTP Servlet is down. HTTP server is down. Network problems	Retry Req/Res	Retry Req/
	EBS Service Comp	Generic Adapter	Adapter Controller (Southbou	[Asynch] Retrieve EP from DB	Unable to find execution plan for response flow	Unknown CorrelationId provided in response message. Database down. Response message structure invalid/unexpected. Same CorrelationId already processed.	Abort Silently (callback endpoint unknown)	Retry Res
	EBS Service Comp	Generic Adapter	Adapter Controller (Southbou	EBO payload filtering	Incorrect EBM provided. Expected:	Xquery statement is not available. Xquery execution return business fault. Business data is inconsistent	Retry Req/Res	Retry Req/ Retry Res
	EBS Service Comp	Generic Adapter	Adapter Controller (Southbou	EBO payload validation	Business data is invalid	Xquery statement is not available. Xquery execution return business fault. Business data is inconsistent	Retry Req/Res	Retry Req/ Retry Res

Defining fault resolution actions

In step **3**, you will have to enter the results of your analysis into ESR assets (Service Profiles in this case). Optionally, you can upload the source of the entire spreadsheet. Sorry, we do not know how to import Excel into ESR with one click, but if the amount of discovered error-handling data is enormous (please do not overcook it, one step at a time), you can construct `insert` statements using ER (*Chapter 5*, *Maintaining the Core – the Service Repository*, which is dedicated to Enterprise Repository Taxonomy). Custom ER that is based on lightweight taxonomy (covered in the previously mentioned chapter) is much easier, and we used the last figure for building SQL `insert` statements for data import.

Now, we can see that handling exceptions in complex compositions will be inevitably complex as well. Similar to security, this complexity can be restrained by proper layering and application of SOA patterns as follows:

- Policy Centralization (fault policies first of all) and Metadata Centralization (steps **1**, **2**, and **3** in the preceding figure)

- Enterprise Repository and Inventory Endpoint
- Logic Centralization (with focus on exception handling)
- Exception Shielding (in assistance to service security)

Now, we are ready to formalize common design rules for taming this complexity.

Error-handling design rules

Common requirements combined with discovered EH handling properties are expressed in the following points. By combining them together, we will strive to deliver fault-handling solutions for the entire SOA infrastructure, where composition controller(s) are the cornerstone. A single component's exception handling (only on BPEL or OSB) is not enough for enterprise-wide implementation, as it will lead to fault misses and mishandling of security threats:

- **Rule 1**: Error Handling Centralization on SB. EH Centralization will be maintained for agnostic **Service Brokers** (**SBs**) that broker service calls between heterogeneous components. Primarily, it must unify the OSB and SCA fault-handling frameworks, allowing error recognition (listening), catching, and propagation of unified policy-based handlers. Two handlers will be maintained for sync and async errors' resolution. This maintenance denotes reuse, as we have already demonstrated two types of Service Brokers that are capable of handling *happy* and *unhappy* tasks. Processing centralization will be based on three other Centralizations (Policy, Logic, and Metadata).
- **Rule 2**: Extend **Metadata Centralization** on Logs. Centralization must be applied to the metadata (service and fault object/message), policies, and logs. The immediate result of this centralization is the decreasing of fault message **Model/Format Transformations** between fault frameworks and fault handlers (logging MDBs, for instance). The main goal is to present ESR with all the metadata elements centralized. The MDS is an adequate option for SCA's metadata and policies.
- **Rule 3**: Contemplate the delegation of primitive recovery actions to service edges. Metadata Centralization after analysis, similar to the one offered previously in the *Maintaining Exception Discoverability* section, will present the list of common faults and primitive recovery actions (`Retry`, `Abort`, and `Continue`) that can be immediately delegated to service edges (OSB and ABCS, the first line of EH handling) and provide ways of fault elevation/propagation back to SB (second line). Common faults should include the standard faults defined in the WS-BPEL specification (20 generic faults; see the BPEL 2.0 documentation). Redundancy with your own custom code will be avoided (don't reinvent the wheel, just discover and abstract).

- **Rule 4**: Balance the **Policy Centralization** on the platform/frameworks. Policy Centralization will be applied to a meaningful extent. This means that by being realistic, we can hardly present some kind of universal policy (as one XML file), suitable for all OFM elements and frameworks. For instance, in SCA, we have fault policy binding for BPEL, Mediator, and SOA composite, and we reference them in `composite.xml`. The location of policies can be defined using the `oracle.composite.faultPolicyFile` property and their bindings to the SCA components in the `oracle.composite.faultBindingFile` property. The physical location quite often is MDS (`oramds:/apps/faultpolicyfiles/...`). In OSB, we have OWSM (including basic security) and WLS 9 (including WS-Policy compliant) policies, and the first one is located in the OWSM policy store. Should we mention the API gateway policy store? So much for centralization. Thus, the rule is to maintain the policy descriptors (in human-readable form) in ESR and clearly define their areas of application. Using this definition one level down, in SCA, we can present clear policy chaining for automated resolutions (`policy-id1 -> policy-id2 -> policy-idN`), where it's possible.
- **Rule 5**: Minimize log data cleansing by applying the **Canonical Expression** pattern. Log Centralization (not an SOA pattern) and activities around data collection and cleansing could be a very heavy and complex operation. No, this way, *very complex*. Why? We will demonstrate it later. For now, consider a separate DB and servers for log assembly/cleansing (depending on the complexity of your logs, of course). The rule of thumb is that the more diligent and accurate you are at the initial design stages (previous figure), the fewer problems you will have at later stages. This rule is so important that we should put it first. Break it (just misspell the stage or resolution code, or name it differently in ESR and your Java/BPEL code) and all further rules will have values that will be close to nothing. The Discoverability SOA principle rules the error-handling process.
- **Rule 6**: Apply ErM based on Service Roles. Observe service roles when applying rule 4. Complex global policies applied to composition members (SCAs), invoked and handled by the agnostic composition controller (SB) dynamically instead of fixing the error, could make the situation much worse. Simply put, errors in complex dynamic compositions must be handled (at least propagated to) by the same composition controller where all composition members are initially registered. An individual member will have no idea about the complexity of a composition. Thus, Error Hospital and OFM Fault Management Framework, in general, are more suitable for static (defined during design time) compositions (classic BPEL Mediator SCA from Fusion Order Demo). At the same time, standard policies (`fault-binding.xml+ fault-policies.xml => composite.xml`) can be good candidates for an agnostic master SB and individual adapter services for the north side.

- **Rule 7**: Maintain strict control on the ErM components footprint on Services and Compositions. As a logical outcome of rule 3, EH decentralization or service decomposition (if we assume Error Handler as a service utility type) will reduce an exception-handling footprint on the service infrastructure. Depending on your hardware specs and daily business workload, you will set thresholds for EH activities such as 5 percent CPU, 10 percent RAM, and 10 percent of total threads (just an example).
- **Rule 8**: Apply the **Redundant Implementation** pattern on critical elements of ErM. Redundant implementation of elements of the SOA infrastructure (WLS Nodes and Node manager, and SAN RAC DB) is an absolute must for a resilient HA infrastructure; however, from the EH standpoint, it is also prudent to consider a separate node for ErM SB.
- **Rule 9**: Provide alternative execution paths for business-critical service compositions. You cannot fix faulty OFM using the same OFM. **Automated Recovery Tool (ART)**, or whatever you call it, must be in the utility service layer with a very shallow technical infrastructure and a maximum of one or two service engines, at least part of it. Just think of this: if an airplane loses its hydraulics, usually it is doomed as it cannot steer. The bitter irony is that there are still plenty of means to keep the plane safe and maneuverable: we could still have the engine's thrust and use it; alternatively, inbound air flows could power a small generator for elevator/rudder servos. Why is it not implemented? Actually it is, but only for some special (mostly military) cases because it's rather expensive. Is it also expensive in SOA? Not at all! We have presented universal XML EPs, and we clearly demonstrated how they can be executed by different engines (BPEL, JEE, XMLDB). So you have it already. Additionally, consider Dual Protocols for helping different engines consume EP and messages (for instance, OraDB can consume an HTTP call more easily than HTTP/SOAP; although, SOAP is not an issue either). Sync/Async pairs are always good, especially with Oracle AQ.
- **Rule 10**: Avoid SPOF on MetadataStore and provide alternative means to access it. Metadata Centralization, like with any type of centralization, has a natural flaw: SPOF. Its weak point is **Inventory Endpoint**, which is used by `ExecutionPlanLookupService` in our design (the same is true for any design). In addition Service Grid and Redundant Implementation, consider the dual storage type for your Metadata Storage in ESR. Technically, it's a service data replication (we can assume service metadata elements that are consumed by SB as its sole data) for execution plans. The realization of this design rule is exceedingly simple; in fact, you have it already. As you will remember, initially we implemented EPs as FSO, stored as simple files, and decomposed them as DB elements in *Chapter 5*, *Maintaining the Core – the Service Repository*.

The `ExecutionPlanLookupService` is capable to work with both implementations, abstracting physical storage resources. Thus, a simple script or service can easily dump the EP from the database every time some changes occur; how it works was explained in *Chapter 6*, *Finding the Compromise – the Adapter Framework*. The same is true for the Oracle ESR implementation, where standard synchronization ESR to the UDDI utility can be extended with an additional functionality for EP FSO. Adhering to this rule will eliminate the risk of ESR runtime unavailability quite gracefully. Unfortunately, it will present a risk when a developer tries to cut some corners and perform alterations directly on FSO, bypassing ESR. Also, this could be the case for an insecure direct object reference vulnerability; see *Chapter 7*, *Gotcha! Implementing Security Layers*. Well, there is no perfection in the world. The first issue can be addressed with proper governance (set the watchdogs around your Service Repository). The second is about establishing a Trusted Subsystem pattern and avoiding using SB in adapters at the Service Gateway for external services (which is not always possible). This rule can also be associated with the newly introduced Reference Data Centralization, but we believe that the existing Metadata Centralization pattern is overkill. There is no need to inflate the SOA pattern catalog with obvious things.

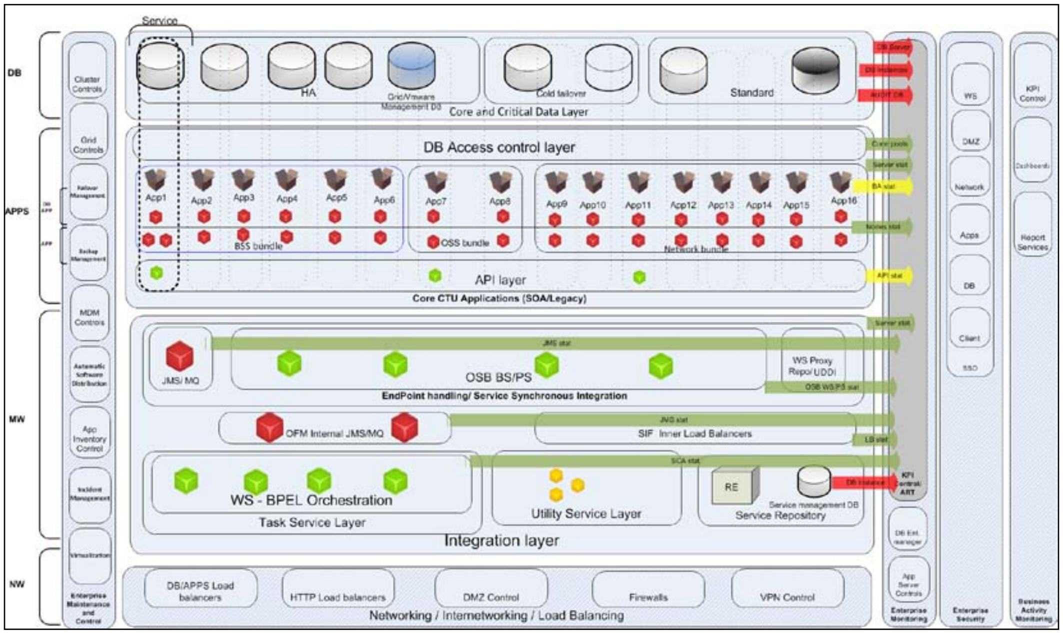
- **Rule 11**: Apply EM rules according to service layers. Passive Northbound adapters do not propagate validation errors to EBF. Transformation errors related to missing XSLT should be propagated, but their probability is rather low (usually related to incorrect deployment and fixed in the first hour).
- **Rule 12**: Propagate the errors according to service layers. VETRO errors from the south will be propagated back to the EBF SB after completing basic recovery operations (if assigned).
- **Rule 13**: Observe Service Message TTL when applying EH rules. Combined logical outcome of rules 11 and 12 strictly observe service message time-to-live (business validity period, set in MH). If business data is actually set to 30 minutes only and you set the `Retry` option on OSB with three different intervals of 5, 15, and 20 minutes, it will do no good. Again, work diligently on Metadata Centralization (previous figure) and define your business exception policies clearly. Use SOAP/SBDH header elements for propagating business policies to local Error Handlers, together with the State Messaging pattern (Message Tracking Data section in CTU EBM, explained in *Chapter 5*, *Maintaining the Core – the Service Repository*).

- **Rule 14**: Maintaining preventive Error Management must be preventive. However, avoid using heartbeat test messages for performing ABCSes (both passive and active, especially for active) health check, either for north or south. This is a valid technique for JIT and Unit tests, but not for production; it could (and most probably will) complicate everything from data cleansing in the production database to opening the door for attackers' message probes. There are plenty of other methods to check the status of your service edges, and JMX should be one of your first choices. Oracle OEM and OEM management packs are shipped with predefined thresholds for main vital parameters. Please use them. It might be funny if it weren't so sad; how many problems could we avoid if the DBA (and Nagios) admin reacted properly on clear `99% of ORABPEL table space is full`?
- **Rule 15**: Automated Recovery Solution should be SB-based. Again, Error Management must be preventive. This means that our service edge handlers (with retries, abort and so on), individual service handlers (BPEL Fault Handlers, Compensative Transactions, and Mediator policies), and SB-centralized policy-based handlers are in fact reactive and not truly effective (yet essential). Real prevention comes from the already mentioned Log Centralization, just-in-time analysis of all technical information from the entire technical infrastructure (through JMX, WLST, or Jython), and an immediate response. To come to this realization at this point sounds like telling your 18-year-old kid that there is no Santa. Actually, this realization just spawns another rule in addition to rules 9 and 14; a proactive component that contains ART will be presented effectively, controlling the already mentioned Log Centralization: data log cleansing, on-the-fly analysis, and triggering preventive or recovery actions. Yes, this is actually the event-driven SOA, a combination of **Event-Driven Network, Event-Driven Messaging**, and **Complex Event Processing (CEP)**; although, for reasons explained in rule 9, we cannot put all our eggs into the "OFM Mediator with BPEL Sensors and Events declared in BPEL's Invoke Operation" basket (those are the main OFM EDN players). Actually, it's not possible technically as we will control the entire OFM's underlying infrastructure; refer to the next figure.

Every rule paragraph is considerably bigger than one line; however, we have not only manifested the rules, but have also put clear reasons behind each of them for you. These reasons are not only the result of the following SOA design principles and SOA patterns (both clearly indicated), but the quintessence of our own combined experience from various projects. Some say rules are meant to be broken. Go on, break these rules, and see what will happen. The severity of consequences will depend on the complexity of your compositions, from mess in static until disaster in dynamic. The best outcome would be to park every fault for manual resolution, which is far from optimal. Again,these rules are for the implementation of an agnostic Composition Controller. For static task-orchestrated processes the standard OFM Fault Management Framework will suffice and for this type of implementation please follow the rules 3, 4, 11, 12, and 13. We also have to mention that this list can be extended with your own rules, as we could have the design variation on south, around Adapter Factory.

Also, please bear in mind that we will use the rule numbers (from 1 to 15) all the way in this chapter. The number 15 will be some kind of magical number here, because at the end we will see how 15 design rules refer to 15 main Audit event sources.

From top to bottom, one rule is leading to the next, thus summarizing all the preceding 15 rules and SOA patterns (presented in bold) around Preventive Error Managing (rule 15). We can extend the SOA infrastructure diagram from *Chapter 7, Gotcha! Implementing Security Layers* (related to security, the first figure), with all the layers—the sources of monitoring information, required for proactive management and automated recovery:



Technical and Functional monitoring flows in a typical SOA infrastructure

Basis for proactive Fault Management

For TOGAF architects, the preceding diagram will present some resemblance to generic horizontal layering, which is similar to the OSI Reference Model where each layer provides services to the surrounding layers. We exclude some generic enterprise layers in order to focus on the SOA enterprise model, as it is realized in OFM. The difference is that OSI mostly depicts seven layers between applications, from one API to another (that is, integration), which is not applicable for service compositions where an individual service spans across several technical and logical layers. We have no intentions here to map the TOGAF/OSI model to SOA (actually, this theoretical exercise is done already). The sole purpose of this diagram is to illustrate the KPI-monitoring sources, their types, and input according to rules 5, 9, 14, and 15. WLS, obviously, is the main OFM server and the Oracle database (XE, standard, or enterprise), so technical monitoring for proactive error handling/prevention will be focused around WLS JMX (green) and database server metrics (red). Functional monitoring is based on the information provided by SOA applications and their APIs (SCA and OSB, yellow flows). Generally, that's what we get from BPEL's activities to `Audit`, `Catch`, `CatchAll`, `Throw`, and OSB's `Log()`. We have to omit the monitoring of the network infrastructure for brevity; as an enterprise architect, you should keep this in mind.

Technical monitoring for proactive Fault Management

Just declaring the rules and listing the SOA patterns to prevent fault handling provides you with little practical help. For green information flows (JMX), we have grouped some core MBeans and their attributes that you should monitor; please see the following table:

Resource category	**MBean name**	**MBean attribute**
Threads	`MinThreads Constraint Runtime`	`OutOfOrderExecutionCount`, `PendingRequests`, `CompletedRequests`, `MaxWaitTime`, `CurrentWaitTime`, `ExecutingRequests`, `DeploymentState`, `InvocationTotalCount`, `ExecutionTimeTotal`, `ExecutionTimeAverage`, `PoolMaxCapacity`, `HealthState`, `ConnectionsCount`, `MessagesSentCount`, `ServerConnectionRuntimes`, and `MaxCapacity`
	`MaxThreads Constraint Runtime`	`CurrentCapacity`, `MaxCapacity`, `ExecutionTimeTotal`, `ExecutionTimeAverage`, `PoolMaxCapacity`, and `HealthState`
	`ThreadPool Runtime`	`PendingUserRequestCount`, `CompletedRequestCount`, `ExecuteThreadIdleCount`, `QueueLength`, `PoolMaxCapacity`, `ExecutionTimeHigh`, `MaxCapacity`, `JMSThreadPoolSize`, `MaxMessageSize`, `DestinationsTotalCount`, and `HoggingThreadCount`
JVM	`JVMRuntime`	`HeapFreePercent`, `JavaVersion`, `HeapFreeCurrent`, `HeapSizeMax`, `HeapSizeCurrent`, `InvocationTotalCount`, `ExecutionTimeTotal`, `ExecutionTimeAverage`, `ExecutionTimeHigh`, `MessagesPendingCount`, and `MessagesReceivedCount`

Resource category	MBean name	MBean attribute
JMS	JMSRuntime	JMSServersCurrentCount, JMSServersTotalCount, HealthState, and JMSPooledConnections
	JMSDestination Runtime	MessagesCurrentCount, MessagesPendingCount, and MessagesHighCount
Queues	ExecuteQueue Runtime	PendingRequestCurrentCount
JDBC	JDBCService Runtime	HealthState, JDBCMultiDataSourceRuntimeMBeans, JDBCDriverRuntimeMBeans, JDBCDataSourceRuntimeMBeans, ConsumersTotalCount, ConsumersCurrentCount, MessagesPendingCount, MessagesReceivedCount, and MessagesSentCount
JTA	JTARuntime	TransactionRolledBackTimeoutTotalCount, TransactionRolledBackTotalCount, and TransactionAbandonedTotalCount
OFM Server Engine	ServerRunTime	Healthstate, State

The preceding attributes are selected from the thousands that are available on WLS; you can add (or exclude) any MBean attribute as you deem prudent. Please refer to the documentation at http://docs.oracle.com/cd/E12839_01/apirefs.1111/e13951/core/index.html regarding each of them, their meanings and metrics; also, check for new and deprecated ones (we do not have space for this here). The ones presented here are the most common ones (actually, those are the ones that we use and we strongly advise you to do the same); these are sufficient for detecting and reacting to most bottlenecks in the Oracle SOA infrastructure.

The ways in which you will implement regular attributes to pull data could be different, and they depend on your KPI consolidated solution. If you already use the mentioned Nagios, check for the Nagios plugins project and WLS plugins on Nagios exchange. Naturally, Oracle has quite a lot of their own tools in addition to the classic Enterprise Manager console on top of a **Diagnostic Framework (DFW)**, and one of the most powerful tools is the Remote Diagnostic Agent (RDA) utility.

To get it, you must have an Oracle support account (old metalink); usually, it is used on one of those unhappy days when you need to generate diagnostic dumps for Oracle's technical specialists, for instance, OSB as `rda.cmd -vSCRP OSB`.

So, RDA is a manual tool by default for collecting static configuration information and runtime statistics. This is not really useful for proactive runtime monitoring and fault prevention, is it? Yes, it's part of the disaster recovery plan (your Orange Book), but we have one good use for tools such as RDA. Some of the typical scenarios were as follows:

- A new release of the SOA application bundle (SCA and OSB) was delivered from UAT to ORT (operation readiness environment) after passing all the tests.
- Performance issues were detected on **Operation Readiness Test (ORT)**, and some fine WLS/SOA Server tuning was applied by admins after consultation with the developers. Traditionally, last-minute changes weren't logged on the ops wiki (that's never happened in your organization, right?).
- After deployment in Production, considerable changes in performance became obvious. Stabilization attempts had no positive effect (or little).
- Even if said otherwise, obviously we have three different SOA runtime environments (assuming that VMs or physical servers are the same).

Dumping the MBeans attributes (as XML, for instance) and automatically comparing the static configs and runtime metrics under the same load from the servers in question will reveal the difference. The problem can be fixed without the traditional swop ORT <-> Prod by RDA. In any case, creating and keeping the *healthy dump* as a reference will be a good start for proactive monitoring.

Actually, by default, you'll be provided with the ability to automatically collect and store diagnostic dumps along with SOA Suite, and it will be provided by Oracle Diagnostic Framework (DFW). As it is primarily related to SOA Suite, it does not cover the WLS configuration and runtime metrics, but it can seamlessly work with WebLogic Diagnostic Framework (WLDF), which will supply DFW with notifications about exceptions. You even get a preconfigured FMWDFW Notification in your DFW after the SOA Suite is deployed.

With this, you understand that these dumps must be stored somewhere, and Automatic Diagnostic Repository (ADR) exists in every managed server in a domain for this purpose (look at `<SERVER_HOME>/adr`). Bear in mind that ADR is not the SOA Suite log you usually read every time you look for the initial diagnosis records. Oracle Diagnostic Logging (ODL) is the basic and primary means of any OFM applications' logging, recording every single step in great detail.

In addition to the traditional Timestamp (actually, several of them), Message ID, and Message text, you will get `MODULE_ID`, `THREAD_ID`, and `PROCESS_ID` that can be correlated to the MBean data acquired using JMX/WLST from other preventive monitoring flows (see the previous figure). Talking about correlation, what's particularly interesting is that it presents us with the **Execution Context ID (ECID)**, a global unique identifier for a service request, and an execution environment for handling this request. The main purpose of this ID is to link error messages from different components, and it will be a good idea to also use it as the basis for your Business Correlation ID within your SCA. We will show you how to do this a little later.

What is the role of RDA in this log's life cycle? The last incidents (the last 10 in total by default in Version 11.1.1.6) will be collected by RDA and incorporated into other information gathered using JMX, static, and runtime. Diagnostic Dumps and Incidents (which are a collection of dumps and are created by DWF, stored in ADR, and aggregated by RDA) can be:

- Adjusted to be more sensitive to log details, that is, additional details can be collected at a certain level
- Bundled and packaged for uploading to Oracle support
- Purged from time to time after they are uploaded for proper analysis

The preceding points are essential tasks for Log Centralization, and Oracle has two additional instruments for them. The first one, Selective Tracing, is available in Version 11.1.1.4; it's a response to one of the major requirements: a low monitoring footprint with an adequate level of tracing. This feature is managed by OEM or, more selectively, by WLST, and fine-tuned selectivity can be based on any attributes (fields) of ODL.

The last two tasks are covered by Automatic Diagnostic Repository Command Interpreter (ADRCI). Later OMF versions include the Perl-based utility, the so-called Incident Packaging System (IPS), which is capable of packaging offline RDA bundles for uploading. To some extent, it competes with the RDA itself.

Everything mentioned in the previous points fit very well to the Log Centralization, incident management, and SOA governance in general, but RDA, ADRCI, DFW, and even WLDF are tools too reactive to be truly preventive. Indeed, aftermath dumps will not contribute much to pulse monitoring of a runtime SOA. Yes, it's true, but you as an architect will be aware that these tools work well in order to advise your ops where to start when it comes to investigate complex composition errors scenarios (not just bare metal or bare OFM infrastructure faults). It is also true that for all these tools and instruments, we have a critical part, which is purely the runtime main diagnostic information provider: Dynamic Monitoring Service (DMS).

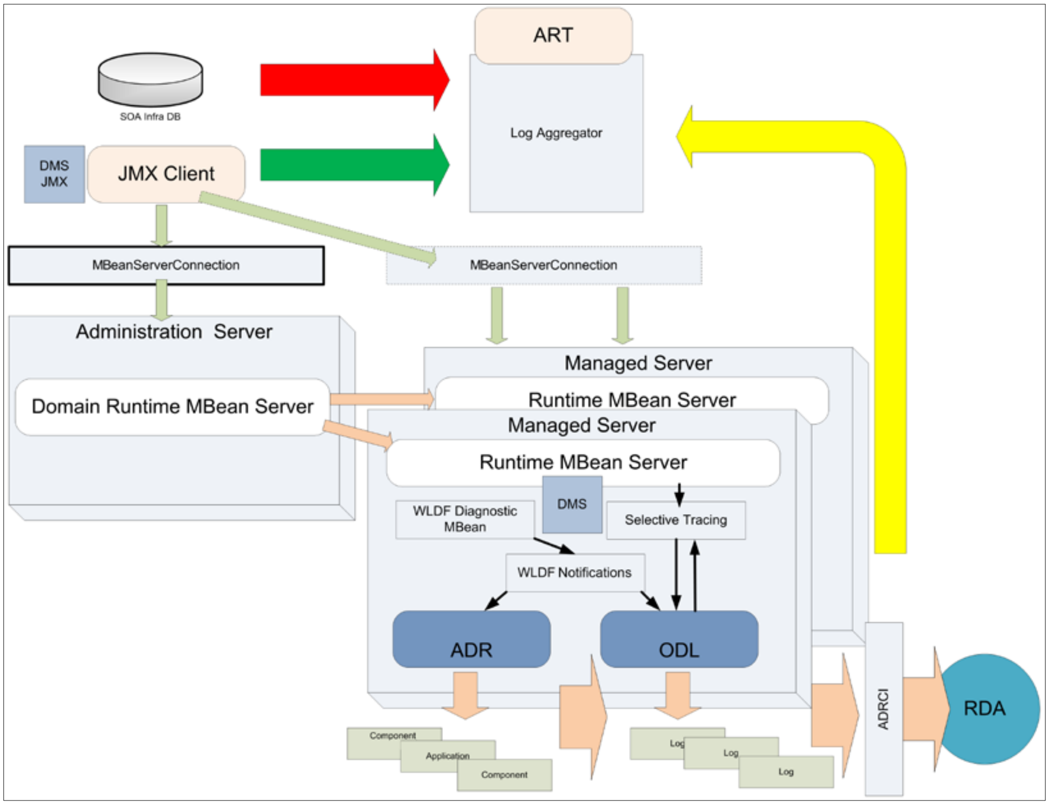
DMS is delivered by default in the form of a servlet web app: `<ORACLE_HOME>/modules/oracle.dms_<your_version>/dms.war`. It can be accessed using `http://telco.ctu.com:7778/dms/Spy` (set your host and port accordingly). You will find a broad range of parameters that are combined in a collection of noun types and their individual attributes and exposed as MBean instances. DMS presents them at runtime for monitoring, which is performed through the already mentioned WLDF. This monitoring is organized in the form of WLDF watches, monitoring particular DMS attributes and their thresholds. Should we say again how important it is to include all the parameters from the previous table in your monitoring pattern?

A notification is fired by WLDF Watches every time a threshold is crossed, and DFW will create an incident in addition to the incidents that DWF can pull from ODL events. You should know that three WLDF watches are configured by default in the SOA Suite for deadlocks, stuck threads, and unchecked exceptions. They are bare essentials; please extend the watches according to the information from the previous table. You should also be aware of the latest SOA Suite diagnostic dumps, based on watches for the following:

- `soa.composite.trail`: These are notifications from your running composites. They are highly important for dynamically running compositions.
- `soa.config`: These are errors in deployment configuration, and they include MDS as well.
- `soa.db`: This provides DB information on SOA-Infra DB and its repository.
- `soa.wsdl`: This provides information on contracts/endpoints.

The preceding list is not complete. For more information about scopes, errors, and error codes, please refer to `http://docs.oracle.com/cd/E17904_01/core.1111/e10113/chapter_ows_messages.htm`.

This very quick walkthrough we had around Fusion DWF seems to be rather complex. We tried to simplify the complexity of DWF composite relations at a functional level in the following figure, but it cannot be used as a reference model for operations planning:



Actually, the main purpose of the preceding diagram is to demonstrate how all three information flows can be consolidated around the centralized Logs Aggregator and thereby ultimately feed the Automated Recovery Tool. It is also obvious that you have complete freedom to implement your own lightweight JMX, monitoring the client for preselected parameters in addition to the existing standard OFM DMS. It will also provide you with an understanding of how a DMS Servlet is organized. As it is not the purpose of this book, the MBean attribute parsing the servlet code is omitted for brevity, but you can find perfect working examples provided by Steven Haines in his book *Pro Java EE 5 Performance Management and Optimization*, 2006 (yes, this book is probably a bit old, but still brilliant and useful).

Just to give you some ideas about servlets' implementation in conjunction with another tool, keep in mind that different log aggregators have different MEPs around the data flow; for instance, Nagios is generally a puller, so you can even consider converting Steven's JMX Servlet into a REST service by providing MBean attributes as JSON or XML. Initially, the proposed servlet has two main parts. The first one is the abstract where all operations are declared, including the main one, that is, service:

```
public abstract class AbstractStatsServlet extends HttpServlet
public void service( HttpServletRequest req, HttpServletResponse res )
throws ServletException
MBeanServer server = ( MBeanServer )this.ctx.getAttribute( "mbean-
server" );
// Ask the Servlet instance for the root of the document
Element root = this.getPerformanceRoot( server, objectNames );
// Dump the MBean info
Element mbeans = new Element( "mbeans" );
for( Iterator i=objectNames.keySet().iterator(); i.hasNext(); )  {
String key = ( String )i.next();
Element domain = new Element( "domain" );
domain.setAttribute( "name", key );
Map typeNames = ( Map )objectNames.get( key );
for( Iterator j=typeNames.keySet().iterator(); j.hasNext(); ){
String typeName = ( String )j.next();
Element typeElement = new Element( "type" );
typeElement.setAttribute( "name", typeName );
List beans = ( List )typeNames.get( typeName );
for( Iterator k=beans.iterator(); k.hasNext(); )    {
ObjectName on = ( ObjectName )k.next();
Element bean = new Element( "mbean" );
bean.setAttribute( "name", on.getCanonicalName() );
// List the attributes
if( showAttributes ){
try {
MBeanInfo info = server.getMBeanInfo( on );
Element attributesElement = new Element( "attributes" );
MBeanAttributeInfo[] attributeArray = info.getAttributes();
for( int x=0; x<attributeArray.length; x++ )   {
String attributeClass = attributeArray[ x ].getType();
//set XML attributes for class:  is-getter, readable, writable
attributeElement.setAttribute( "description", attributeArray[ x
].getDescription() );
            // Output the XML document to the caller
            XMLOutputter out = new XMLOutputter( );
            out.output( root, res.getOutputStream() );
            ......
    }
```

The result is the creation of a complete DOM document with your MBean's attributes tree. Using domain keys and attribute names, you can filter it or just construct the required part.

This abstract class is extended by the JMX statistic servlet; this is where you connect to your server and gather statistics:

```
public class StatsServlet extends AbstractStatsServlet
....
      String config = getServletContext().getResource("/WEB-INF/xml/
stats.xml").toString();
      SAXBuilder builder = new SAXBuilder();
      Document doc = builder.build( config );
      Element root = doc.getRootElement();
      Element adminServer = root.getChild( "admin-server" );
      String port = adminServer.getAttributeValue( "port" );
      url = "t3://localhost:" + port;
      username = adminServer.getAttributeValue( "username" );
      password = adminServer.getAttributeValue( "password" );
```

As you can see from Steven Haines' code, the connection is established with the admin server to acquire information from runtime servers. The direct connection to manage a server bean, as depicted in the earlier figure, is not advisable.

Why this is so important is clear from the second table in this chapter, and we advise you to familiarize yourself with Steven Haines' book and Oracle DMS Servlet in particular. We have more than 16,000 MBean attributes in WLS, and you have to pick the correct ones from the beginning, understand their roles and relations, and monitor them diligently.

If for some reason you have no time for this, but the necessity of configuring Log Centralization/Aggregation by external tools is clear to you (see rule 9), then you can look at open source tools such as Jolokia (`http://www.jolokia.org/`). Jolakia is a JMX-HTTP connector with many adapters to many servers, including WebLogic (9.2.3.0, 10.0.2.0, and 10.3.6.0 at the time of writing this book). Technically, it will be the same servlet as mentioned earlier, and you should use it together with DMS for complete monitoring.

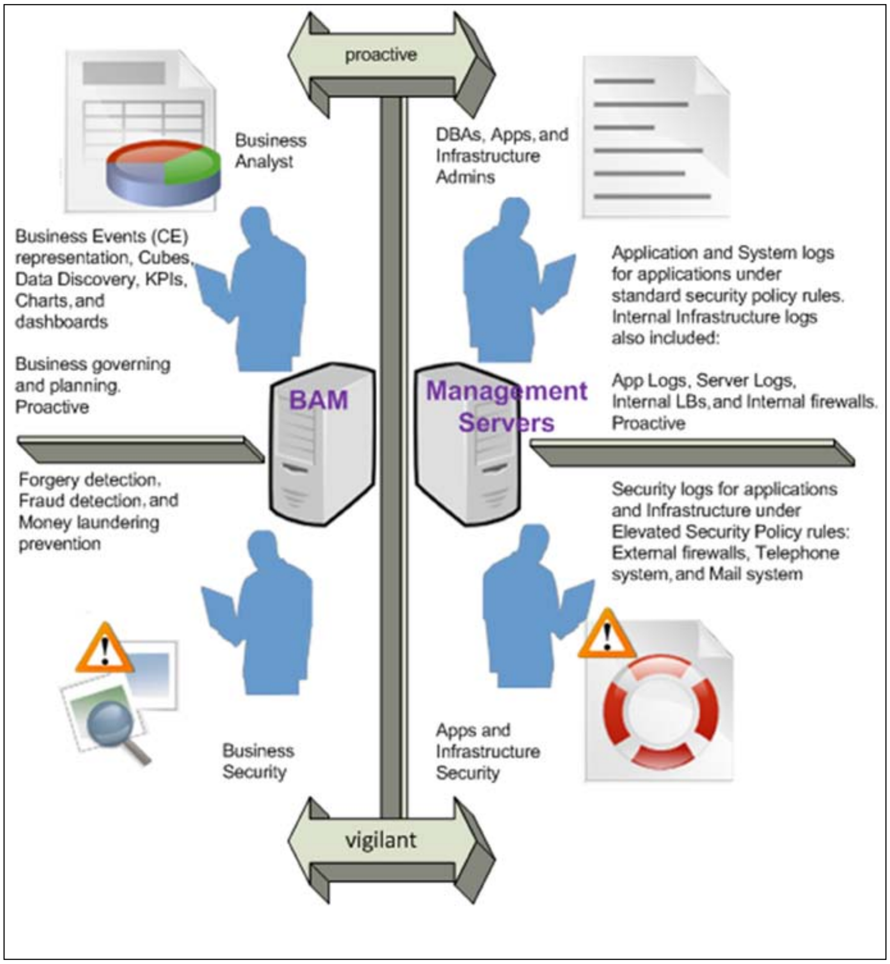
Continuing on with rule 5 (Log Centralization), we cannot avoid discussion, as old as the hills, about the consolidation of security, operational, and business logs. Our advice from *Chapter 7*, *Gotcha! Implementing Security Layers*, was to deploy the Oracle API Gateway for proper implementation of all eight SOA security patterns (for static service deployment and message in transit). Please see the following sample of Jython code for extracting web service names from the gateway and displaying them on the Nagios dashboard. You can extend it in parts by gathering more service attributes for completeness and better visibility:

```
list the web services in a Gateway
from java.lang import Integer
from deploy import DeployAPI
from esapi import EntityStoreAPI
from vtrace import Tracer
import common
t = Tracer(Tracer.INFO) # Set trace to info level
dep = DeployAPI(common.gw_deployURL, common.defUserName, common.
defPassword)
es = dep.getStore('')
webServices = es.getAll('/[WebServiceRepository]name=Web Service
Repository/[WebServiceGroup]**/[WebService]')
i = 0
for webService in webServices:
               name = webService.getStringValue('name')
               //gather here all statuses for each web service
                ....
               t.info(Integer.toString(i) + ': ' + name);
               i = i + 1
es.close()
```

Regarding the consolidation of different log types, we have to stress the fact that we have been talking about different technical types from/within generic SOA technical frameworks (two previous figures, first and third). We are quite far from suggesting that you should put all logs in one location (DB) and monitor them from the same dashboard by the same personnel. Trying to find analogy, we can say that even if it is possible to put all of the corporate traffic in one TCP/IP-based backbone (for cost optimisation, for instance), security guys and emergency brigades would never be happy to have fire sensor wires, VIP landline phones, and intrusion detection channels combined with a regular business network (simply put, it' a bit more than just SPOF).

Similar to this, we have not one, not two, but many types of logs, which will be grouped for diligent monitoring by different teams of experts, sometimes reasonably separated; please see the following figure. Although it could be annoying, there are some strong reasons why SOA business ops cannot have immediate access to Secure Gateway or IDS logs. At the same time, Security and SOA Architects must constantly coordinate their efforts on a regular basis, and what is absolutely unacceptable is when DB logs are kept only for DBA.

Thus, a comprehensive but still a minimal model on the following figure must be designed as the basis; however, it should be carefully adjusted according to your business model and Industrial Policies/Regulation (PCI, Healthcare, and so on).


proactive
Business Analyst
DBAs, Apps, and Infrastructure Admins
Business Events (CE) representation, Cubes, Data Discovery, KPIs, Charts, and dashboards
Application and System logs for applications under standard security policy rules. Internal Infrastructure logs also included:
Business governing and planning. Proactive
BAM
Management Servers
App Logs, Server Logs, Internal LBs, and Internal firewalls. Proactive
Forgery detection, Fraud detection, and Money laundering prevention
Security logs for applications and Infrastructure under Elevated Security Policy rules: External firewalls, Telephone system, and Mail system
Business Security
Apps and Infrastructure Security
vigilant

To conclude, we have to again highlight the importance of proactive monitoring in adhering to the 5, 7-10, and 14-15 rules; further, we will show how it can leverage the implementation of Automated Recovery Tools to fulfil rule 1. Now, we have to spend a little time discussing how Oracle contributes to the first and second line of exception handling: rules 6 and 11-13.

OFM Fault Management frameworks

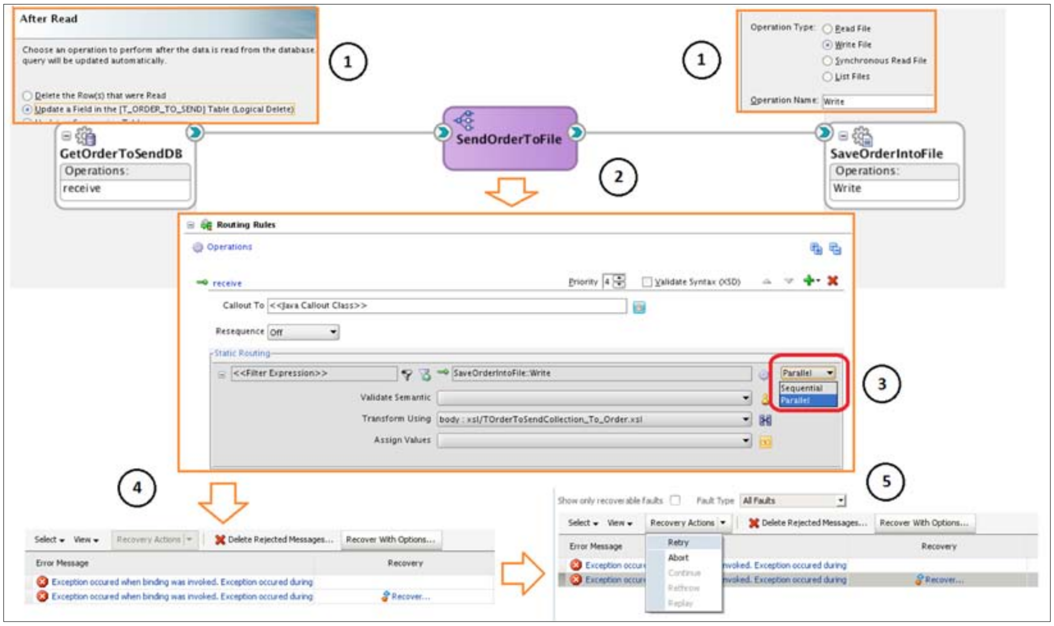
Frameworks, in plural form? Yes, we have more than one in OFM, as we already mentioned. From the components' inheritance, we have similar features that are based on:

- Policy fault handling (the core of Error Hospital): SCA and EBF/ABCS Frameworks
- BPEL fault management (compensative transaction): SCA and EBF/ABCS Frameworks
- OSB error-handling mechanisms: ESB and the EBS Framework

We are not going to spend a lot of time discussing standard OFM EH mechanisms; they have been around since 10g, and since 2008, they have been presented in many books and Oracle ACEs' blogs in great detail (here again, we can refer to Lucas Jellema's handbook). Another, and probably the main reason to make it short, is because of rule 6: Standard OFM EH mechanisms are excellent for services with strictly predefined service roles, preferably, static. In our case (the CTU example), when complex compositions are handled by an agnostic composition controller dynamically, standard tools must become part of a more complex Error Management solution, as we will see further. Now, we will quickly touch upon some of the most common realizations, and if you are familiar with the basics, please proceed to the complex Exception Handler.

Policy-based handling

This classic example involves a Mediator (but it can be BPEL or the whole of SCA), as presented in the following figure. We will use the Oracle illustration to clarify the concept, and we are quite sure that you will find many other examples if you are not already familiar with mediator policies.



After Read

Choose an operation to perform after the data is read from the database query will be updated automatically.

○ Delete the Row(s) that were Read
● Update a Field in the [T_ORDER_TO_SEND] Table (Logical Delete)

Operation Type: ○ Read File ● Write File ○ Synchronous Read File ○ List Files

Operation Name: Write

Priority: 4
□ Validate Syntax (XSD)
Callout To: <<Java Callout Class>>
Resequence: Off
<<Filter Expression>> → SaveOrderIntoFile::Write Parallel
Transform Using: body : xsl/TOrderToSendCollection_To_Order.xsl

□ Show only recoverable faults — Fault Type: All Faults

Error Message	Recovery
Exception occured when binding was invoked. Exception occured during	
Exception occured when binding was invoked. Exception occured during	Recover...

SCA policy-based fault handling

Following are the implementation steps that explain the approach taken in the preceding figure:

1. Create the SCA project used to transfer orders from OrderDB to any file location. You will need two PartnerLinks (adapters): one for reading DB, extracting the Order data, and updating the status field (Logical Delete) and another for FileWriter as presented in the preceding figure. Feel free to implement your own XSDs (for database and file-based orders) and a simple table structure; just a couple of fields will suffice. Set the polling frequency to 10 seconds or any other duration of your choice. Set the statuses for (N)ew, (P)olling and (R)ead Orders accordingly; you will need them for the select criteria.
2. Add a mediator initially with sequential operations and one transformation, for instance, changing the order ID. Use the current `dateTime()` for timestamp elements. Deploy and test the application. Everything should be fine, calm, and simple.

3. Let's emulate the error while operations are sequential (3). Any method will do if you are running it on Unix chmod 444 on the destination file folder; simply assign the OrderID initially defined as `xs:integer` in the message XSD, any string value in DB, and so on. Redeploy and test the application. You will see the dull and simple red-marked status `Failed` on the SOA Dashboard for Recent Instances. Recent Faults and Rejected Messages will inform you that `Exception occurred when....` Go to the Flow Trace and check what the reason was, which you already know. The main point here is that we can't do much about it; we cannot even Retry or perform any other action, and Recovery actions are unavailable.
4. Copy the following two files to the `project` folder (or any other location, including MDS, but then change the references for `fault-bindings.xml` and `fault-policies.xml` in `composite.xml` as explained earlier).

   The following is the first file that contains the Fault Policy XML definition:

```
<?xml version="1.0" encoding="UTF-8" ?>
<faultPolicies xmlns="http://schemas.oracle.com/bpel/faultpolicy">
  <faultPolicy version="2.0.1" id="SendOrderFaults"
               xmlns:env="http://schemas.xmlsoap.org/soap/
envelope/"
               xmlns:xs="http://www.w3.org/2001/XMLSchema"
               xmlns="http://schemas.oracle.com/bpel/faultpolicy"
               xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance">
    <Conditions>
      <faultName>
            <condition>
                    <action ref="ora-human-intervention"/>
           </condition>
      </faultName>
    </Conditions>
    <Actions>
             <Action id="ora-human-intervention">
             <humanIntervention/>
      </Action>
    </Actions>
  </faultPolicy>
</faultPolicies>
```

In the second file, we performed the binding for the policy ID `SendOrderFaults`:

```
<faultPolicyBindings version="2.0.1"
                     xmlns="http://schemas.oracle.com/bpel/
faultpolicy"
                     xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance">
  <component faultPolicy="SendOrderFaults">
            <name>SendOrderToFile</name>
  </component>
</faultPolicyBindings>
```

5. Go to the Mediator and change routing to **Parallel**. Now we have to redeploy and test the application again.
6. Fix the error's cause (chmod 777 or modify message element in Trace Flow). Go to **Enterprise Manager** and **Fault and Rejected Messages** and see that the **Recovery Actions** options are available now. Select **Retry** for the process with the **Recovery Needed** status and confirm your choice. Open the Trace Flow after its completion and see that the problem is as good as gone.

Well, we did a `Retry` operation manually; so much for automation you say! This is because for this action, we selected `ora-human-intervention`. Guess what `ora-retry` would do? You will have plenty of additional parameters to set: Retry count, Retry interval, and what would be the next action if a Retry fails/succeeds (action chaining)?

The `ora-terminate` option is similar to human intervention; it's a simple realization, and no parameters are really needed. We can also rethrow the fault or replay the scope. No doubt, the most powerful option is `ora-java`, which allows you to execute any Java class defined in the related element as shown in the following code:

```
javaAction className="somepackage.someClass"
```

Generally, all that you should do is describe an error's condition and declare the action (reference to action) for this condition. The condition will be tested against the fault code, part of fault message/payload, and so on. See the following examples:

- `$fault/*:reason/text();`
- `$fault/ctx:errorCode/text();`
- `ora:getFaultAsString();`
- `$fault.payload`
- `$fault.code`

The list is not complete; you have great freedom when it comes to selecting error sources in addition to the standard `$fault`. Conditions/faults can be (and actually shall be) grouped into Business, Technical, and so on, with the desired level of details. For instance, refer to the following bullet list:

- The `$fault.code="WSDLReadingError"` error is purely a technical error, but it is in fact related to the remote API availability, that is, the error is related to service edges on North/South. When `$fault.code="3220"`, it indicates the standard ORA-03220 code, which is a problem related to the data quality (we got `NULL` instead of something meaningful). Necessary automated actions can be clearly defined for these situations.
- A fault (raised by your application) that is displayed with `$fault.payload="Client with bad credit history"` is a business fault (not really a fault, but the condition is still critical). Although the temptation to put this kind of fault on the `ora-human-intervention` resolution is high, it is better if you devise an automated solution; it's not that hard.

The last step is to associate all policies with the SOA composite application or individual component (BPEL or Mediator as in the *Policy-based handling* section) using fault binding. It would be either `<composite faultPolicy=…>` or `<component faultPolicy=…>`, where you list all your individual policies, associated with the component/composite name.

In addition to this, you have great flexibility for declaring `<Properties>` in the property set within your policies in order to support your complex actions; please see the syntax in the Oracle documentation.

Oh! the possibilities. They are truly boundless. Honestly, the OFM policy-based Exception Handling Framework is the second best thing in OFM after BPEL; this mechanism is extremely powerful. You can employ any category of Faults (embedded, default, or declared), assess (test) faults as you want, group them under any types (technical or functional), create new or employ existing actions for resolutions, chain actions within any fault, associate the actions `returnValue` parameter with any other following action (another form of action chaining), declare property sets for your actions (highly useful for Java Actions, but for others as well), and finally, centralize your policies using bindings. Applied to Mediator as in the earlier example, which generally acts as a mini-ESB within SCA; you will have an excellent realization of Policy Centralization. What else can you wish for?

Please look closely at rules 6, 9, and especially 5 when it comes to Metadata Centralization. Before building complex policies for agnostic controllers, you must be 110 percent sure that all possible scenarios are accounted for, all composition members and their roles are identified, and all appropriate resolutions are detected:

1. Even in static composition controllers, due to BPEL's easy-to-implement ways of services' invocation, you could have long chaining of synchronous and asynchronous task-orchestrated services. Some of them can have parallel flows, spawning subsequent parallel flows; some sync BPELs can call async BPELs comprising JMS adapters with triggered dehydrations causing callbacks in separate threads so that you lose your request-response correlations; third-party components can propagate faults incorrectly; and so on and so forth. In dynamic composition controllers, without understanding the initial situation, your analysis will be even worse. So, quite soon all your Policy Centralization will be centralized around a single `ora-human-intervention`, as it is the only option to keep control.
2. The simplicity of default fault actions can be deceptive. Regarding the preceding point, we could have different use cases for seemingly identical `throw-versus-reply` and `reply-with-fault`. Please see the Oracle documentation for more details.
3. The endless policy-based exception handling possibilities multiplied on endless Java capabilities of the `ora-java` action can lead to disaster if applied uncontrollably. A Java guru can decide to put all the handling scenarios into Java action. Quite soon, business logic will silently sneak into this handler as well, and why not? It's fast, simple, and understandable for all Java coders. Do we need to explain where the catch is? (Refer back to *Chapter 1*, *SOA Ecosystem – Interconnected Principles, Patterns, and Frameworks*; check all the principles and the initial CTU's SOA status.)
4. The policies are preloaded on startup for best performance. That's surely a decent thing to do. Therefore, we have two concerns here: naturally, your policy size matters (and not always positively), and secondly, you have to restart/redeploy your application every time you update your policies. This is another reason to perform the service-error mapping exercise we started with results presented in an Excel spreadsheet.

Yet again, OFM fault policies are brilliant and the concerns expressed in the preceding points should not prevent you from using them for static compositions. Just follow the proposed rules and generic SOA principles/patterns.

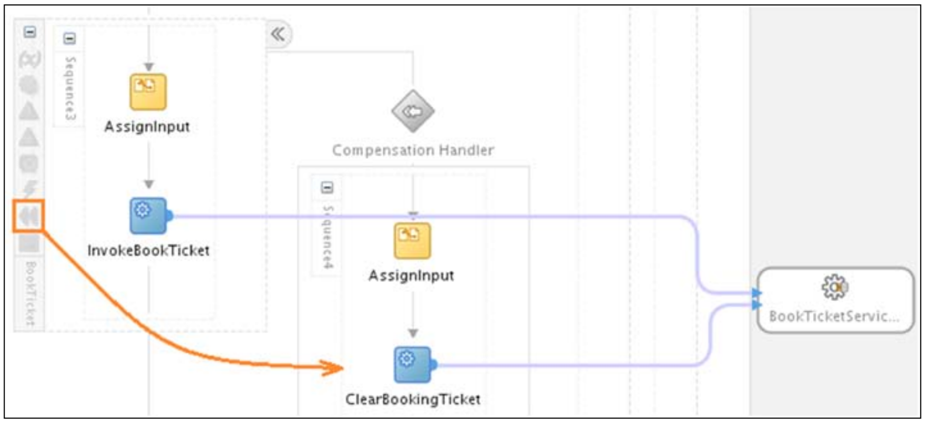
Compensative transactions

The compensative transaction feature is proprietary for long running transactions, and thus, it's incorporated into BPEL. Like with most Oracle BPEL features, it is presented graphically; thanks to this, it's quite well known and widely used. Probably, we should not waste your time here by explaining the obvious, but as long as the Compensative Transaction SOA pattern is involved, a brief walkthrough should be in order.

The classic example (and you can find plenty of them) is based on SCA's booking application: while planning your vacation, you consequently book a hotel, a flight, and a car. If a car is not an option, you still have no reason to cancel your vacation, flight, and hotel reservation (the error resolution action would be Continue). Now, if there are no rooms available (you should start earlier this year) and living in a tent hardly makes your spouse happy, you definitely have to cancel your flight and consider another destination (we hope that you can).

So what do we know? Refer to the following bullet points:

- We are aware of the blue double-arrow icon (like the one on your remote control) on the left-hand side of the BPEL scope. It is second from the bottom, named **Add Compensation Handler**.
- We know that we are not in the same atomic transaction: ACID rollback is not possible.
- As rollback is not possible, physical delete (as opposite to insert, the so-called business rollback) will be implemented, probably as a separate BPEL scope or additional process call: invoke the "ClearBooking". Thus, some more effort is required from us. Our Entity services must have the full set of basic operations: `add<..>`, `get<..>`, `delete<..>`.



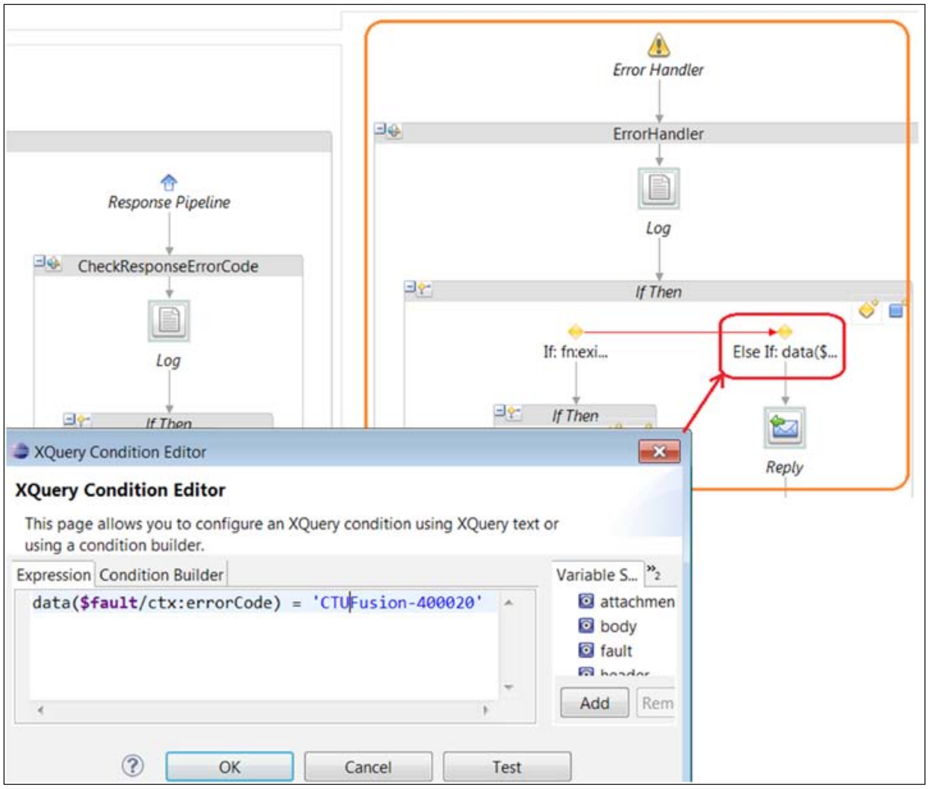
In the preceding figure, we performed compensation to scope the `BookTicket` element, that is, `<compensate scope=" BookTicket"/>`, in the booking task service. This service is a static composition controller with three scopes dedicated to three different bookings explained in the previous points. As you understand, to the modularity, every type of booking is a separate service (could be BPEL). You can keep `insert` and `delete` operations for every booking type in one service or present them as different BPEL processes if you want to maintain them in a single SCA. In this case, you can connect them to the master controller using Mediators (`HotelReservation`, `BookTicket`, and `CarRent`) and apply (optional) Policies to Mediators as well, as in the previous example. Thus, in the preceding figure, we connected to the `BookTicketService` mediator, WSDL, for reservation and cancellation (from the compensation sequence). Do not forget to extend your master controller with the `Catch(All)` exception handler for technical errors provided by composition members, and we are done.

The realization of the Compensative Transaction SOA pattern by the Ora BPEL compensation handlers is elegant and intuitive, and you can always rely on the business rollback feature in your static composition controllers.

Exception handling in OSB

Now, we have to look at the standard handlers available on service edges, usually maintained on OSB as Business/Proxy services. The fault handler is one of the three pillars (together with request and response) on which OSB stands. For handling any abnormal situation, you have the same activities as you have in happy flows (refer to the following figure). Thus, everything we said about the mapping exercise during the error analysis phase is true here as well.

After catching the error, you can analyze the content of the fault message (the same `$fault/ctx:errorCode` or any other part of it) and act accordingly; route to the predefined destination, update the message content (for instance, Message Tracking Data), and perform the service callout to EJB or another service.



XQuery Condition Editor

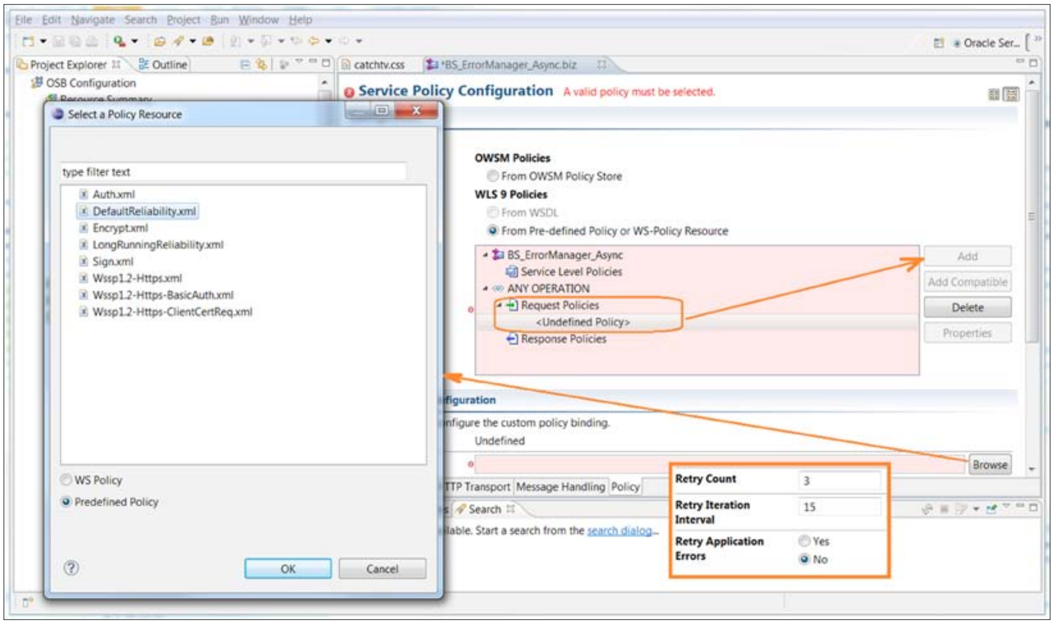
This page allows you to configure an XQuery condition using XQuery text or using a condition builder.

```
data($fault/ctx:errorCode) = 'CTUFusion-400020'
```

As we are discussing the cooperation of service edges (the composition member's endpoints) and composition controller, we do not have much freedom here at OSB, especially at the South end (see the figure from the *Maintaining Exception Discoverability* section again). We should be very careful with individual handlers, as they can be quite off from the entire composition logic; technically, we should be concerned about three main things:

- Log the caught error (the first step in the preceding figure) properly.
- Update the message tracking information in the XML container to help the service broker with error resolution. We can even perform the conversion of the error code if necessary, but this will usually require SR lookup.

- Depending on the service location and use case, we can try basic recovery operations such as **Retry** (see the following figure) on Business Service, which is under the **Transport** tab. Despite its simplicity, the **Retry** operation must be applied with extra care, after the analysis of fault types. For instance, we can double the transaction if the service in not idempotent and our timeout interval is not adjusted according to the processing time.



Retry Count: 3
Retry Iteration Interval: 15
Retry Application Errors: ○ Yes ● No

Maintaining service policies on OSB

As the name service edge denotes, here we have to apply a lot of message handling and security policies as well. Not all of them are directly EH-related, but all of them contribute to the composition's resiliency. Another thing that is apparent from the preceding figure is that they are different policies from what we discussed while explaining SCA EH. Well, complete Policy Centralization is not really achievable in OFM, and we have already mentioned that; however, we are quite capable to maintain the desired level of centralization per SOA framework. At least two policy types are essential for the OSB implementation: WSM and standard (almost) WS-Policies. You can see a list of the available default policy resources in the preceding screenshot; they are applicable for Business and Proxy services (see the **Policy** tab).

Two WS-Policies are contributing a lot to EH in general: WS-Addressing and WS-ReliableMessaging. Despite all the differences with SCA's fault-handling policies, the mechanism of binding them to the WSDL has a lot of similarities. A discussion on practical WS-Policy implementation is a bit out of the scope of this chapter, but you could find good explanations in *Web Service Contract Design and Versioning for SOA*, *Thomas Erl*, *Prentice Hall* (*Chapters 16 and 17*) in addition to the Oracle OSB documentation.

Complex exception handling

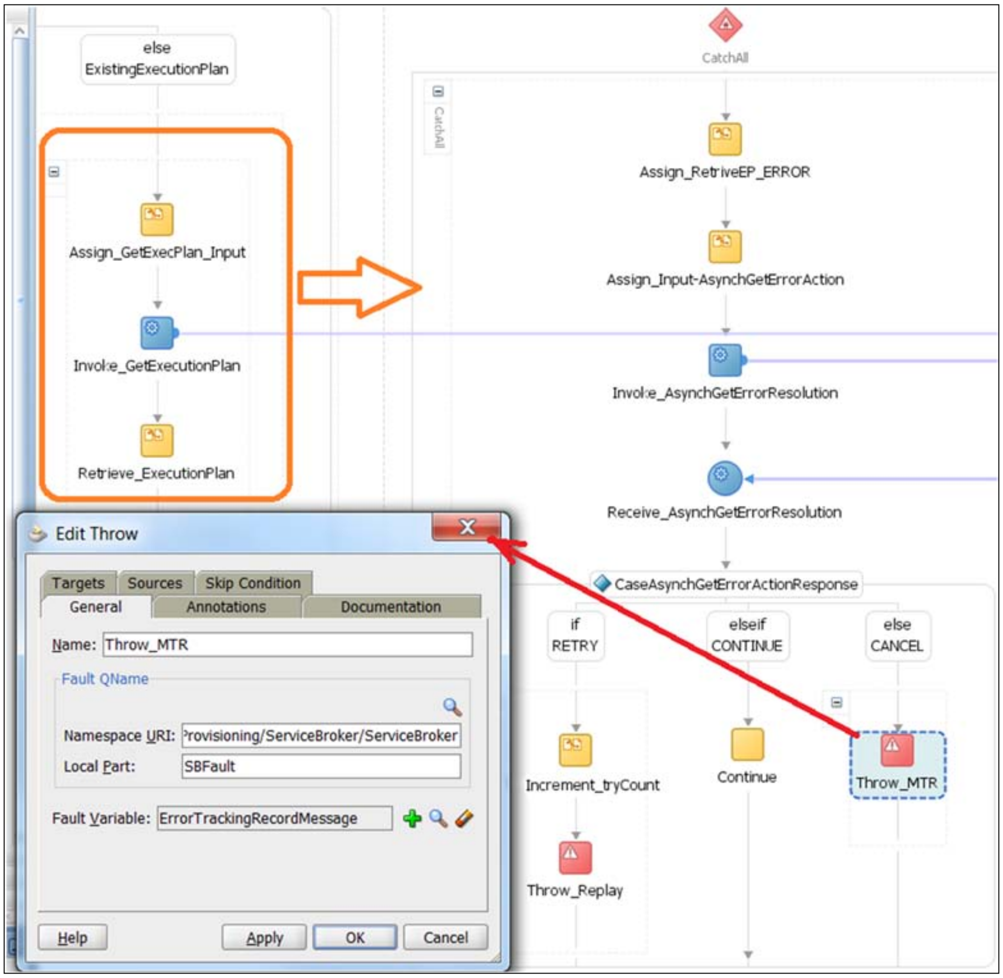
Before we discuss the essential steps in exception handling in an agnostic composition controller implemented in the CTU SOA farm, we will mention the importance of clear and consistent identification of all fault messages related to certain process instances. You could use a standard Ora ECID for this purpose in addition to the initial Java-based labeling process instance at the beginning of every BPEL:

1. Go to the **Receive activity** tab at the top and select **Edit** after a right-click.
2. In the **Properties** tab, click on the green cross and select **tracking.ecid** from the drop-down list. Assign it to the variable of your choice. Use it within your Message Tracking Records Object.

We will start with recalling the structure of the composition controller (async Service Broker). Basically, it has two parts (if we omit the standard initialization): acquiring the execution plan and looping through EPs elements. Thus, we should have three exception scopes: sequentially, one for master loop and one for the EP endpoint, and one generic outer handler for the entire process. (In your version, you can implement additional handlers for every new scope.) For all cases, you can declare a generic `SBFault` and use it with fault variables based on the Message Tracking Record type (see the declaration in the following figure).

When the entire process fails, we can employ standard catches: one for `SBFault` and the master, `CatchAll`; however, in any of these handling sequences, we cannot really fix the problem. All that we have to do is identify at what nested level the error occurred (master or subcontroller; this should be visible in message tracing records) and assign status code in the Message Header and MTR objects. After this, we perform the Audit (depending on the Audit level of the current invocation) and return to the composition initiator (client, using Invoke).

When the extraction of the execution plan fails, we should invoke `ErrorManager` for the first time; see the following figure:



Name: Throw_MTR

Namespace URI: ?rovisioning/ServiceBroker/ServiceBroker

Local Part: SBFault

Fault Variable: ErrorTrackingRecordMessage

The number of actions required is still limited as we are not in EP's traversing mode yet. If SB is acting as a master composition controller, actions such as `Retry` and `Cancel` (after several retries) would be appropriate. If we are in a subcontroller, then the Continue resolution is quite acceptable, depending on the business logic.

Prevention measures are more important here than just getting the possible resolutions; we should have redundant EP storage implementation as we mentioned it in the rules: DB- and File-based. `ExecutionLookupService` should encapsulate these two approaches.

Finally, the EP execution could fail because of error(s) occurred during the execution in the main tasks loop. This is the place where the first table in the chapter will be exceedingly handy. Most importantly, not only primitive suggestions will be returned, but also the entire Message payload after the error-compensation process, instantiated by ErM. Thus, for keeping SB truly generic, two main blocks must be implemented for the main loop scope:

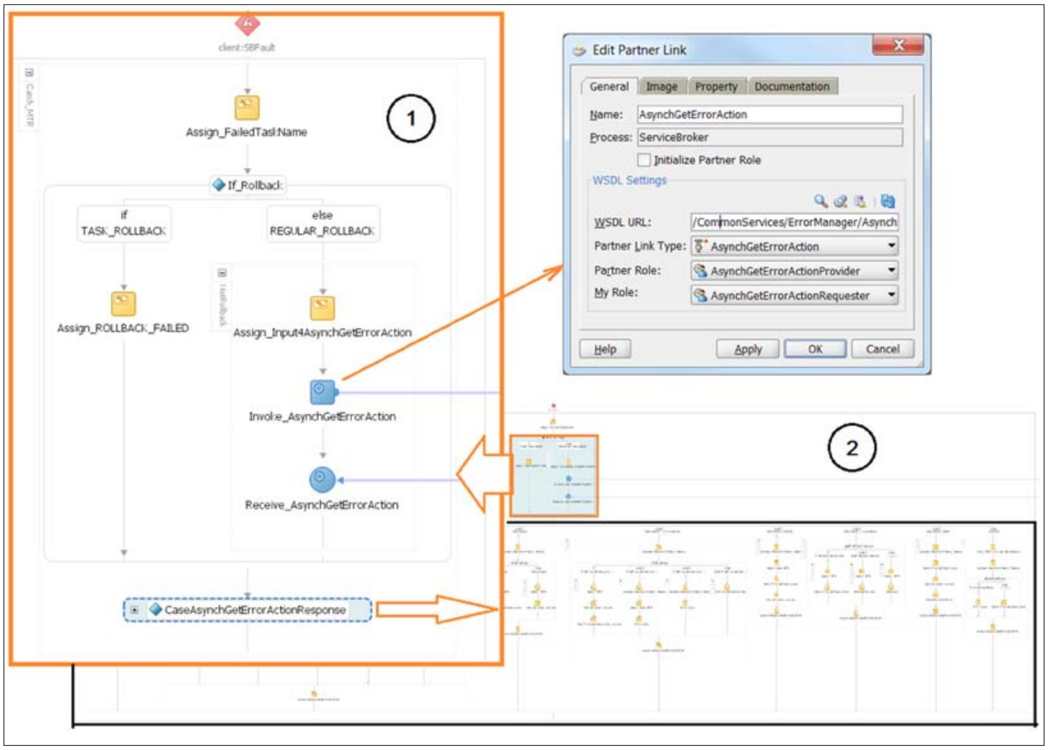
- The invocation of ErM with the entire payload and descriptive `SBFault` from the Catch handler (see the next figure, part 1)
- Updating the MTR object and processing flags in the message header after receiving the response and updating the payload (see the next figure, part 2)

Providing a list of possible combinations in the second part can be extremely long, as you can see from the following figure; therefore, please consider only some of the logical outcomes:

- When the `RETRY` resolution is returned, we should:
    - Increment the loop counter
    - Check the new payload received from EH and update the current task's payload accordingly
- When `CONTINUE` is suggested, we should check the following:
    - Are we continuing with `ROLLBACK` (this is an agnostic composition controller used everywhere) or a regular task?
    - Is it the last task in a loop, and do we have to summarize all executions (such as calculate orders and grand totals)?
- When `CANCEL` is received:
    - Stop execution

Certainly, we should have other resolution options such has `ROLLBACK`, `ROLLBACK_FAILED`, and `ROLLBACK_DONE`. If we are in the first task in the loop, then we probably do not need to perform any rollback. One really important thing to understand here is that we cannot put *any* error-specific execution logic into our agnostic composition controller; it will just break the whole idea. This big *case* logic in the second part of the following figure is only setting the flags and MTR/MH assignments.

Only the invocation of services with standard contracts through the Adapter Factory is allowed here. Resolution and execution logic is completely abstracted and centralized in Enterprise Repository and provided in the form of XML execution plans by the Service Inventory Endpoint. It is implemented as a database with a friendly interface that will allow you to apply new execution policies without SCA redeployment. Proper testing and other governance procedures will be applied later.

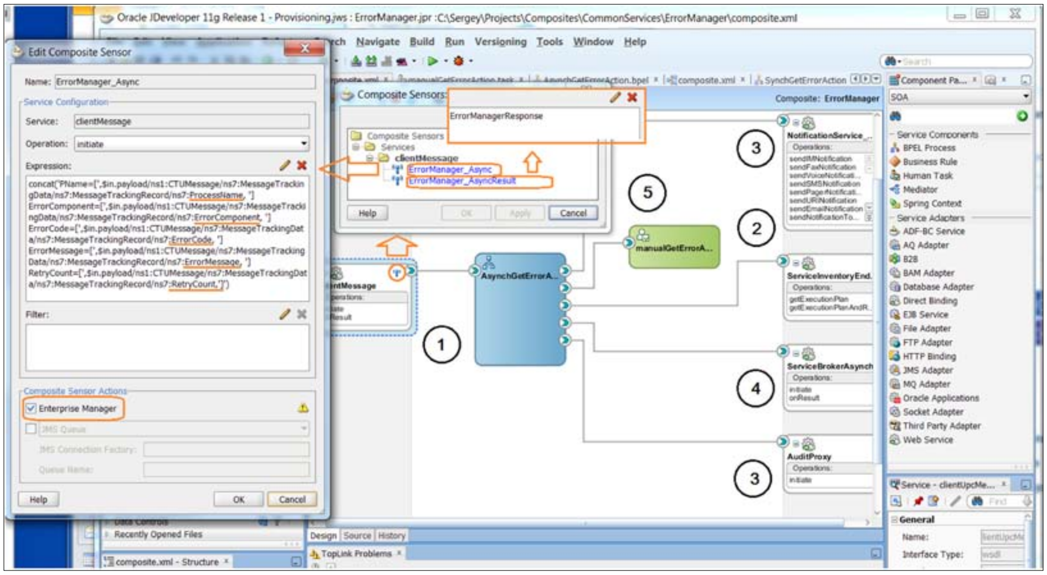


Name: AsynchGetErrorAction
Process: ServiceBroker
□ Initialize Partner Role
WSDL Settings
WSDL URL: /CommonServices/ErrorManager/Asynch
Partner Link Type: AsynchGetErrorAction
Partner Role: AsynchGetErrorActionProvider
My Role: AsynchGetErrorActionRequester

Fault handling logic in Agnostic Composition Controller

If SB fault-catch scenarios are clear, we can now look into the internal Error Manager architecture, presented as SCA (refer to the next figure). We have five main blocks completely aligned with our generic requirements that are expressed in the first paragraph of this chapter:

1. We employ Oracle's **Composite Sensors** to monitor ErM's incoming and outgoing messages. This information will be available for search and analysis of the Instances page of the SB SOA composition application in the Oracle Enterprise Manager Fusion Middleware Control Console. From the following figure, you can see what elements we decide to concatenate in the Sensor's expression. Bear in mind that all these elements must be parts of the payload. Thus, as mentioned earlier, ECID could actually be part of the `ProcessName` element. Assigning a Sensor for an outgoing message is much simpler: it's an ErM Response. Here, we are not using Mediator as the central component of the handler; all inbound messages are going to the BPEL process, which will help us with dispatching faults to other components, thereby fulfilling generic requirements.
2. For error code conversions, a resolution action's lookup, and the extraction of compensation workflows, we have to call **ServiceInventoryEndpoint**. If the first task is optional, as it can be handled by **Domain Value Maps** (**DVM**) in SCA Mediators, the second and third are the core of Error Manager.
3. It would be prudent to notify Ops or other involved parties as part of the resolution action. This part is implemented as **NotificationService**, employing the whole bunch of Oracle communication adapters.
4. When the resolution action is identified, it will be passed back to the caller. If the resolution is complex and requires a new instance of Service Broker (as a compensative EP), then we assign the extracted EP to the new Process Header's container and invoke an async SB.
5. Our last resort is the manual resolution that is used when automation is not possible, number of retries has exceeded, or we get a critical error during rollback.



Edit Composite Sensor

Name: ErrorManager_Async

Service Configuration

Service: clientMessage

Operation: initiate

Expression:

```
concat('PName=[',$in.payload/ns1:CTUMessage/ns7:MessageTrackin
gData/ns7:MessageTrackingRecord/ns7:ProcessName, ']
ErrorComponent=[',$in.payload/ns1:CTUMessage/ns7:MessageTracki
ngData/ns7:MessageTrackingRecord/ns7:ErrorComponent, ']
ErrorCode=[',$in.payload/ns1:CTUMessage/ns7:MessageTrackingDat
a/ns7:MessageTrackingRecord/ns7:ErrorCode, ']
ErrorMessage=[',$in.payload/ns1:CTUMessage/ns7:MessageTracking
Data/ns7:MessageTrackingRecord/ns7:ErrorMessage, ']
RetryCount=[',$in.payload/ns1:CTUMessage/ns7:MessageTrackingDat
a/ns7:MessageTrackingRecord/ns7:RetryCount,']')
```

Filter:

Composite Sensor Actions

☑ Enterprise Manager

□ JMS Queue

Complete Error Manager SCA for Agnostic Composition Controller

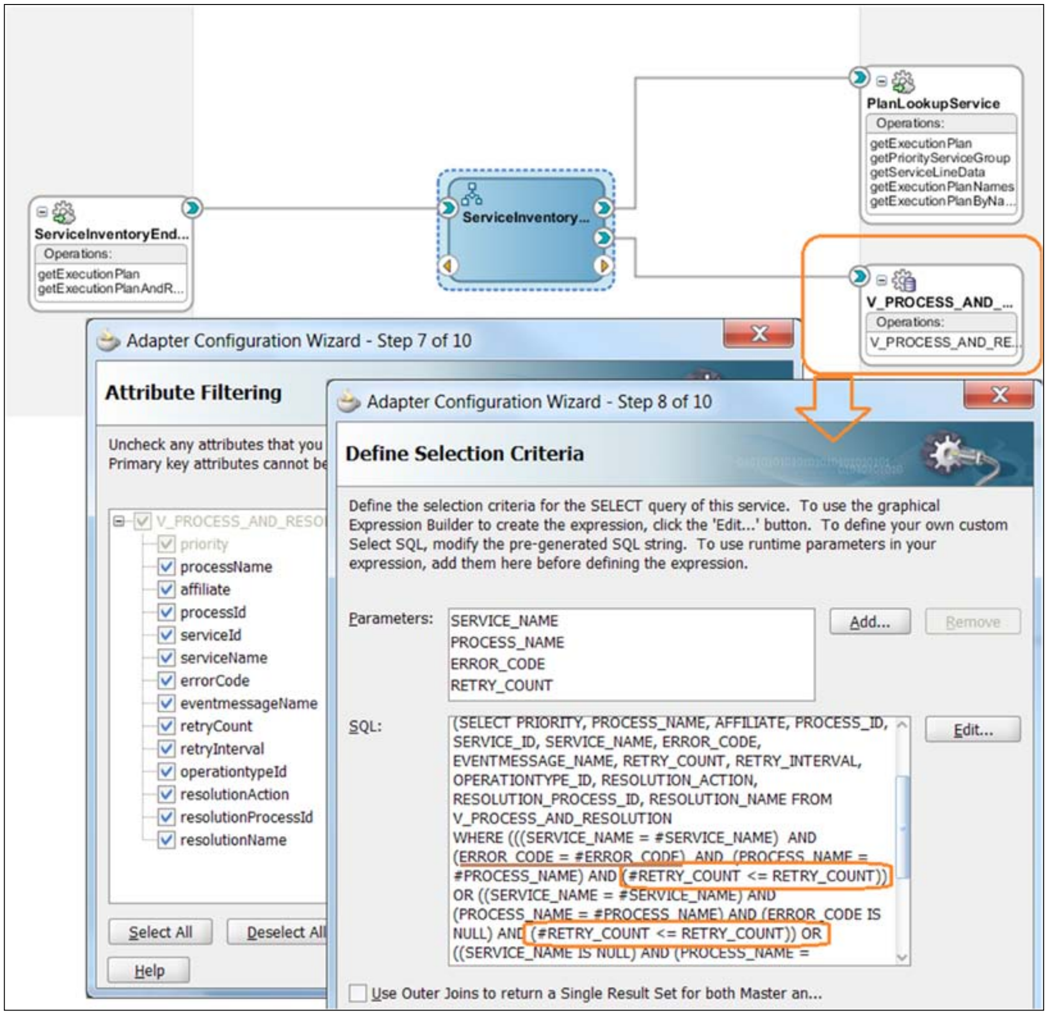
Sensors are another nice feature in OFM SCAs, and we encourage you to use them actively, although with some limitations applied (such as the payload as the only source); please see the Oracle documentation. Thus, we can go directly to the main feature here: **ServiceInventoryEndpoint** (see the next figure).

In the following figure, you can see the developer's version of this service, presenting a typical versioning strategy around Oracle SCAs. We do this on purpose, as you may remember. Initially, when we started with the Service Broker implementation (*Chapter 3*, *Building the Core – Enterprise Business Flows* and *Chapter 4*, *From Traditional Integration to Composition – Enterprise Business Services*) and relied only on standard Oracle fault handlers, we implemented `ExecutionPlanLookupService` for happy paths only. Now we have to consolidate the Service Repository DB (from *Chapter 5*, *Maintaining the Core – the Service Repository*) under a unified endpoint for consolidated EP and the extraction of resolution actions'. Ideally, this should be that one entity service with all the required operations (Java is a good choice for this type, and if functional decomposition is required, we will apply it later); for now, we can implement the BPEL process with an additional DB adapter, wrapping `ExecutionPlanLookupService`.

For this DB adapter, the set of parameters provided by EH and MTR is employed in the simple `SELECT`, as presented in the next figure. Needless to say, ESR DB records are completely based on the first table in this chapter.

For better resiliency, we should add another adapter for files to extract execution plans and resolution actions stored in XML FSO. With the RAC DB and NAS-based FSO in a clustered environment, this type of realization will be truly bulletproof.

We leave it to you to count how many SOA patterns we covered in the preceding paragraph. Surely, the presented implementation is not really suitable for production, and we offered it only for demonstrating the capabilities of Oracle SCA.



☑ V_PROCESS_AND_RESO
- ☑ priority
- ☑ processName
- ☑ affiliate
- ☑ processId
- ☑ serviceId
- ☑ serviceName
- ☑ errorCode
- ☑ eventmessageName
- ☑ retryCount
- ☑ retryInterval
- ☑ operationtypeId
- ☑ resolutionAction
- ☑ resolutionProcessId
- ☑ resolutionName

Define the selection criteria for the SELECT query of this service. To use the graphical Expression Builder to create the expression, click the 'Edit...' button. To define your own custom Select SQL, modify the pre-generated SQL string. To use runtime parameters in your expression, add them here before defining the expression.

Parameters:
SERVICE_NAME
PROCESS_NAME
ERROR_CODE
RETRY_COUNT

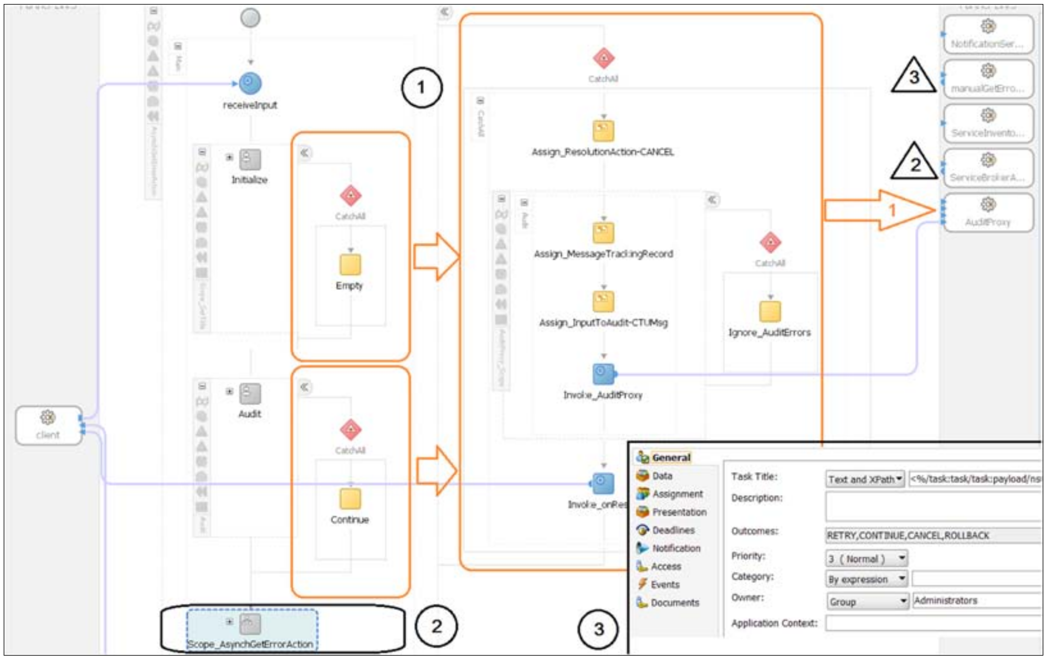
SQL:

```
(SELECT PRIORITY, PROCESS_NAME, AFFILIATE, PROCESS_ID,
SERVICE_ID, SERVICE_NAME, ERROR_CODE,
EVENTMESSAGE_NAME, RETRY_COUNT, RETRY_INTERVAL,
OPERATIONTYPE_ID, RESOLUTION_ACTION,
RESOLUTION_PROCESS_ID, RESOLUTION_NAME FROM
V_PROCESS_AND_RESOLUTION
WHERE (((SERVICE_NAME = #SERVICE_NAME)  AND
(ERROR_CODE = #ERROR_CODE)  AND  (PROCESS_NAME =
#PROCESS_NAME) AND (#RETRY_COUNT <= RETRY_COUNT))
OR ((SERVICE_NAME = #SERVICE_NAME) AND
(PROCESS_NAME = #PROCESS_NAME) AND (ERROR_CODE IS
NULL) AND (#RETRY_COUNT <= RETRY_COUNT)) OR
((SERVICE_NAME IS NULL) AND (PROCESS_NAME =
```

□ Use Outer Joins to return a Single Result Set for both Master an...

Now, we should look inside the main ErM dispatcher, the BPEL process that will handle different recovery scenarios (see the next screenshot). From the error handling perspective, it can also be divided into three main areas where the faults will be accounted for: request parameter initialization, a call to Audit service, and the **ServiceInventoryEndpoint** invocation (see the next figure, part 2). This entire process will also be covered; all that we can do here is perform proper logging. The whole design will be kept as simple as possible for better resiliency, and we must assume that any disaster happened during compensative actions can be fixed either manually or by ART, which is implemented externally (rule 15, Proactive Automated Error Management).

Manual Recovery (see the next figure, part 3) is a typical SCA Human Task Service component with standard outcomes. All configuration parameters around it are pretty basic, and we will not focus on it here. Again, in the Oracle documentation, it is explained very clearly.



Task Title: Text and XPath <%/task:task/task:payload/ns

Description:

Outcomes: RETRY,CONTINUE,CANCEL,ROLLBACK

Priority: 3 (Normal)

Category: By expression

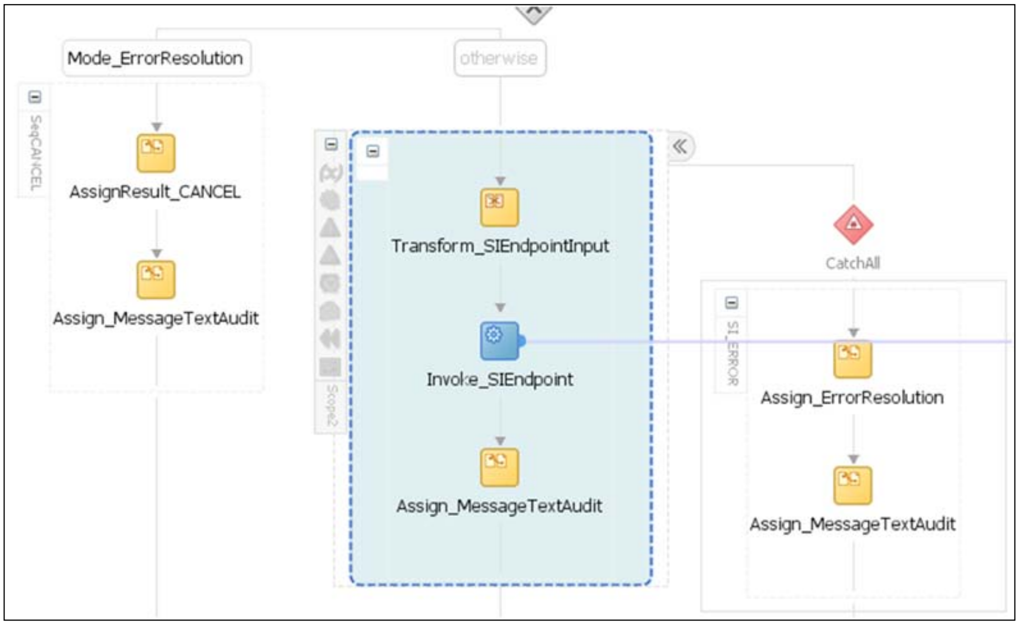
Owner: Group Administrators

Application Context:

Fault handling scenarios in Error Manager

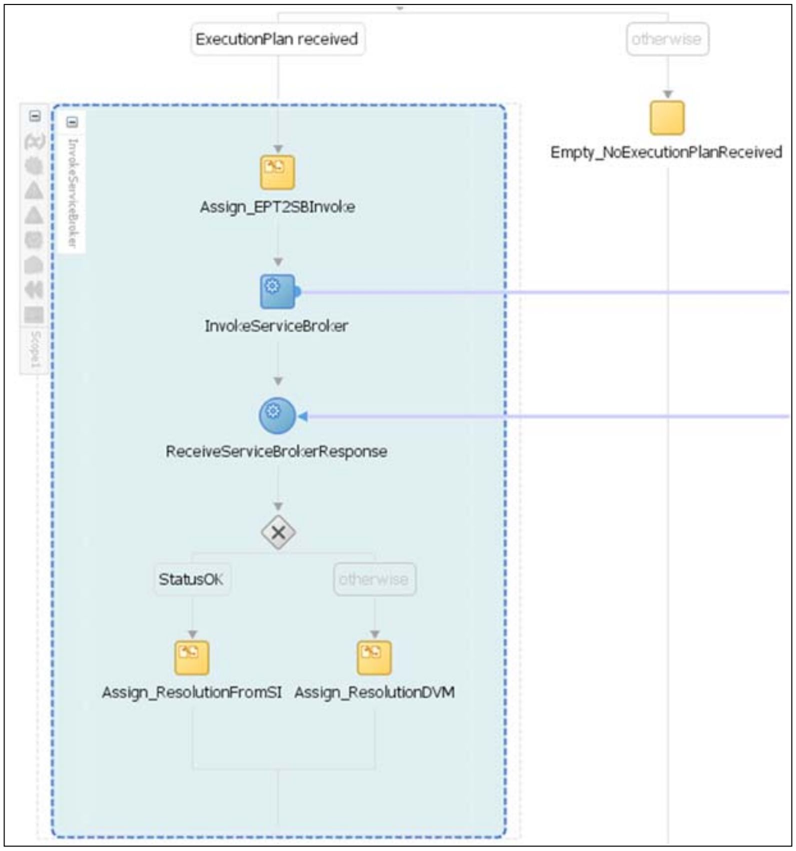
The last two scopes of this BPEL dispatcher within ErM that we should mention here are as follows:

1. The first is the **ServiceInventoryEndpoint** invocation scope (see the next figure). This is where we actually invoke the service explained in the previous figure.



2. With a positive outcome from the previous step, we invoke Service Broker to execute compensative transactions. (See the next figure.)

The compensation outcome (or resolution action if the case is simple) with modified payload is returned to the master controller. In some complex cases, we can put a lot into compensation EP(s) in order to return to a consistent state, and all that the master controller needs to do is finalize its activities, that is, exit gracefully. Adapter poller will start with another complex composition in time, based on the new (or as good as new) data's state.



This is it! We have shortly demonstrated how Oracle's standard fault-handling tools can be employed for fixing errors in a static composition (and not only policy-based framework is really powerful for all cases) and how we can reuse an agnostic composition controller to handle errors in dynamic compositions.

Finally, we have to look at how we can use external solutions for proactive monitoring and automated recovery.

Automated recovery concepts

Combining the design rules and patterns from *Chapter 3*, *Building the Core – Enterprise Business Flows*, *Chapter 4*, *From Traditional Integration to Composition – Enterprise Business Services*, and *Chapter 5*, *Maintaining the Core – the Service Repository*, and from this one, we can assume the following:

1. The Service Repository with all the service metadata elements is properly maintained. We have clear associations between services, endpoints, and Audit messages under the roof of processes. Simply put, every process has a distinct service invocations footprint. Technically, it can be presented as a master `tModel` for a task-orchestrated service if we look at the UDDI analogy.

> The preceding assumption is too bold. In dynamic compositions, the sequence of service invocations and even the number of invocations (that is, footprint) can be different for the same abstractly defined process. The key here is to watch out for the services with specific service roles, sometimes ignoring low-level composition members.

2. Auditing a sequence is strictly observed, which means that we can always find records in logs according to the declared design rules. Simply put, we can always reconstruct the process path from the logs in the exact way as we see it in the Oracle SCA Enterprise Management Console (instances).
3. Every service (Entity or Task) is clearly documented in ESR in terms of execution policy, performance, and consumed resources. Generally, from practical tests and business requirements, we should know that an end-to-end time of 700 ms for this service is acceptable, but 1200 ms is not.
4. Our unified composition descriptors (EPs and routing slips) are redundantly implemented by ESR and FSO/NAS. They are critical for any type of composition controllers (dynamic or static), and redundant implementation of this part in not expensive if we decide to keep the XML EP definitions as files in parallel with DB storage (which is SAN/RAC-based).
5. In *Chapter 4*, *From Traditional Integration to Composition–Enterprise Business Services*, in the *A simplified Message Broker implementation* section, we learned that the design and deployment of a simple service broker for most of our compositions (and definitely for all compensative compositions as well) is absolutely achievable outside of OFM.

6. Our log monitoring tools (Oracle BAM, Nagios, and so on) can control individual service runtime metrics and task service footprints in general. BAM connectors are explained in the next chapter.
7. Obviously, we have our original message logged (orange step 1; refer to the figure from the *Maintaining Exception Discoverability* section), as well as optionally, the message with header and tracing records at the moment our composite application crashes.

> Why is the crash record with message payload optional? Because we simply cannot count on it. If we can, then the first and second lines of defense discussed previously will be more than enough. However, how many times in complex compositions do messages just disappear without trace? At best, you can only have a record indicating that a response was sent from the composition member that never reached the destination. This is exactly the situation we are discussing here, and our primary goal is to keep the business running and buy some time for Ops to find the root cause.

Regarding the last position, proactive monitoring and fault prevention generally means that you collect and analyze technical and business data for an extended period of time and analyze them against different thresholds, which are specific to individual processes at the time of execution. You already have a comprehensive list of WLS/SOA MBean attributes to monitor. After completing your homework (sorry, we do not have enough space here to explain the meaning of every attribute), you will learn that, for instance, Hogging Thread Count indicates we have some threads that take too much time; we can assume that they will never be returned (send an alert when you have more than 10 of these). What should you do? Increase the thread pool, maybe? Yes, it might help, but only a little and for a short period of time. If you start getting this error after deploying a new composite, most probably we will have an indication stating that it is poorly designed and not misconfigured.

Therefore, technical monitoring must be combined with functional monitoring to select a proper action; bare minimum policies should be as follows:

- Every abstract process must be monitored according to the SLA of the total execution time. That's it! From start to stop (from the composition's initiation until its delivery to the ultimate receiver), we must have two minutes (set your number here), not more. Simple isn't it? Yes, if you assure that start/stop indications are clearly provided to Audit (Canonical Expressions in logging) and that the execution time can vary, depending on business hours. Let your developers wonder about record subjects and log consolidation, and data cleansing will not be so simple.

- Monitor the process footprint as a collection of individual invocations according to the composition member's records in Service Repository. Obviously, there should be no misses in the process log, and all entries and exits of individual services must be according to the SLA.
- The last and obvious thing is that we should have no *Error* or *Exception* clauses in the process log.

Frankly, if all conditions for technical and functional logging are met, all business compositions with related compensation logic are recorded in SR. The standard Oracle BAM will perform monitoring gracefully, so triggering the recovery action is just a matter of passing the initial message to any composition controller, with a new composition plan as a parameter (this can be a dynamic EP, static BPEL/SCA, standalone PL/SQL, or a Java function—whatever you can fire using IoC or NDS).

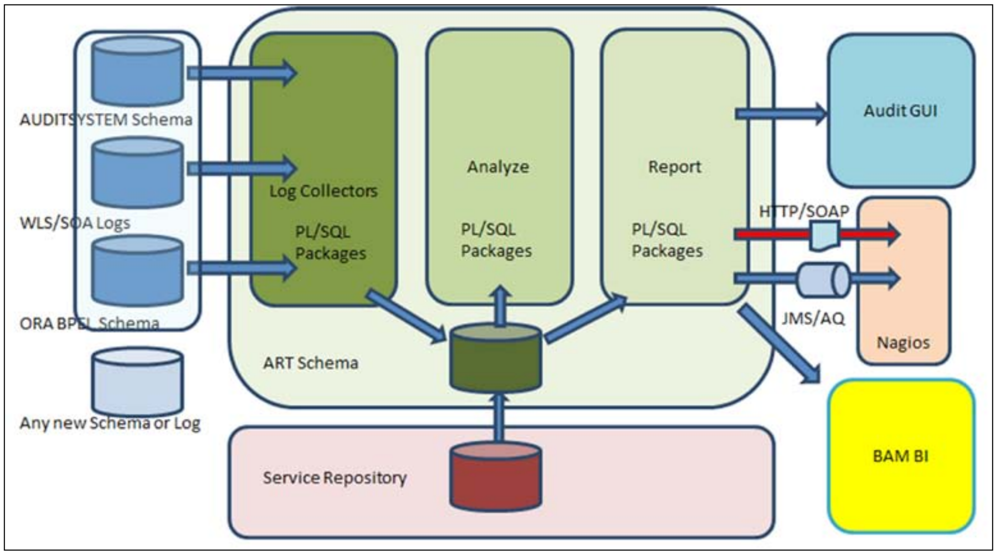
Some more side notes. Why should we consider an external simple Service Broker (or message broker) for ART if we can follow the Redundant Implementation SOA pattern for the entire OFM stack?

Well, it is always interesting to see how top management's opinion changes about absolute necessity for the business to have HA with 99.95% availability (4,38 hours downtime per year), when you explain in figures that moving to this presents from 95% (18 days) will double the investments into infrastructure (and quite often more than double). Most importantly, the Redundant Implementation pattern prevents you from HW malfunctions, not from OFM misconfigurations (one of the top OWASP risks from the previous chapter), poorly designed service, or a single illogical business rule.

Combining all the preceding factors, the basis for ART can be presented as a simplified block diagram (refer to the next figure) with three generic parts:

- Log Collectors (deep green) are part of the adapter framework with individual adapters for your custom logs, WSL/SOA Infra logs, and BPEL schema in particular. With every new source added, you have to implement a new adapter. Most of these adapters already exist in OFM, so the trick here is to consolidate your own formats that exist in OFM. This is not always straightforward; BPEL logs have two records (entry and exit) for SCA components, where you could have one record with two related columns in your logs.

- The granularity of the records depends on the Trace/Audit level you have in your system. The completeness of your data is the key factor to lend precision to your analysis. A bit of data cleansing is sometimes required before analysis; with this, you update and enhance incomplete records such as assuming a component's exit time as the entry time of the next component in the invocation chain. The complexity of this process can be quite high, depending on the complexity of your compositions. A rule engine (custom, Oracle, or any other) is employed for making decisions based on functional policies and SLA metrics that are stored in ESR.
- The final results are delivered into the last component, usually a Report schema with a clean log that contains conclusions and resolution suggestions. This data is consumed by any dashboards, consoles, and reporting tools, both standard and custom. This is also the source for a dedicated process that will be responsible for performing complex recovery actions and compensations. The simple implementation can be the Adapter (on SCA, the BPEL or Mediator will do) checking for the resolution flag and the name of the process/composition to execute. A Java process reading AQ will increase the resiliency as it will comply with design rules 9 and 15. Needless to say that most of the operations in the technical and functional monitoring described in the preceding bullet list shall be performed on the DB side to make it more simple, fast, and resilient (less components involved); Oracle DB works perfectly well with AQs and is the top choice for BAM and Oracle BI.



Summary

We would like to remind you about the statement we made at the beginning of *Chapter 3*, *Building the Core – Enterprise Business Flows*. Yes, OSB (the ESB SOA composite pattern from Oracle) is probably the only pattern you need in your SOA infrastructure. Maybe your business does not need long running asynchronous processes; it also could be that your security requirements are soft, and two core security policies implemented in OSB by default will suffice. A number of service artifacts can be insignificant, and you can survive without Central Repository; however, if your business applications do not need proper exception handling, please check the pulse of your business—maybe it's already dead!

Whatever framework is described in *Chapter 1*, *SOA Ecosystem – Interconnected Principles, Patterns, and Frameworks*, will be responsible for your core business operations; exception handling will be an essential part of it. If you, as an enterprise SOA architect (and we assume that you are), are involved in the resolution of critical situations around apps that are delivered by your team, then carefully designed logging and EH will save you from a lot of trouble (and not only technical). Make it proactive and preventive using the patterns and rules explained in this chapter and most of these situations will be avoided.

This chapter concludes the CTU architectural and technical example started in *Chapter 3*, *Building the Core – Enterprise Business Flows*. In the next and last chapter, we will discuss the most complex SOA patterns and their realizations in support of the distribution and analysis of events, parallel in-memory processing, and the readiness of Oracle Cloud.

9
Additional SOA Patterns – Supporting Composition Controllers

SOA is a form of distributed computing, or more precisely, an architectural approach to establish it. It is maintained on heterogeneous environments by means of API standardization and canonicalization of service activities by service messaging/state messaging. The ability to arrange the service composition dynamically and in an agnostic way is arguably the main benefit of this architectural model, as demonstrated in all the previous chapters. This is where the money is. About 80 percent of all SOA patterns are focused on achieving composition-centric SOA characteristics (among others mentioned in *Chapter 1, SOA Ecosystem – Interconnected Principles, Patterns, and Frameworks*) directly:

- Implementing Service Inventory and maintaining a dependable Service Registry
- Minimizing the Service Adapter layer and avoiding transformations
- Abstracting services and maintaining their state, even for completely stateless services
- Allowing service invocations in an agnostic way

We have been working on all of this attentively in the previous chapters to provide you with a blueprint of the working solution. Now you have all the essential components to implement a business-driven, composition-centric solution right away. This is suitable not only for Order Management (as an alternative you can go for OSM, Oracle Service Management/Order Management at `http://docs.oracle.com/cd/E35413_01/doc.722/e35415/cpt_overview.htm` but also for Telecom and getting another silo, where you will implement all special cases as custom cartridges and integrate them into your Service Inventory using OFM/AIA). At the very least, it will be quick. Maybe, in addition to this, it will be telecom-specific with about 40 percent coverage of real telecom needs. Mind you, it's not bad at all; it totally depends on your business/IT strategy.

Still, something quite important is missing in this picture, something that's not directly connected to any SOA pattern in the public catalog but essential in terms of practical realization. In the previous chapter, we mentioned several technical monitoring tools, but business monitoring must be in place as well. Oracle BAM is an obvious choice from the Oracle stack and is not related to any pattern but is capable of addressing this requirement. While discussing the Adapter framework and fault handling, we tapped into the Event Driven Messaging pattern several times and presented working solutions, employing this pattern. Here, we will look at conventional OFM ways of event utilization and discuss their benefits and limitations. These limitations could have very profound effects on the overall performance of an SOA solution; the next topic for this chapter will teach us how to boost performance using the Service Grid pattern and its Oracle realization: Coherence. Some other HA patterns will be discussed as well.

In the final part of the chapter, we will talk about the relations between SOA composition patterns and cloud implementation patterns, mainly focusing on aspects related to converting the Service Composition Broker into the Cloud Broker. The Cloud Broker's responsibilities such as cloud Resource Balancing with an Elastic Resource Capacity management, cloud services' consolidation and aggregation, and Burst In and Burst Out, are derived from the agnostic controller's features; we have to highlight them in order to avoid the pitfalls that SOA met in its early days. There is no magic in the cloud; it is a form of distributed computing with principles based on SOA (among others). We'll discuss this, but our introduction will be short as it is not the subject of this book.

Processing complex events

We are glad to see that the number of publications that oppose EDA and SOA have considerably reduced in recent years, but still there are some who believe that these technologies differ.

Even the word *extending* is a bit too exuberant to describe the relations between SOA and EDA in general (again, being practical and factual, we do not participate in any discussions of that kind). Why is there, then, such a statement seen so often in publications related to OFM? Well, because in the case of OFM, it is quite true. The following concepts in Oracle SCA are not exactly new but were not established from the very beginning:

- Establishing sensors in BPEL to detect events
- Emitting (signaling) events using BPEL (via **Invoke | Interaction Type | Event**) or Mediator by subscribing to them (Mediator)
- Consuming emitted events

As you may have noticed, we have been discussing the implementation of SOA patterns on a wide range of Oracle products and sometimes even avoiding using particular products, striving to maintain the vendor-neutrality SOA characteristic (yes, still using the Oracle platform). Maybe it is too late to mention this in the last chapter, but there is more than one way to skin a cat, and SCA/BPEL is just one of the ways to establish EDN. We have already discussed the DB-based EDN in detail, speaking about the optimization of adapters' layer, and now we are about to glimpse the Oracle CEP EDN.

So, what do we have in terms of EDA/EDN apart from SOA Suite and Service Bus? As always, Oracle has several options to offer, depending on your technology stack:

- **Oracle Event Processing (Version 11.1.1.7 at the time of writing this)**: This is a standalone solution for building applications to filter, correlate, and process events in real time. It is generic and suitable for designing an all-purpose SOA infrastructure.
- **Business Event System (BES)**: This is part of the Fusion Application. Although it is not an atomic EDN in the common sense, its emitted events are a significant type of composition activations (initiations), and they have considerable impact on the ABCS/SB design. If you have OEBS as the centerpiece of your infrastructure (a mix of silo and SOA), then it is inevitable that you will have to handle it.
- **Oracle Real-Time Decisions (RTD)**: This is a rule-based recommendation engine that is most commonly used as part of the decision solution in Siebel E-Commerce Suite. This tool is interesting because of the prediction it brings into the analysis of a runtime event. This option makes its highly attractive for its integration with eligibility and entitlement servers (Oracle) in up sales and customer retention solutions, for instance, in telecom for ad insertions and VOD-personalized recommendations.

In addition to this, in the chapters dedicated to adapters and exception handling, we demonstrated how to capture and process different kinds of events in a product-agnostic way. Obviously, these examples were related to business and error events only and processing was quite simple, although, very common. If we step aside from the telecommunication/logistic primers in previous chapters, the magnitude of EDN tasks will definitely be wider. Thus, some analysis will be carried out in order to clarify the requirements and clearly position the EDN stack in SOA frameworks.

Initial analysis

Traditionally, we will start with the analysis in a vendor-agnostic way, and we would like to repeat the basic things here again just to summarize the facts from previous examples and clarify the EDN-SOA hierarchy. Please get back to our SOA metadata taxonomy that is visualized in the *Managing Service Repository* section in *Chapter 5*, *Maintaining the Core – the Service Repository*. An event is the next thing that follows an object, indicating the object's change of state. That's it! It's not just some faceless and abstract "occurrence on the input receptor", as someone may describe it. Sorry, but occurrence of what? The event has always happened on (or around) something solid, and when it happens, object(s) can be serialized in their transportable form: a message. It is not always required though, and we will describe when it will be.

In the Event Processing documentation (`http://docs.oracle.com/cd/E28280_01/doc.1111/e14476/overview.htm`), Oracle provides examples of possible events, including the following:

- **Communication events**: These are based on the fact of message/transport container reception. The object is a transport container.
- **Machine events**: Usually (in Oracle realization), these are handled by Java-embedded devices; some on ME for small devices without complete Java support (pacemakers or microwave microcontrollers, for instance); some on SE, such as the ones installed on high-speed communication gateways in datacenters that handle service edges; and some on communication events mentioned in the preceding point. For telecom, a good example of such events would be booting/starting STBs or cable modems with the TR-69 device management protocol's support when devices' changes of state are emitted to the **auto communication server (ACS)**. It is obvious that in this case, the object is a device itself.

- **Security events**: These are derived from the preceding points and are an object's AA request to change its privilege status (`I want to invoke... I want to participate`); these are usually detected on the Service Gateway's edge. Equally, all other objects' alterations related to all security patterns from *Chapter 7, Gotcha! Implementing Security Layers*, can be included in this event type.
- **Environmental events**: These include infrastructural events such as WLS node up/down and basic environments such as temperature/pressure shift and so on.
- **Business logic events**: Generally, all events are associated with business objects' change of state in our entity services.

Surely, the list is not complete, but we believe that even a short enumeration clearly indicates the master object behind each of them. At first glance, the list looks obvious, so why have we mentioned these items? This is because we would like to analyze the common regularities associated with these event types. At least two of them can be easily spotted.

Machine, environmental, and communication (at the service edges) events are most commonly synchronous or asynchronous requests. We must receive a transportable form of the object (EBM/ABM) in order to react to changes on a remote component/layer. Signals from the embedded Java or hardware sensors are also object-based, as they transport information about certain changes in an object (for example, the pulse is changed, the device is booting, the OSGi container is initiated, and so on). These requests are typically produced (emitted) by a composition initiator (or just the initial sender in simple cases).

Security and business logic events can also be propagated by a request from the current object handler, where the change occurred. The object handler is not exactly the application-object owner (or entity service) that could be any of the current composition members, processing EBO according to business logic (or security policy logic in the case of security, or both). In this case, when we have complex compositions (business logic) or lots of environmental data (security logs) to analyze, we have to deal not only with the object itself, but with the object's environment as well (please see the figure in *Chapter 5, Maintaining the Core – the Service Repository*, in the *Message* section). Simply put, some changes can go undetected or even misinterpreted if we only take into account a single object's visible changes without the object's context. Ultimately, some complex changes can be registered only by analyzing the context's information during an extended period of time, and this is true for environmental events as well. Thus, the time-driven object-handling activation (after event recognition) is the second type of event-driven service activity initiation. The main difference with request-driven realization is that for time-driven activation, we need a special type of service agent.

These agents were depicted in the figure in the *Automated recovery concepts* section in *Chapter 8*, *Taking Care – Error Handling*, and so we can count at least three major agent groups:

- Event collectors
- Event aggregators
- Event analyzers

The first type of agent group most commonly is a set of individual adapters/sensors, collecting object change metrics from various sources; we use different colors (deep green) to indicate that they have little potential of being reused. You will have to add a new adapter for every new source; the Adapter Factory pattern with universal adapters has limited application here due to performance and reliability constraints.

Indeed, as mentioned earlier, a reflection attack involves two separate handshake sessions, and the service agent running on Security Perimeter must analyze the broad session context to detect it. Additional agents must be employed in order to analyze logs' repeatable patterns or irregularities. For business services, the agent must control service invocation footprints to check for invocation misses, belated responses, or certain response code (as was mentioned during our ART discussion).

The last two agents' types can be combined in one Event Aggregator;
this is how it is done in AIA Foundation Pack (refer to the Development Guide DevGuide Version 11.1.1.6.3). This aggregation model can be used for relation- and time-based aggregation (usually both). Naturally, data cleansing and normalization will be required, so both transformation patterns will be involved. Events aggregation should provide the following:

- Synchronization of an entity, providing a single, holistic view of the entity
- Consolidation of several fine-grained events into a single, coarse-grained event
- Merging of duplicates of the same event

Aggregation is probably the most difficult part of event filtering because correlations between disparate event streams from event collectors and the necessity to analyze a data block for an extended period of time is not always obvious. Declaring the constrained collections against the available data, so-called data patterns, is one of the possible techniques that requires Metadata Centralization in the way we described it in chapters dedicated to metadata taxonomy and exception handling (*Chapter 5*, *Maintaining the Core – the Service Repository*, and *Chapter 8*, *Taking Care – Error Handling)*, and the involvement of Rule Engine. As RE is necessary, Rule Centralization is also obligatory for the second type of event service agents, and this is true not only for a time-driven event but for request-driven event processing as well.

> As an immediate conclusion of these analyses, we can point out Rule Engine as an essential part of EDN; consequently, a set of service agents (SCA decision services) are linked to RE, which in turn implements the Rule Centralization pattern (at least for event processing purposes). This RE-SR-SA implementation model is highly flexible because of its decoupled realization. At the same time, the role of RE in EDN is so crucial that sometimes EDN can be entirely based on the RE alone. For instance, one of the leaders of commercial rule management, namely, Blaze Advisor RE (from FICO) with a highly advanced **Structural Rules Language (SRL)** and an enhanced concept of data patterns is in fact a combination of RE, Workflow Engine (so-called RuleFlow), and EDN. Although this product is brilliant, we cannot describe this concept as very flexible because several SOA principles are not really supported. If the nature of your events and their object sources is broader than just the calculation of taxes/pensions, please consider separating EDN layers and agent groups.

Event consolidation, filtering, and aggregation serve the following two main purposes:

- Identification of the business (complex or compound) event from a series of time- or correlation-related basic events
- Recognition of consumers' event subscribers of these business event(s)

Needless to say, event message delivery must be strongly conducted to **Idempotent Activity** settings for **Partner Link** (if you are using BPEL). It not only affects the performance (when set to false), but can also be undesirable from the business perspective (resending the same notification twice if set to true by default after restarting the faulty process.) In any case, resending the same notification is not a good idea. We will return to the application of reliable messaging in EDN a bit later; now, it would be prudent to demonstrate the handling of temporal and business-correlated basic events, and their aggregation and recognition as a compound business event using RE. As a theoretical background, we will refer to the classic publication *Formal Semantics For Composite Temporal Events in Active DBS, Iakovos Motakis, Carlo Zaniolo, UCLA/Cambridge Technology Partners Publication*. This publication introduced the Event Pattern Language (EPL).

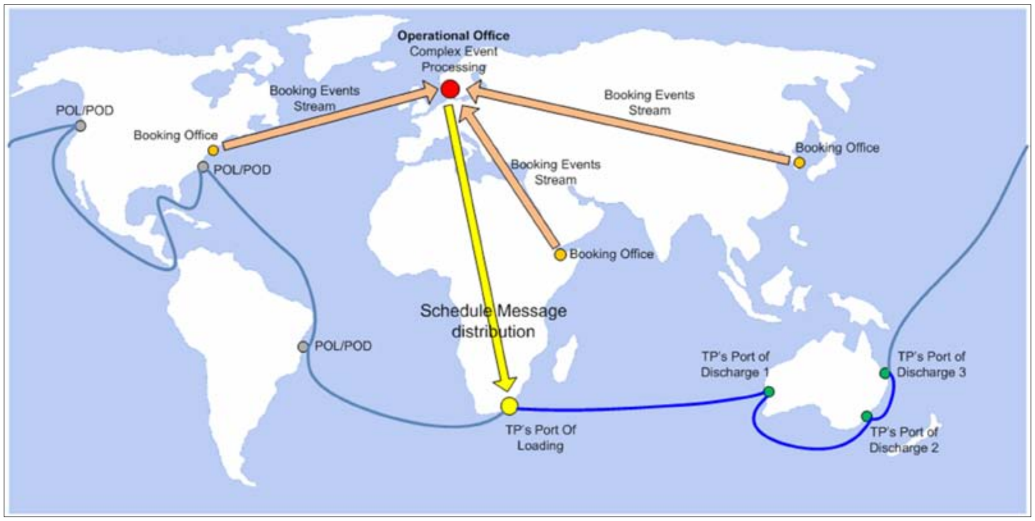
Processing Object Context in business logic events

For practical purposes, we would like to revisit the logistic example we demonstrated earlier, now with the example of a shipping company.

The shipping company has a number of vessels, transporting cargo on various routes with multiple port calls. (Again, how many objects have we just listed here? Count carefully.) Port calls could have several different purposes such as loading, unloading, bunkering, maintenance, and so on (quite often combined). Some customers (cargo owners) would like to be informed only about schedules for certain routes with certain port call types in selected ports, performed by certain types of vessels. Business reasons could be different, and some of them are as follows:

- A customer is looking for the shortest delivery time, without any transshipment
- A customer (such as US Ministry of Defense, for instance) wants to deliver special cargo without entering certain ports in certain countries
- Obviously, the vessel must be appropriate, for instance, you cannot load a caterpillar heavy duty truck on any Ro-Ro vessel; sometimes vessels that have decks with adjustable heights will be necessary
- Obviously, a customer's port of destination must have *unloading* as the port-call property; otherwise, it serves no purpose

This is just a tiny part of the business rules; the complete one will require three pages, where some conditions will be technically unrelated to the Schedule object, such as the nationality of the crew or the last three ports where the crew members embarked. A change in any of these conditions could lead to publishing/propagating the schedule to the customer or revoking it: port-call purpose change, removing port call in restricted area, and so on. The purpose of detecting and analyzing the Object Context is clearly visible from this example:



Rules are a source of information for Rules Engine to make decisions regarding messages' trading partners and actions. Rules are based on the following available information:

- **Basics Event (BE)**, provided by **Business Application (BA)**. BA can provide a Business Event's description directly in some cases.
- Sender ID (Sender's BA TP code).
- Object reference and Object Context.

The Rules Engine mechanism is explained using the following examples:

Business Case	Message, Basics Events, and Business Application statuses
Case 1	BE: `Port call ETA has been changed for voyage <voyage id>` BA Sender: `BA1` BA Operation: `UPDATE`, a basic event pattern Message Code: `Schedule`, a generic message
Case 2	BE: `Port call purpose has been changed for port call <port call id>` BA Sender: `BA1` BA Operation: `UPDATE`, a complex event pattern Message Code: `Schedule`, a generic message
Case 3	BE: `Port call has been changed for cargo unit <cargo id>` BA Sender: `BA1` BA Operation: `UPDATE`, a basic event pattern Message Code: `Cargo`, a generic message

Case 1 – basic event type

We can combine solutions for these use cases in two main groups, based on the event's type:

- The execution of a simple event expression will be sufficient in this case
- Using this information, Rule Engine will select all TPs that are bounded to this rule unconditionally and with no historical retrospective, such as DWH and corporate portal
- The rule is simple; select the TPs bounded to this message and BE

Cases 2 and 3 – complex event type

- This is the same BE, but TP here is interested only in port calls for one or many ports in the voyage.
- TP maintains its own application database messages received from BA-sender (complete decoupling). In this case, an operation on the Rule Engine side should be performed, namely, extracting/parsing historical information about a business object, such as the previous port name of the voyage and the previous ETA of the port call or previous vessel.

Realization of business cases using event processing (Rule Engine)

To do business with a shipping company, the Trading Partner (TP) must be provided with Schedule object(s) and should be compliant with the following composite conditions:

- All voyages with the POL `PORTCODE_A <and>`
- ALL PODs in certain `COUNTRY_ID <and>`
- With two certain PODs in `TRADE_AREA`

These conditions are rather obvious, as TP must have a port with goods waiting to load (POL) and all the trade ports for unloading goods (PODs) according to the carrier's schedule. In addition to this, some more operational conditions must be applied:

- A port call must *not* be canceled (some kind of suspension status that is undecided)
- A voyage must be in the public distribution state (available for booking, that is, the vessel is on a commercial route and is not going for maintenance)

Let's see how these five conditions can be interpreted using basic events on the sender's side in terms of DB CRUD operations:

- Adding a port call to the voyage using `INSERT` on the TP side
- Updating port call information, which includes operations that depend on conditions based on TP rules, in the following manner:
    - `DELETE FOR ALL` ports in the voyage if POL `PORTCODE_A` is removed (simply, TP is not interested in the voyage without its port of loading)
    - `INSERT FOR ALL` ports in the voyage if POL `PORTCODE_A` is added and POLs
    - Are in the TP's area of interest (opposite to the previous point; the entire voyage will be provided if the TP's port of loading added)

    - `INSERT FOR SINGLE` port if new POD is added in the TP's area of interest (simple adding new port of discharge)
    - `DELETE FOR SINGLE` port if the POD belongs to the TP's area of interest
    - Removed from voyage (opposite to previous point)
    - `DELETE FOR ALL` ports in the voyage for previous condition if removed

  POD was the last port in the TP's area of interest, which means that the voyage had one POL and one POD. Two messages should be provided in this case: one for single port and the other for the entire voyage.

- Deleting a port call from the voyage using `DELETE` on the TP side for one or more voyages in the list.

You can see from the analysis of the preceding operation that updating the port call information is the most complex task and requires historical retrospective. The Event Processor (such as Rule Engine (RE), for instance) will need to have information about the previous ports in voyages to make the decision regarding using `Insert`, `Update`, or `Delete` operations provided in Message Header to TP. The `Insert` and `Delete` operations do not reflect physical operations on the BA side. As long as the message body provided by TP cannot contain new and old information at the same time, previous information about the key values must be provided in the Message Header's Object Context part. This fact was explained in *Chapter 5*, *Maintaining the Core – the Service Repository*, dedicated to metadata taxonomy, where we linked ESR taxonomy to the message structure.

The preceding realization can be physically presented as a two-level rule. Here, we have a conjunction of a single entity with disjunctions.

<root> voyage must have POL1[and]

<level one> POD must be in PODn1,PODn2…PODm1,PODm2

For the second use case, we have additional realization requirements, as follows:

- The number of keys needed for resolving TP can be more than one.
- To be effective, RE (if we use Rule Engine) should execute as many rules as possible in one go. It can be achieved by:
    - Using an effective cost-based parsing ruleset and indexes.
    - Reducing the number of rules applied to one complex event pattern.
    - Composite events signatures can be invoked. Signature composition.

- Rules' tasks (or references to the tasks) and expected values should be stored in ESR ruleset tables. Consequently, they can be part of the execution plan, but the performance could be affected. Furthermore, we will see how CQL could address this issue.
- Rules' validation could be based on the evaluation of XPath expressions. As mentioned in the preceding bullet item, performance can be affected negatively here as well.

Based on these two samples, RE can be briefly described as a mechanism that works on the already mentioned **Event Pattern Language (EPL)** semantics and principles. An EPL-based Rule Engine comprises the following components:

- Components and basic events (as in case 1):

  `Insert(RName),Update(RName),Delete(RName);` RName is the Object Ref
- Simple Event expressions (basic event plus the condition of the basic event, by default, "ALL"):

  `evntkind(RName(X), <condition-expression>)`
- Actions; they refer to the action performed on the rule. According to architectural rule 1, you cannot perform backward lookups on RE actions from MB/SB but can perform forward lookups from BA.
- EPL Modules (MB RE rulesets) or collection of rules. This collection is indexed.
- Any module has two parts: the first is the declaration of the basic monitored events and the second is a list of rules, which will be applied to the declarative part. Declaration of basic monitored events is part of the Object Context and provided by BA. Rules are stored on the MB RE side and linked to the BE code and associated with BE's basic events:

```
Begin Ruleset
        Begin declaration
              Monitor Insert(CARGO)
End declaration
Begin Rule
      CompareWith(EventSatisfaction,
      Apply(valueOf mhs:< XPath-to-key-element>)
              End Rule
 End Ruleset
 EventSatisfaction -> POD_CODE
     where
             declarations shall be provided by BA
     and
             rules extracted from Rule DB by RE on MB/SB side
```

Event expressions could be of the following types:

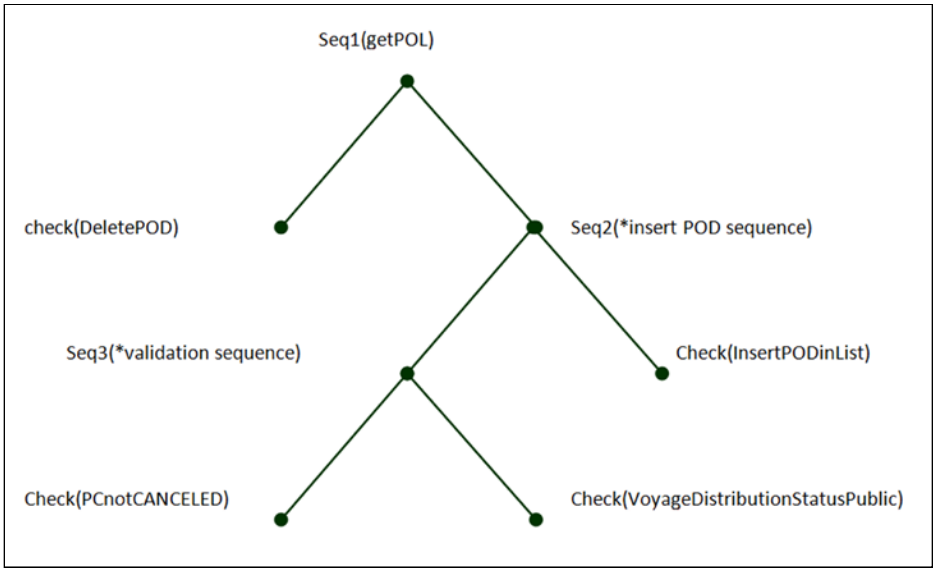
- Immediate sequence
- Star sequence
- Conjunction
- Disjunction
- Negate

Some of these expressions were demonstrated in the case 2 sample.

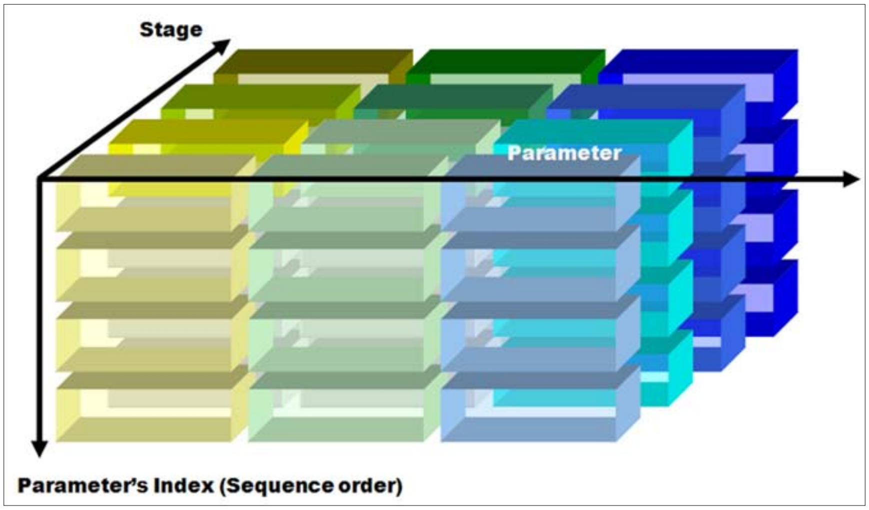
Additional constructs can be derived from the following expressions:

- Any (for the rule `ALL`)
- Relaxed sequence
- Prior operator

The EPL RE semantic will also require the presence of event tables and event histories on both sides (MB RE and BA); this is crucial for the Trading Partner Business Event's interpretation, and not only for TP recognition. To avoid exponential blow up during ruleset invocation satisfaction, predicates must be presented for every rule-execution in MB RE. This can be explained by presenting the rule-parsing tree:


Seq1(getPOL)
check(DeletePOD)
Seq2(*insert POD sequence)
Seq3(*validation sequence)
Check(InsertPODinList)
Check(PCnotCANCELED)
Check(VoyageDistributionStatusPublic)

The preceding figure is the decision tree with meta-rules expressions for the sample in case 3. The decision nodes of the tree are numbered according to the post-order traversal sequence. Sequences **2** and **3** will potentially require historical information for the referenced object via the Object Context. Declaring this, we have to convert the tree-parsing model to a 3D composite event matrix of parameterized events, where parameters are separate object references to the same object at different time stages and coexisting objects; see the description of the Object Context:



We believe that even if oversimplified, this complex business case is descriptive enough to understand the 3D nature of complex business events. Every business object such as a consolidated Voyage Schedule can only be seen in the following scenarios:

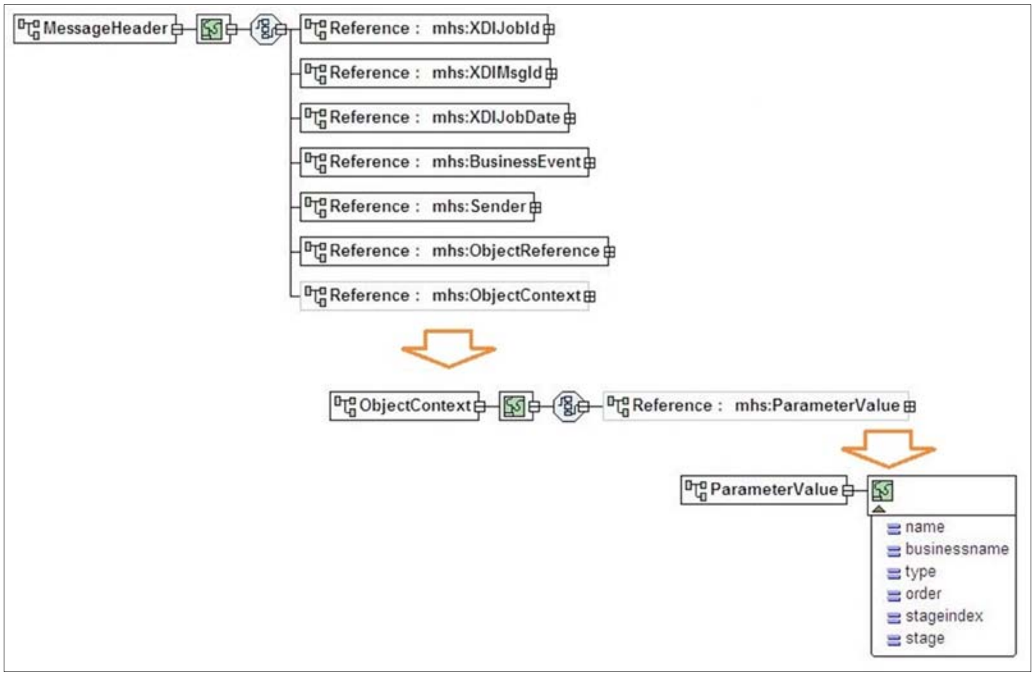
- In the context of other supporting business objects, such as ports, port calls and their purposes, vessels, and crew
- With an object's historical retrospective (history of the port calls, crew shifts, and so on)
- It can also be seen as an unbounded set of uncorrelated object data streams registered by event interceptors

In this matrix, we have the following properties of parameterized events, which comprise the Object Context XSD part:

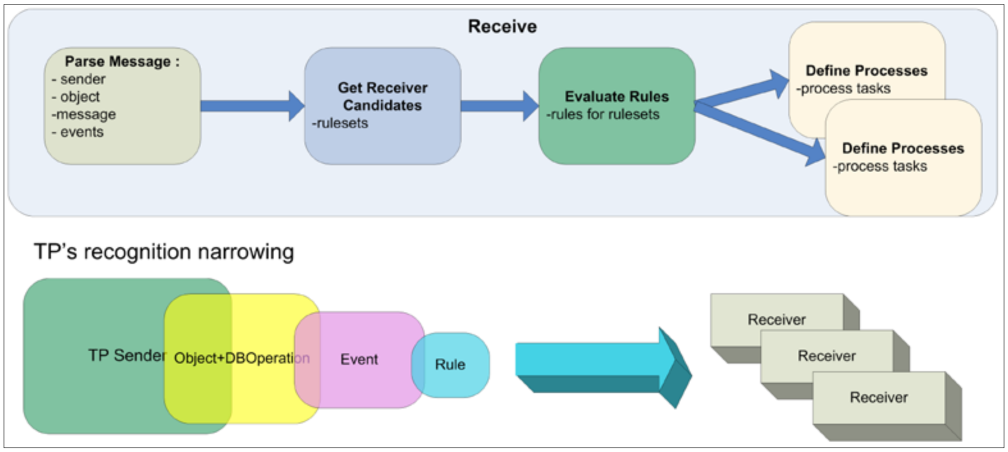
- `Stage`: This is the historical indicator on the time axis for the object entity. Theoretically, it can be endless, but at present, DB practice can have new and old values.
- `ParameterName`: This is how a parameter is identified as an object's property.

- `Value`: This is the parameter's actual value.
- `CodeName`: This is the class name key reference that is identified using keys, namely, `PORT_CODE`, `VESSEL_CODE`, and `GARGO_ID`.
- `BusinessName`: This is the business meaning of `PORT_CODE`, such as POD, POL, Port of Destination, Port of Origin, and so on. You need it for your own sake—the context of the message must be human readable.
- `ParameterType`: This is an optional, scalar (string, number, or byte), or composite data/object (array, CSV list, or vector) type.
- `Order`: This is optional. As long as a parameterized event can be an array, vector, or sequence of basic types, this index will represent the position of an individual element in the sequence.

Technically speaking, you can see the object context as a form of name-value pairs array with some additions for historical retrospection; we discussed it earlier in *Chapter 5, Maintaining the Core – the Service Repository*, while talking about Message Header. Here, in the following figure, you can see the simplified version of this complex type that is suitable for events propagation (or, more correctly, basic events' notifications). The following structure is essential in order to understand the event-type repository of Oracle Event Processing:



The propagated object context along with other MH elements will be used for aggregation, correlation, and filtering TP-Subscriber recognition; see the following figure:



This block diagram is strictly functional, and its only purpose is to summarize the actions that RE and EDN agents have to perform in order to initiate the predefined orchestrated process (upper part) or deliver the event notification to the ultimate receiver(s) (lower part). For further reference, please be aware that the receiver could be anything, for instance, Nagios, as mentioned earlier, or Oracle BAM, as we will demonstrate later. Actually, we are missing one crucial design rule here and will return to it: formalizing the EDN requirements.

You will have certainly noticed some references to the DB operations in the figures and drawings. This is normal for Oracle with its immense DB legacy, and consequently, with its own realization of the CEP query language in addition to the already mentioned SRL and EPL: the CQL. This is a continuous version of the traditional SQL with constructs for supporting the streaming of data, and it is an essential part of Oracle's. Event Processing Suite (`http://docs.oracle.com/cd/E28280_01/apirefs.1111/e12048/toc.htm`).

Thus, take a look at the figure in the *Realization of business cases using event processing (Rule Engine)* section that describes the 3D nature of complex events as a parallel flow of events (or basic event-indicating tuples, from left to right), combined with the required processing sequence (see the lower part of the previous figure; it is just one of the possible processing routines around TP recognition).

There you will see the old SQL-like approach (including the oldies but goodies `GROUP BY`, `ORDER BY`, `JOIN`, and `HAVING`) perfectly match the parallel processing of disparate event streams that are enforced with the following CQL query clauses:

- `MATCH_RECOGNIZE`
- `XSTREAM` (and `IStream`, `DStream`, and `RStream` in relation to stream operators)
- Built-in window queries for stream-to-relation operations
- Also, surely, `XMLTable` operations on the Oracle `XMLType` data, using XPath clauses that are similar to the EPL meta-code snippet shown previously

So, you have all the necessary means to fulfill core EDN tasks (collect, aggregate, correlate, and analyze) using CQL for all types of data (by location: streams, and DB) and with all types of relations (time-based or CorrId/CorrSet-based, or both); the `MATCH_RECOGNIZE` clause is probably one of the most powerful options here. Frankly speaking, this clause is not an exclusive invention of CQL; it's quite common for all EPL types of languages. Look at Esper for instance; it has also superseded the old 10*g* `MODEL` clause in 12*c* RDBMS, so you can use it in regular SQL statements as some form of case.

> To avoid possible confusion, we have to give very short explanations here. Firstly, there is no contradiction between EPL and CQL here; you will find them both in the Oracle EDN documentation: `http://docs.oracle.com/cd/E23943_01/dev.1111/e14301/toc.htm`. Simply put, EPL is the old-fashioned way of defining filtering, recognition, and so on. Rules (in *Chapter 5, Maintaining the Core – the Service Repository*, we advised to keep the structure and format in XML for better portability) and CQL are the modern query-based workhouse, performing all of the heavy work or aggregation, correlation, and matching. Both require a dedicated processor. Now what? Yes! You are right. The second figure in this chapter with the decision-parsing tree is in fact pretty close to the branch and bound method from discrete programming and combinatorial optimization. We cannot stress enough how important it is for real-life event processing (in the e-commerce era, classic problems such as MAX-SAT, TSP, and NNS were not just theoretical exercises; they were means to generate money—even Oracle's EDN examples are mostly stock-related); understanding these subjects is highly desirable for efficient EDN implementation, but unfortunately, this subject deserves a separate book.

To be consistent, we would like to repeat the previously mentioned EPL meta-code snippet using CQL and take into account some additional requirements we expressed earlier, such as loading the ramp type and port of discharge in a certain area. This statement is still oversimplified as the actual number of parameters and conditions is substantial. Some may think that the logistic primer is not completely relevant for this kind of demonstration (Oracle uses stock trade events), where the time window and PREV clauses are not really needed. Surely, telecom with event-detection abilities based on customer experience could be more appropriate for setting up a quicker event-handling process; however, if you think about it, a large shipping company with 300 vessels for 4,000 cargo units each, hundreds of booking offices, and thousands of clients will desperately need the voyage schedule optimization based on the event's recognition for up and downstream:

```
<?xml version="1.0" encoding="UTF-8"?>
<n1:config
    xsi:schemaLocation="http://www.bea.com/ns/wlevs/config/application
wlevs_application_config.xsd"
    xmlns:n1="http://www.bea.com/ns/wlevs/config/application"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <!--  Root: Defining  the CQL Processor. Note:  EPL processor has
the similar root-->
   <processor>
   <!--  Processors name declaration: Abstract reference -->
        <name>voyageSchedulerProcessor</name>
   <!—Here goes the main block: Rules  -->
    <rules>
   <!—Declaring the view , partitioning it by port call purpose-->

      <view id="recentPortVoyageUpdateEvents" schema="cusip mbid srcId
bidQty ask askQty seq">
          <![CDATA[
                Select voypc.voy_code,
                voypc.port_voy_code, voypc.port_name,
                voypc.call_purpose,  voypc.call_duration,
                voypc.call_status,    voypc.port_terminal_type
                from filteredStream[partition by call_purpose]
            ]]>
  </view>
 <!-- Here goes the query, based on the view, declared above -->
 <!-- We using the patterns and MATCH_RECOGNIZE here for DEFINING the
types of ports, purposes and loading facilities -->
```

```
 <!-- we want to identify according to Customer preferences and inform
him accordingly -->
 <!-- This statement is for demonstration purposes only -->
 <query id="detectScheduleByPOD">
         <![CDATA[
                 SELECT *
                 FROM   recentPortVoyageUpdateEvents
                 MATCH_RECOGNIZE (
        MEASURES
                              A.port_voy_code as port_voy_code,
                              B.call_status as call_status,
                              C. port_terminal_type as port_terminal_
type
        PATTERN (A+ (B+ C+))
        DEFINE
                    A as (A.port_voy_code = "<CLIENTS_POL>" and
A.call_purpose ="<LOAD_ONLY_CODE>"),
                    B as (B. voy_code = A. voy_code and  (B.port_voy_
code="<CLIENTS_POD1 >"  or B.port_voy_code="<CLIENTS_POD2 >"),
                    C as (A.voy_code = C.voy_code  and C.port_
terminal_type  = "<REQUIRED_RAMP_TYPE>"
    ) as voypc
           ]]>
       </query>
     </rules>
   </processor>
</n1:config>
```

In the preceding example, we partitioned the input stream (just one of the available streams; we have many of them, such as the one for cargo, for instance, and coming from booking offices) `filteredStream` using a port call purpose in order to separate loading and discharge port registrations. To detect the complex event and report on it (providing the available schedule details to the client), we use the following:

- The PATTERN of three main variables—the certain POL followed by one or more PODs with correct ramp types
- Define the tuple's correlations

The statement can be further optimized by declaring `SUBSET` for a collection of PODs, `DURATION`, and `INCLUDE TIMER EVENTS`. For more details, please see the clause syntax at `http://docs.oracle.com/cd/E12839_01/doc.1111/e12048/pattern_recog.htm`, and you will find a lot of examples at `http://docs.oracle.com/cd/E15523_01/doc.1111/e14476/examples.htm`.

Communication and machine events

Some may say that just sending the change notification with the description of an event could be enough for the customer. Yes, sometimes it is. Interestingly, event description is an object itself, as explained in *Chapter 5*, *Maintaining the Core – the Service Repository*, while discussing the SBDH-compliant message structure. Thus, if in your design you are following the patterns related to canonical service messaging (messaging metadata, state messaging, canonical schema), then the event notification message could be an EBM without an EBO, with only the SBDH part with an object context part, if necessary.

The Message Tracking Data can be omitted as well. Here, we can refer to the point we discussed in *Chapter 6*, *Finding the Compromise – the Adapter Framework*. If an events notification consumer can process the request (or control the possible composition based on this event), then only the message header will suffice. If not, we should instead put some effort in the events source side to construct a complete EBO/EBM; otherwise, we will have to pull the object from the client side. This is the general design rule, and we have demonstrated earlier in *Chapter 6*, *Finding the Compromise – the Adapter Framework*, how to follow it in the "Integration of OeBS/BES -> OFM use case"; however, unfortunately even with a relatively small EBM in telecom, it is not always possible to avoid the object's polling. Talking about communication events, we would like to refer to one of the highest authorities on this subject: CISCO. Refer to the following quote from the SNMP events notification at `http://www.cisco.com/c/en/us/support/docs/ip/simple-network-management-protocol-snmp/7244-snmp-trap.html`:

> *Trap-directed notification can result in substantial savings of network and agent resources by eliminating the need for frivolous SNMP requests. However, it is not possible to totally eliminate SNMP polling. SNMP requests are required for discovery and topology changes. In addition, a managed device agent cannot send a trap, if the device has had a catastrophic outage.*

SNMPv1 traps are defined in RFC 1157 with the following fields:

- **Enterprise**: This identifies the type of the managed object that generates the trap
- **Agent address**: This provides the address of the managed object that generates the trap
- **Generic trap type**: This indicates one of the numbers of generic trap types
- **Specific trap code**: This indicates one of the number of specific trap codes

- **Time stamp**: This provides the amount of time that has elapsed between the last network reinitialization and generation of the trap
- **Variable bindings:** This refers to the data field of the trap that contains PDU. Each variable binding associates a particular MIB object instance with its current value

In SNMPv2c, a trap is defined as `NOTIFICATION` and formatted differently compared to SNMPv1. It has these parameters:

- `sysUpTime`: This is the same as a time stamp in the SNMPv1 trap
- `snmpTrapOID`: This is the trap identification field
- `VarBindList`: This is a list of variable bindings

SOA on SNMP traps? Yes, why not! We believe that nobody here at the final chapter is under the illusion that SOA is only SOAP (or REST). The concept of EDN is just one of the many patterns under the roof of service orientation; SNMP as a transport protocol works just fine for communication, in addition to the traditional TR-69 telecom.

Just look at the fields from the previous specification and compare them with the elected SBDH elements we proposed earlier. It is also needless to say that here we are also dealing with objects and notifications based on the objects' changes, not just with some occurrences. The technical monitoring solution discussed in *Chapter 8*, *Taking Care – Error Handling*, will be incomplete without handling SNMP notifications from network devices. Oracle Event Processing Suite is highly suitable for aggregating all types of events: Business/Functional and SNMP Traps notifications.

Fast events + Big Data

Yes, this simple equation leads us to something that Oracle branded as Fast Data (`http://www.oracle.com/us/solutions/fastdata/index.html`), and this is just a logical outcome of what we have learned so far:

- We provide most (if not all) of the basic data required for the complex event identification process, as the object context in the form of key-value (or name-value) pairs. The CQL with its `MATCH_RECOGNIZE` and data `PATTERN` definition allows us to do practically everything with inbound key-value event streams; however, the main point of the following bullets is that we have something to rely on in terms of data model standardization, even in such a simple form as key-value pairs.
- Oracle's own NoSQL DB is also a key-value-based, robust, and high performing database. Actually, you can configure almost any DB to be your events store as usual; it's all about the available driver that you should declare in the `<driver-name>` element. You just have to remember that sometimes this kind of integration could affect the performance, which is highly important for CEP.

- As a logical continuation of the first two points in addition to the inline events stream processing, we have the possibility to record and then playback events using Oracle's CEP/CQL capabilities. This option is very powerful, and you can consider it in additional queues and topics.

Thus, Oracle EPN Suite provides us with a truly powerful arsenal using which we can process events in FIFO style from streams of various sorts—including memory caches—pile up events in the intermediary storage (NoSQL, the most popular, or in traditional RDBMS), and reprocess them at our convenience at any given time interval.

There are two main technical reasons to use this approach. Some systems have to deal with so many events that that we simply cannot predict the message format, representing even a basic event. Downcasting any incoming object context and describing the event to the key-value pairs that are suitable for NoSQL storage should not impose huge difficulties. Naturally, we also cannot predict events' occurrence or their frequency, so the correlation and aggregation of dispersed events will require enormous computing resources and memory in particular (yes, Oracle has the answer to these high demands as well, but we will touch upon HW triad and the Exalytics In-Memory machine in particular a bit later). Here again the NoSQL storage comes to the rescue.

So, what do we solve here? The scalability problem, of course. Secondly, the ability to record, store, and replay (aggregate and correlate) is the cornerstone for truly parallel and distributed processing. That's all fine, some can say, but as long we are dealing with event notification messages, why not use the traditional JMS instead? Indeed, JMS is one of the sources/adapters in Oracle EPN and can be configured in quite a traditional way within the `wlevs` config file (EPN `config.xml`) for multiple consumers (20 in the following example for a Voyage cargo-handling event):

```
<?xml version="1.0" encoding="UTF-8"?>
<wlevs:config xmlns:wlevs="http://www.bea.com/ns/wlevs/config/
application">
    <jms-adapter>
        <name>cargoBooking2VoyageJMSAdapter</name>
        <event-type> cargoBooking </event-type>
        <jndi-provider-url>t3://localhost:7001</jndi-provider-url>
        <connection-jndi-name>jmsConnectionFactory</connection-jndi-
name>
        <destination-jndi-name>distributedQueue</destination-jndi-
name>
<session-transacted>true</session-transacted>
        <concurrent-consumers>20</concurrent-consumers>
    </jms-adapter>
</wlevs:config>
```

Several points must be kept in mind when employing the JMS queue for the discussed solution.

Apparently, message sequencing is our concern, as we cannot guarantee the correct sequence of events, especially in the multiconsumer queue. Of course, the CQL MATCH_RECOGNIZE pattern construct is directly responsible for recognizing the sequence of consecutive events (or tuples) in the input stream. We have demonstrated it in the CQL statement described previously and links to the documentation are also available. Also, Oracle CEP provides several ways to configure JMS adapters (both inbound and outbound). For instance, the inbound JMS adapter receives map messages from a JMS queue and automatically converts them into events by matching the property names with a specified event type. We can optionally customize this conversion by writing our own Java class to specify exactly how we want the incoming JMS messages to be converted into one or more event types. We will talk about event partitioning a bit later.

From this point, we can pass events data to the processor <processor> as demonstrated in the previous CQL code snippet with MATCH_RECOGNIZE. That's all good, but as we know, the problem with JMS in the default configuration is that if we took out the message for processing and fail for some reason, we will not have any possibilities to recover this event. What's even worse is that, by doing this, we most probably would have already sent the acknowledgment message to a JMS events provider. So, with this approach, the third "-ility" is now in question: reliability. It is true that in some scenarios, you really can tolerate event losses, but if you don't, it will be better to use an intermediary storage for such events.

On second thoughts, not everything is exactly well with JMS in terms of scalability either. Yes, it is multiconsumer alright, thanks to the concurrent-consumers property; however, by handling large and extra-large event streams through JMS, we will (most probably) employ the MapMessage interface (http://docs.oracle.com/javaee/7/api/javax/jms/MapMessage.html). This is the standard way for name-value pairs, but we have to remember that the allocation of JVM memory heap by this class is rather aggressive, that is, the total allocated memory could be almost twice as big as the actual message size.

So, we have to keep that in mind while planning the EPN horizontal scaling and contemplate garbage collector activities when JVM memory is depleted or too fragmented; this will consume almost all of the processor time. Just have an additional 30 percent of processor time available and keep them symmetrical. Here, the Oracle EPN clustered deployment model will be really handy (http://docs.oracle.com/cd/E14571_01/doc.1111/e14301/scalconfig.htm).

Establishing it will require you to copy your default server directory and rename the new subfolders into something more meaningful in the CEP domain root. You can create several copies (epnserver1, epnserver2, epnserver3,… epnserverN) and modify the config file (<DOMAIN_HOME>/ epnserver1/config/config.xml), setting correct values for <netio> and <sslnetio> (usually 9001 and 9011 for related tags) and edit the content of the <cluster> section according to the Oracle documentation (not much, just four elements). These modifications in config.xml must be done for all servers (ports on the same physical box must be different, of course), and the domain must be restarted.

One certain advantage of the JMS approach (compared to DB storage) is that it's relatively easy in terms of new application deployment and infrastructure support, but some changes should be made to the CEP application itself in order to make them truly scalable and highly available at the same time; establishing the clustered servers in the domain is only the beginning:

1. Now we have parallel subscribers (or actual processors), and we have to propagate a notification to them simultaneously from our adapter, depending on the processing conditions. To do so, we have to change the JMS type from distributedQueue to distributedTopic (that is, we will replace the JMS queue with Topics) on both sides, including the one at the provider end. This change will ensure that all consumers will be notified about all the events. We will also need to define an event's partition criteria at our JMS adapter configuration file. In fact, it is a form of scalability dispatcher-broadcaster, *also* providing HA and it's implemented by com.oracle.cep.cluster.hagroups.ActiveActiveGroupBean for each BroadcastGroup member. Thus, the jms-adapter section of our CEP application config file must be extended right after the destination-jndi-name tag. Also, remember that you do not have to be exceedingly elaborate in defining the dispatching conditions. Our processor will handle the main tasks about filtering, matching, and aggregation; here, we need to just define something suitable to split the group workload, similar to **Content-Based Routing** (SOA and EAI patterns). In our cargo-handling example, area_code of the Booking Office would suffice:

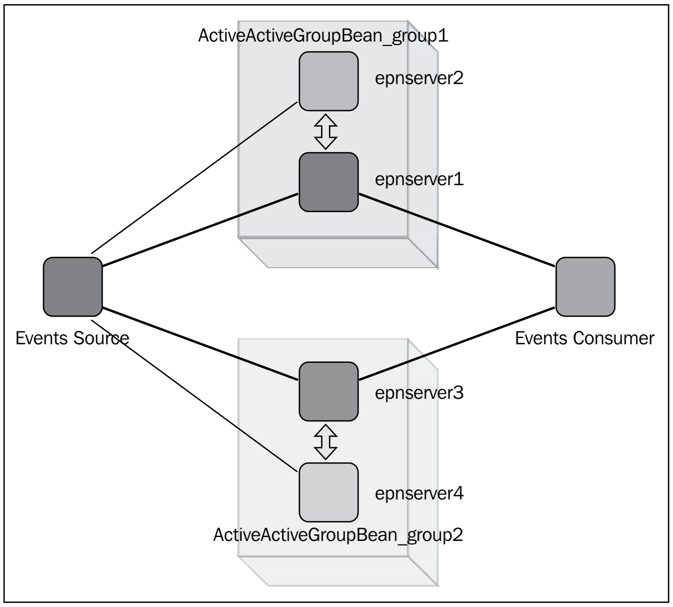
```
<destination-jndi-name>distributedTopic</destination-jndi-name>
    <message-selector>${CONDITION}</message-selector>
    <bindings>
        <group-binding group-id="ActiveActiveGroupBean_group1">
            <param id="CONDITION"><your_condition_1></param>
        </group-binding>
        <group-binding group-id="ActiveActiveGroupBean_group2">
```

```
            <param id="CONDITION"><your_condition_2></param>
        </group-binding>
     </bindings>
```

Consequently, the server's `config.xml` file must be updated accordingly with `ActiveActiveGroupBean` identifications in the `<group>` element:

```
<cluster>
    <server-name>epnserver1</server-name>
    <multicast-address><your_server_ip></multicast-address>
    <groups>cargoHandlingDeploymentGroup,  ActiveActiveGroupBean_
group1 </groups>
    <enabled>true</enabled>
</cluster>
```

As you can see from the preceding examples, we are moving in two-by-two formation here: four hosts (servers) in two notification groups and two servers in each of these groups as primary and secondary. The event input has two selector conditions in the message selector section. With more than one host in the notification group, `GroupBean` makes sure that only the primary node gets the event notification message. If the primary node is down, the next in line will be promoted as primary:



2. Perfect! We have parallel logical processors and active-active physical server clusters and dispatchers that are capable of routing the message to the primary (or secondary in case of a failover) node. We have reached a certain degree of parallelism, but we still do not have HA. What is missing? Yes, we still have a single JMS adapter with a single queue listener. Oracle EPN handles this beautifully. We just have to add a similar HA adapter into our application's assembly descriptor and assign failover properties, which could be just a single time-based parameter or a combination of time and the event's primary key; refer to the following example:

```
<wlevs:adapter id="HAcargoBookingAdapter " provider="ha-inbound">
    <wlevs:listener ref="cargoBookingChannel" />
        <wlevs:instance-property name="keyProperties"
value="cargoID"/>
        <wlevs:instance-property name="timeProperty"
value="bookingTime" />
</wlevs:adapter>
```

With many parameters to correlate, you can assign an entire class as an instance property. The outbound adapter is configured in a similar way; for more details, please see the Oracle EPN documentation. For brevity, we also skipped configuration details for channels and listeners.

Compared to JMS's configuration routines, the DB intermediate store will require at least one more in addition to the inbound event's source adapter; however, the data source and store provider must be configured first:

```
<data-source>
    <name>VoyBookEvt</name>
    <connection-pool-params>
           <initial-capacity>30</initial-capacity>
           <max-capacity>80</max-capacity>
    </connection-pool-params>
    <driver-params>
        <url>jdbc:derby:bookingevents;create=true</url>
        <driver-name>org.apache.derby.jdbc.EmbeddedDriver</driver-
name>
    </driver-params>
</data-source>
<rdbms-event-store-provider>
    <name>event-rdbms-provider</name>
    <data-source-name> VoyBookEvt </data-source-name>
</rdbms-event-store-provider>
```

To record the events in DB, we need to specify what type of events we want to record for the adapter. Please note that in addition to the adapter, we can use the processor stream and event beans for event recording:

```
    <wlevs:event-type-repository>
        <wlevs:event-type type-name=" cargoBooking ">
            <wlevs:properties>
                <wlevs:property name=" cargoID "  type=" long" />
                <wlevs:property name=" bookingTime "
type="timestamp" />
            </wlevs:properties>
        </wlevs:event-type>
    </wlevs:event-type-repository>
  <adapter>
    <record-parameters>
        <dataset-name>bookingRecPlay</dataset-name>
                      <event-type-list>
                           <event-type> cargoBooking </event-type>
                    </event-type-list>
</adapter>
```

In this case, it's a `cargoBooking` event, and this event could be one of the many defined events in the event type repository at the beginning of the configuration file. Actually, the `<event-type-list>` part is optional in adapter declaration, and if the event type list is omitted, all types of events will be recorded for further playback.

3. And now the playback itself. It's configured in the `<playback-parameters>` group and is actually pretty similar to `<record-parameters>`; here again, we can use various components for playback events. In the following code, we are adding the playback functionality to the stream, and it must be the downstream node from the recording component (which is absolutely logical):

```
<stream>
      <name>cargoBookingStream</name>
      <playback-parameters>
          <dataset-name> bookingRecPlay </dataset-name>
          <event-type-list>
              <event-type> cargoBooking </event-type>
          </event-type-list>
          <provider-name> event-rdbms-provider </provider-name>
      </playback-parameters>
    </stream>
```

Surely, we should be aware of the events type we want to stream from the store (the same cargoBooking as we recorded earlier); naturally, it should be the same event provider with the same dataset.

4. Finally, some more fine-grained tuning of this event processing setup. In the <stream> configuration presented previously, we will play all the recorded event types from our event-type list (or all events if this list is omitted). In addition to this, for recorder and player, we can assign the time internals in which we want to record and consequently replay the events using the <time-range> group. In the following example, events will be recorded/replayed from 09:00 until 18:00 on February 18:

   ```
   <time-range>
              <start>18-02-2014:09:00:00</start>
               <end>18-02-2014:18:00:00</end>
   </time-range>
   ```

   Here, we have some differences between the recorder and player. This configuration group is optional and can be omitted; however, without it, the player will play all the recorded events, but the recorder will not record the events at all. To record events, you will have to explicitly start the recording using the Visualizer console or the command-line admin utility.

   Alternatively (but not at the same time), you can define the duration of the recording/playback as follows:

   ```
   <time-range>
              <start>18-02-2014:09:00:00</start>
               <duration>09:00:00</duration>
   </time-range>
   ```

As you can see, this definition is similar to the previous one.

For brevity, we will not discuss other parameters related to store policy, playback speed, threads, event batching during the recording time, streaming parameters, and so on. We will touch upon caching in a related session as it has a certain significance for SOA pattern realization.

In the previous paragraphs, we analyzed different event processing and delivering options and practically demonstrated how Oracle EPN Suite can contribute to your CEP/EDN infrastructure, in addition to the previously discussed event-handling techniques. Let us now repeat our usual exercise and clarify the roles of SOA patterns in EDN.

EDN in the SOA stack – a practitioner's approach

We have to admit that for SOA Suite developers and architects (especially from the old BPEL school), the Oracle Event Processing platform could be a bit outlandish. This could be the reason why some people oppose service-oriented and event-driven architecture, or see them as different architectural approaches. The situation is aggravated by the abundance of the acronyms flying around such as EDA EPN, EDN, CEP, and so on. Even here, we use EPN and EDN interchangeably, as Oracle calls it event processing, and generically, it is used in an event delivery network.

The main argument used for distinguishing SOA and EDN is that SOA relies on the application of a standardized contract principle, whereas EDN has to deal with all types of events. This is true, and we have mentioned this fact before. We also mentioned that we have to declare all the event parameters in the form of key-value pairs with their types in `<event-type-repository>`; this is perfectly aligned with what we presented in *Chapter 5*, *Maintaining the Core – the Service Repository*, as a lightweight SOA taxonomy and visualized as possible implementation of the Message Header's Object Context. We also mentioned that the reference to the event type from the event type repository is not mandatory for a standard EPN adapter, but it's essential when you are implementing a custom inbound adapter in the EPN framework, which is an extremely powerful Java-based feature. As long as it's Java, you can do practically everything! Just follow the programming flow explained in the Oracle documentation; see the EP Input Adapter Implementation section:

```
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import com.bea.wlevs.ede.api.EventProperty;
import com.bea.wlevs.ede.api.EventRejectedException;
import com.bea.wlevs.ede.api.EventType;
import com.bea.wlevs.ede.api.EventTypeRepository;
import com.bea.wlevs.ede.api.RunnableBean;
import com.bea.wlevs.ede.api.StreamSender;
import com.bea.wlevs.ede.api.StreamSink;
import com.bea.wlevs.ede.api.StreamSource;
import com.bea.wlevs.util.Service;
import java.lang.RuntimeException;
public class cargoBookingAdapter implements RunnableBean,
StreamSource, StreamSink
{
    static final Log v_logger = LogFactory.
getLog("cargoBookingAdapter");
    private String v_eventTypeName;
    private EventType v_eventType;
    private StreamSender v_eventSender;
    private EventTypeRepository v_EvtRep = null;
    public cargoBookingAdapter(){
        super();
    }
    /**
     * Called by the server to pass in the name of the event
     * v_EvTypee to which event data should be bound.
     */
    public void setEventType(String v_EvType){
        v_eventTypeName = v_EvType;
    }
    /**
     * Called by the server to set an event v_EvTypee
     * repository instance that knows about event
     * v_EvTypees configured for this application
     *
     * This repository instance will be used to retrieve an
     * event v_EvTypee instance that will be populated
     * with event data retrieved from the event data file
     * @param etr The event repository.
     */
    @Service(filter = EventTypeRepository.SERVICE_FILTER)
    public void setEventTypeRepository(EventTypeRepository etr){
        v_EvtRep = etr;
    }
    /**
     * Executes to retrieve raw event data and
     * create event v_EvTypee instances from it, then
     * sends the events to the next stage in the
     * EPN.
     * This method, implemented from the RunnableBean
     * interface, executes when this adapter instance
     * is active.
     */
    public void run()
    {
        if (v_EvtRep == null){
                 throw new RuntimeException("EventTypeRepository is
not set");
        }
        // Get the event v_EvTypee from the repository by using
        // the event v_EvTypee name specified as a property of
        // this adapter in the EPN assembly file.
        v_eventType = v_EvtRep.getEventType(v_eventTypeName);
        if (v_eventType == null){
                throw new RuntimeException("EventType(" +  v_eventType
+ ") is not found.");
```

```
          }
        /**
          *  Actual Adapters implementation:
          *
          * 1. Create an object and assign to it
          *    an event v_EvTypee instance generated
          *    from event data retrieved by the
          *    reader
          *
          * 2. Send the newly created event v_EvTypee instance
          *    to a downstream stage that is
          *    listening to this adapter.
          */
          }
      }
  }
```

The presented code snippet demonstrates the injection of a dependency into the `Adapter` class using the `setEventTypeRepository` method, implanting the event type definition that is specified in the adapter's configuration.

So, it appears that we, in fact, have the data format and model declarations in an XML form for the event, and we put some effort into adapting the inbound flows to our underlying component. Thus, the **Adapter Framework** is essential in EDN, and dependency injection can be seen here as a form of dynamic **Data Model/Format Transformation** of the object's data. Going further, just following the SOA reusability principle, a single adapter can be used in multiple event-processing networks and for that, we can employ the **Adapter Factory** pattern discussed earlier (although it's not an official SOA pattern, remember?) For that, we will need the Adapter Factory class and the registration of this factory in the EPN assembly file with a dedicated provider name, which we will use further in applications, employing the instance of this adapter. You must follow the OSGi service registry rules if you want to specify additional service properties in the `<osgi:service interface="com.bea.wlevs.ede.api.AdapterFactory">` section and register it only once as an OSGi service.

We also use **Asynchronous Queuing** and persistence storage to provide reliable delivery of events aggregation to event subscribers, as we demonstrated in the previous paragraph. Talking about aggregation on our CQL processors, we have practically unlimited possibilities to merge and correlate various event sources, such as streams:

```
<query id="cargoQ1"><![CDATA[
  select * from CragoBookingStream, VoyPortCallStream
  where CragoBookingStream.POL_CODE = VoyPortCallStream.PORT_CODE
  and VoyPortCallStream.PORT_CALL_PURPOSE ="LOAD"
]]></query>
```

Here, we employ **Intermediate Routing** (content-based routing) to scale and balance our event processors and also to achieve a desirable level of high availability. Combined together, all these basic SOA patterns are represented in the **Event-Driven Network** that has **Event-Driven Messaging** as one of its forms.

Simply put, the entire EDN has one main purpose: effective decoupling of event (message) providers and consumers (Loose Coupling principle) with reliable event identification and delivering capabilities. So, what is it really? It is a subset of the **Enterprise Service Bus** compound SOA pattern, and yes, it is a form of an extended **Publish-Subscribe pattern**.

Some may say that CQL processors (or bean processors) are not completely aligned with the classic ESB pattern. Well, you will not find OSB XQuery in the Canonical ESB patterns catalog either; it's just a tool that supports ESB VETRO operations in this matter. In ESB, we can also call Java Beans when it's necessary for message processing (we demonstrated this in *Chapter 4*, *From Traditional Integration to Composition – Enterprise Business Services*); for instance, doing complex sorts in Java Collections is far easier than in XML/XSLT, and it is worth the serialization/deserialization efforts. In a similar way, EDN extends the classic ESB by providing the following functionalities:

- Continuous Query Language
- It operates on multiple streams of disparate data
- It joins the incoming data with persisted data
- It has the ability to plug in to any type of adapter
- It has the ability to plug to any type of adapters

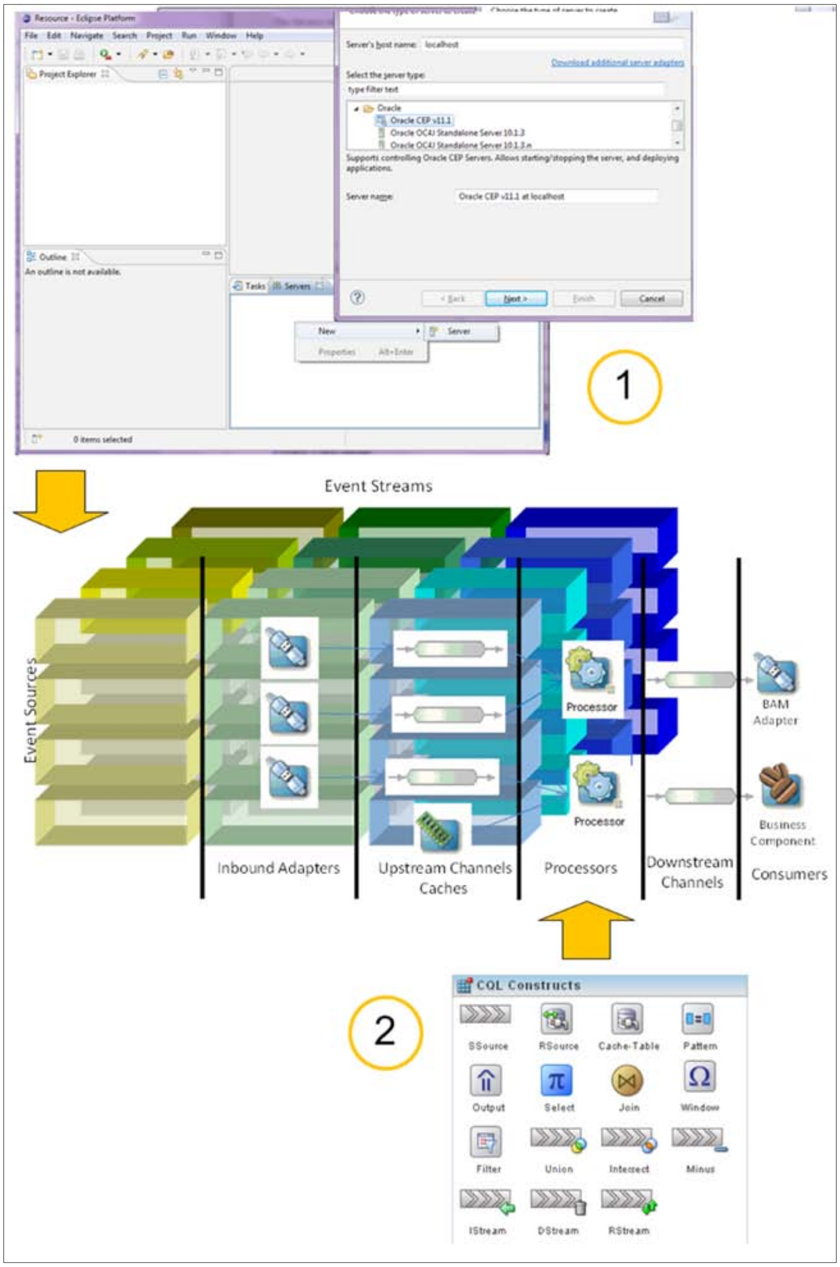
Combined together, all these features can cover almost any range of practical challenges, and the logistics example we used here in this chapter is probably too insignificant for such a powerful event-driven platform; however, for a more insightful look at Oracle CEP, refer to *Getting Started with Oracle Event Processing 11g*, *Alexandre Alves*, *Robin J. Smith, Lloyd Williams*, *Packt Publishing*. Using exactly the same principles and patterns, you can employ the already existing tools in your arsenal in the way we demonstrated in *Chapter 6*, *Finding the Compromise – the Adapter Framework* (exactly as we did for this shipping company). The world is apparently bigger, and this tool can demonstrate all its strength in the following use cases:

- As already mentioned, Cablecom Enterprise strives to improve the overall customer experience (not only for VOD). It does so by gathering and aggregating information about user preferences through the purchasing history, watch lists, channel switching, activity in social networks, search history and used meta tags in search, other user experiences from the same target group, upcoming related public events (shows, performances, or premieres), and even the duration of the cursor's position over certain elements of corporate web portals. The task is complex and comprises many activities, including meta tag updates in metadata storage that depend on new findings for predicting trends and so on; however, here we can tolerate (to some extent) the events that aren't processed or are not received.
- For bank transaction monitoring, we do not have such a luxury. All online events must be accounted and processed with the maximum speed possible. If the last transaction with your credit card was at Bond Street in London, (ATM cash withdrawal) and 5 minutes later, the same card is used to purchase expensive jewellery online with a peculiar delivery address, then someone should flag the card with a possible fraud case and contact the card holder. This is the simplest example that we can provide. When it comes to money laundering tracking cases in our borderless world—the decision-parsing tree from the very first figure in this chapter—based on all possible correlated events will require all the pages of this book, and you will need a strong magnifying glass to read it; the stratagem of the web nodes and links would drive even the most worldly wise spider crazy.

For these mentioned use cases, Oracle EPN is simply compulsory with some spice, like Coherence for cache management and adequate hardware. It would be prudent to avoid implementing homebrewed solutions (without dozens of years of relevant experience), and following the SOA design patterns is essential.

Let's now assemble all that we discussed in the preceding paragraphs in one final figure. Installation routines will not give you any trouble; just install OEPE 3.5, download it, install CEP components for Eclipse, and you are done with the client/dev environment. The installation of the server should not pose many difficulties either (`http://docs.oracle.com/cd/E28280_01/doc.1111/e14476/install.htm#CEPGS472`). When the server is up and running, you can register it in Eclipse **(1)**. The graphical interface will support you in assembling event-handling applications from adapters, processor channels, and event beans; however, knowledge of the internal organization of an XML config and application assembly files (as demonstrated in the earlier code snippets) is always beneficial.

In addition to the Eclipse development environment, you have the CEP server web console (visualizer) with almost identical functionalities, which gives you a quick hand with practically all CQL constructs **(2)**.



Server's host name: localhost

Select the server type: type filter text

Oracle
Oracle CEP v11.1
Oracle OC4J Standalone Server 10.1.3
Oracle OC4J Standalone Server 10.1.3.n

Supports controlling Oracle CEP Servers. Allows starting/stopping the server, and deploying applications.

Server name: Oracle CEP v11.1 at localhost

An outline is not available.

0 items selected

Parallel Complex Events Processing

High service performance combined with High Availability

Extracting the highest possible performance in events processing or in any distributed computing systems for that matter can be achieved only if we can leverage data caching. A new layer called data grid as a distributed in-memory processing fabric must be established around most I/O and processor-consuming frameworks and service resources, primarily around ESB and databases. Regarding service resources and DB in particular, the clustering technique has been around for quite a long time; therefore, we doubt that your mission-critical DBs are not installed on RAC using SAN and so we will not dwell into it.

To serve HA and high-performance purposes, this fabric must satisfy certain criteria:

- Data Grid/fabric partitioning around a resource-consuming framework must be dynamic and automatic; that is, you should be able to add new servers into the cluster, automatically change the partition, and consequently, data replication and the processing workload.
- The amount of data distributed around every grid node must be configurable.
- The aggregate data throughput of the fabric is linearly proportional to the number of servers.
- The in-memory data capacity and data-indexing capacity of the fabric is linearly proportional to the number of servers.
- The aggregate I/O throughput for disk-based overflow and disk-based storage of data is linearly proportional to the number of servers.
- Partitioning must provide load balancing and a configurable level of data redundancy to maintain the required level of data resiliency (usually, zero tolerance to data losses).
- As every node should maintain data management (I/O processing) at the proportional level, the scalability rate must be close to the linear and directly proportional to the number of nodes in a data grid cluster.
- As a logical outcome from the preceding points, the more nodes in the grid, the more resilient the fabric would be.
- Another logical outcome is that the number of served clients (task submitters) is proportional to the number of nodes in the grid. Proportion must be linear, of course.
- Clients (task submitters) must see this fabric through a unified interface, completely decoupling the client from the fabric's size/complexity.

- The fabric load balancer must work well with other load balancers in the environment.
- To increase resilience, each grid node server must back up a configurable amount of data from other servers.
- Load balancing and workload distribution is an essential part of the fabric's architecture, and they should not represent SPOF either. Thus, hub-and-spoke or single message, or the workload broker pattern is not applicable here. All dispatchers must be redundant and interconnected to provide maximum resilience.

As you may have already noticed, all these characteristics are properties of the Oracle Coherence product, which is based on the distributed Java cache specification and is essential to implement the last two use cases.

If we look at the SOA patterns catalog again, we will probably find only one pattern related to the requirements of this distributed fabric: Service Grid. Generally, it is related to the replication of service state deferrals, that is, the BPEL dehydration DB, which is highly important to replicate the following:

- The MDS store
- The SOA infrastructure dehydration store
- Audit and process cubes

Although it's highly important, it's only a small part of what Coherence can provide; this SOA pattern is only the tip of the iceberg as we have lots of service components patterns that are employed for maintaining a declared list of Coherence characteristics. In a similar way, you can see the Observer pattern as the low-level architectural pattern, supporting EDN at the service component level. Here, we have several patterns that support data caching, serialization, replicating, and processing. Before touching on them, let's briefly look at the roots of Coherence.

At the very basic level, Coherence uses the idea of a HashMap (`java.util`) as probably the fastest way of storing, sorting, and retrieving data with two main functions around the data object: `put(key,..)` and `get(key,value)`. So, as you can see, like in the previous chapter, once again we are dealing with key-value pairs that make the whole idea highly universal and suitable not only for EDN, but for all kinds of data implementation processes, for NoSQL in particular. We can combine multiple entry keys into logical storage units, so-called buckets.

Coherence takes this idea further by representing partitions that are stored on single or multiple cache servers. It also provides mechanisms for taking the key-value pairs of buckets/entry from the local cache and distributing them between the partitions. Thus, Coherence in general is a distributed implementation of `java.util.hashmap`.

How do we distribute objects? We had this technique long before SOA was developed (CORBA and RMI; look at the classic *Java Distributed Computing, Jim Farley, O'Reilly Media*), and this is the serialization we mentioned here a countless number of times. Well, you can say that serialization is quite a heavy process, and actually, we have confirmed it many times as well. It is slow and serialized objects can be large; how can we optimize memory utilization, especially considering the problems associated with the already mentioned `MapMessage` interface?

In addition to classic serialization, Coherence provides two additional extensions for object distribution:

- The `ExternalizableLite` interface with two main methods, `readExternal` and `writeExternal`. It performs a bit of data compression to optimize memory utilization.
- The other is **Portable Object Format (POF)**, which is a more advanced `ExternalizableLite` implementation.

Of course, compression comes with a cost; it requires some more processing power, and the implementation is not always simple; however, it is worth all the effort:

- POF supports the ability to automatically apply indexes to the classes. This ability solves one-third of the generic distributed cache tasks which includes indexing, Partitioning, and replication.
- POF supports interoperability between multiple languages, that is, it's not just Java anymore; you can use C# and some other languages, including .NET and C++. That's a *really big* advantage of Coherence.
- What's *even bigger* about Coherence POF is that it allows you to version data. It is hard to overstress this feature. If data grid is the fabric, stretched around at least three products (OSB, SOA Suite, and Oracle EP) and three frameworks (EBS, EBF, and ABCS), then we expect it to run 24/7 and not jeopardize but support our new deployments. POF supports multiple versions of the same object in the memory, and most importantly, these versions will be used by the same service without data conflicts.

The last advantage is possible because of the key-value nature of HashMap. Imagine that Version 1 of App1 uses Object Version 1 with tree fields/properties and App1 Version 2 has the same Object with four fields. When Object Version 2 is serialized, it will be presented with all the fields; however, when it is deserialized for Version 1 of App1 to provide updates in the old application, the extra pieces of data (fields 4, 5, and so on), POF will create the extra bucket as a byte array for extra fields in V2 and squeeze them into it without reading/parsing and then add it to the stream. When the Object is serialized again, this portion will be added back to Object Version 2, so we will maintain consistent backward compatibility.

So now let's see how Coherence addresses two other main fabrics' tasks: Partitioning and Distribution. Coherence supports several Distribution models, and the most obvious one is direct replication (fast read / slow write), which is the first thing that comes to mind. We must remember that for data/object consistency, all our replications must be synchronous. If we have two grid nodes with two replicated data objects each (objects are different on single node, but nodes are identical; this is a complete replication), we have to synchronize each object between the two nodes every time the objects get updated. The problem with this method becomes obvious when we move to more nodes and more objects. Synchronous objects' synchronization soon will consume all our HW resources.

Well, actually lots of Distributed Caches (fast write / slow read) work on that model. With Coherence, we have better options: the Partitioned Cache. Let's not keep all the objects in one node, but partition it. If we have four data objects, then let's split them equally between two nodes (`Obj1` and `Obj2` on `Node1`, which is the primary for these objects and `Obj3` and `Obj4` on `Node2` with the same rules). For resiliency backup, copies of objects 1 and 2 will be stored on `Node2` and vice versa. They will be synchronized when the master object data is updated. Thus, we considerably reduced the number of synchronous replications in the Coherence fabric.

When `Node1` goes down, `Node2` will be promoted as the primary for all objects, which is basically the first model with a single node. The extreme implementation of this method would be one node per single primary and backup object. Coherence allows you to configure this replication model according to your realities; it is always a trade-off between performance, resilience, and cost. The good news is that Coherence takes care of proxy layers between task submitters and task processors, and data indexing and internal buckets' synchronization. Of course, Coherence is also responsible for promoting backup nodes to the primary when the master node(s) become unavailable.

So, we have a highly performing Replicated Cache and very scalable Partitioned Cache. We have the third model that is devised to combine the best sides of both: Coherence Near Cache and the fastest possible access to MRU and MFU data. In this approach, every node has a local cache store of limited size and a large backend store. Imagine that a submitter is working with `Node2` and holding the master object `Obj2`; at a certain moment `Obj3` will be required. Coherence will transport `Obj3` from its master node (`Node3`, for instance) and put it into a smaller local cache (for example, MFU). If a service changes (invalidate) `Obj3` at its main location, its updated replica will be propagated to the local cache of `Node2`. Thus, the local cache keeps a local snapshot of the distributed information. There are several cache invalidation strategies that are completely configurable:

- If your business can tolerate a proportion of the objects becoming temporarily out of date, then you can use the fastest strategy known as Listen None strategy, ideal for data with a slow rate of change.
- If your most recently used data is also the most frequently used one, that is, your application is only interested in the data stored in the local cache, then Listen Present is the best strategy.
- Listen All strategy is the heaviest, but it covers all the possible scenarios in backend caches.
- If you do not know which of the last two strategies is best for you, choose Listen Auto; it will dynamically switch between Listen Present and Listen All based on the cache's statistics.

As you can see, Coherence provides a lot of functionalities and is probably the most advanced distributed cache system in the market. Apparently, just one SOA pattern mentioned in the catalog must be supported by a number of underlying patterns, actually implement indexing, Partitioning, and replication in various forms, suitable for all possible user scenarios.

Luckily, Coherence is not only one of the most complex Oracle products, but it is also arguably the best documented one, thanks to the Coherence Incubator Project (`https://java.net/projects/cohinc/`) and to all who contributed to it, providing practical examples of various implementation patterns. We urge you to read and try the published materials. Here, we will just mention the basic patterns that are available for implementation, as the title of this book requires.

The following table includes Coherence Object Distribution patterns:

Pattern	**Problem**	**Implementation**
The Messaging pattern	We need an absolutely reliable SPOF and ultrafast store-and-forward messaging network without a central hub (as in Hub-and-Spoke) as a potential bottleneck.	Coherence from the moment of its creation has been a message distribution system that is capable of supporting queues and topics. The advantage of this pattern implementation is that in Coherence, you do not need to wrap objects into messages, that is, you can skip serialization operations; you use the same distributed infrastructure around your applications without establishing a new one. Although the main purpose of Coherence is different than just being a distributed hubless SAF, some of us who experienced troubles with the original WLS will agree that SAF greatly appreciates this pattern implementation. Old JMSes and AQs can definitely perform the same task cheaply in all ways; still, this pattern is the foundation of the next one. One word of caution, which is common for all object distribution patterns: plan the distribution path carefully between your nodes and place the aggregation nodes close to the processing nodes for better throughput.

Pattern	Problem	Implementation
The Event Distribution pattern (EDN)	If you want something done fast, do it yourself and do it in cache. If your service component can put the updated object into a distributed cache, then it would be the fastest way possible to propagate the event notification. Thus, the need for cache in events processing is justified.	Yes, Coherence is the cache we need and the change of objects' state will not go unnoticed with a properly arranged invalidation strategy (if you do not know which one to use, go for Listen Auto). Remember the last figure from the previous paragraph? Coherence is the closest companion of Oracle Event Processing and can provide the following: • Five types of channels for files, errors, and all types of caches, both local and distributed. • Guaranteed delivery of events. No events losses. This pattern sets the necessary infrastructure for the next pattern in line.
The Push Replication pattern	How can you keep all the objects in all the Grid nodes synchronized in the most fastest and reliable way? How can you propagate the changes from one cluster to another?	As the name suggests, data is pushed from the place where changes occurred to all the nodes that are configured to be notified. This is a bit more than a pattern; it's an entire framework within Coherence that provides the following types of object pushes: • active-passive • active-active • hub-and-spoke • multi-master • centralized replication We touched upon some of these aspects while discussing the three possible distribution models and ways of data invalidation.

The following table includes Coherence Object Processing patterns:

Pattern	Problem	Implementation
The Command pattern	An operation on a certain object that should be executed possibly several times (or none) must be represented as an executable object called Command in the context of the object's data. Command must clearly provide a method for executing the task.	This is a distributed version of the classic Command pattern, and the code presented in the incubator is quite self explanatory. The executable method provided by the Command interface simply called `execute`, which requires a single parameter: `ExecutionEnvironment`. This is where the context is encapsulated, and Context Manager is responsible for maintaining it. To be distributable and replicable within the data grid, context and command objects must be serializable (`ExternalizableLite` or `PortableObject` would be even better). This pattern does not always return a value back to the task submitter (client). If a return value is expected, the Functor pattern must be allied. This pattern is an extension of Command and should be omitted for brevity.
Processing pattern	The previous pattern (Command) is responsible for creating an executable object based on the data object's context (as depicted in the 3D composite event matrix's figure, for instance), but this is just a prerequisite (although critical) for distributed computing. We must make sure that the client will see the fabric as one big computer with a unified way to submit, start, pause, and resume the predefined task with the associated data object. This processing framework must seamlessly support not only computers assembled into the Coherence grid, but also the ones that are plugged using Coherence Extend (TCP or JMS connectors to remote systems).	Coherence as a virtually unlimited distributed network of data storage buckets and associated data processing nodes represents the possibility of asynchronously processing (almost) everything that is called or run by Java. This pattern allows the client to use the fire-and-forget submission model for the predefined object (with the data structure declared in the key-value form and standard executable method). This pattern is capable of reporting back to the sender about the task submission's outcome to keep the submitter informed about the progress of the execution. Based on the status (or even without it), the submitter has the ability of canceling the processing if submission doesn't reach the final stage of processing. Due to the asynchronous nature of the execution, the client has the ability to disconnect from the processing grid and connect later to collect the results or even delegate it to another client. The last option is possible because the client will decide on the ID that is associated with submission.

There are more patterns in the incubator (`https://java.net/projects/cohinc/`), including Coherence Commons (similar to Apache Commons) that holds lots of useful utilities. However, of all the patterns, the Processing pattern is arguably of paramount importance and is available for implementation on Coherence. This pattern is so full of functionalities and covers so many requirements that sometimes it is very rightfully called a framework. This pattern will require the following components:

- Submission dispatchers, polling the individual submission from local submission caches and registering them on Dispatch Controller, which in turn passes this registration to logging.
- A registered and logged task passed to the task dispatcher. The client is notified about the submission's outcome.
- The task dispatcher dispatches the task (executable submissions with the implementation of the `Task` interface) to the Task processor. Here, we can have different variations of the tasks. If a submission contains Java runnables and callables, it will be executed locally by `LocalDispatcher`, but this is not desirable due to limited distribution capabilities. The task can also be resumable, which means that it can be suspended during the execution.
- `TaskProcessor` is the actual worker that executes the task. `TaskProcessor` is assigned according to `TaskDispatchPolicies`, which could be simple Round-robin or Attribute Matching. The last type is purely content-based routing/mediation.
- The submission's result is returned to the `SubmissionResult` cache.

Although very schematic, this sequence is quite complete to recognize the resemblance with patterns we implemented using SOA Suite in *Chapter 3*, *Building the Core – Enterprise Business Flows*; please see the figure with the block diagram for Composition Controller with Business Delegate and Service Locator. Exactly as with Service Broker / Business Delegate patterns, follow the ensuing steps:

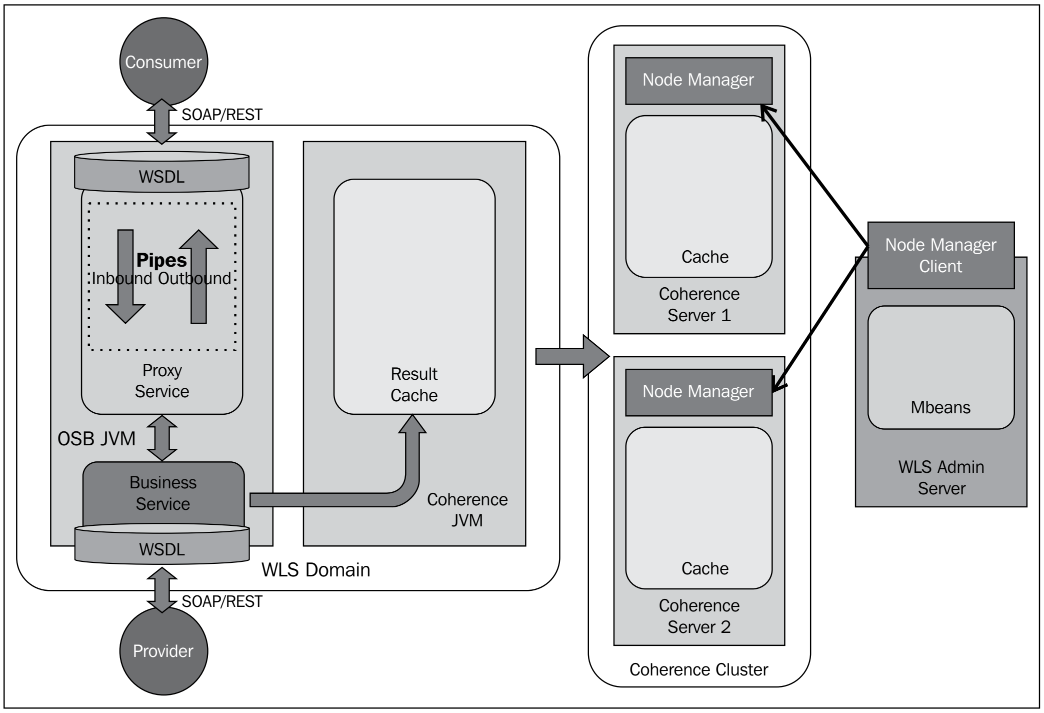
1. First of all, we decouple the consumer and actual worker.
2. We use the dynamic execution capability (here in Java).
3. We can postpone the execution until a certain event has occurred.
4. We can route our object conditionally or unconditionally to the predefined `TaskProcessor` (Service) and we use an external policy for the Task-Process association.

Coherence empowers these patterns with the highest resiliency and top processing speed. Its distribution model is based on event mappings and notifications, and it fully supports multithreading. All of this makes it a perfect companion for Oracle EPN.

Here, we would like to discuss two most common types of Coherence implementations, and we will traditionally start with the classic OSB and Coherence integration.

Coherence and OSB

In simpler cases, if our business service returns something close to static or if, as we mentioned above, our service consumer can tolerate the not really up-to-date result object, and if in addition to that (optionally), service operations are quite expensive, then we can configure our business service to cache results.



Cached results can be configured for **time-to-live (TTL)**, so after the cache result expires, the next call will result in updating the cache with a fresh result by executing the business service. The cached result is stored on a separate Coherence JVM, so it does not take any resources from OSB JVM and can be distributed/partitioned between the Coherence nodes. Although these machines are completely separated, WebLogic Administration Server can still control Coherence servers on individual machines using its own node manager client, which is connected to individual node managers on each Coherence machine.

To establish this integration, you can follow the ensuing steps:

1. Configure startup arguments in the server's **Start** tab as described in the OSB documentation (`http://docs.oracle.com/cd/E17904_01/doc.1111/e15867/configuringandusingservices.htm#OSBAG1413`), and best practices at `http://docs.oracle.com/cd/E14526_01/coh.350/e14510/bestpractices.htm`.
2. Now we have to configure the business service for caching in Coherence. Go to **Advanced Properties** in **Message Handling configuration** and check **Supported** for **Result Caching**. Also set **expiration time** at the bottom of the page. If nothing special is part of your requirements, you can set it according to the default setting. You can also preset the fixed interval or establish something that is more business-oriented using the XPath expression, which is linked to expiration timestamps in your SBDH-compliant message header.
3. As you remember, distributed cache is a partition of key-value pairs. So, we need to link the cached data to our object key, which could be a part of our Message Header, that is, `$body/*:CTUMessage/*:MessageHeader/`. Put your XPath expression in the **Cache Token Expression** field.

Using this configuration, you can considerably reduce the number of business service invocations and provide the output much faster.

Coherence and event processing

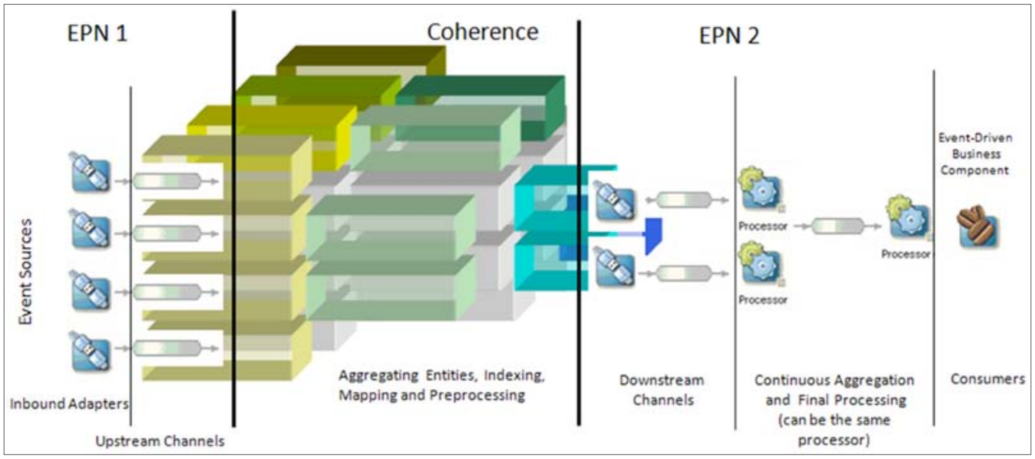
Now, we will approach a more complex realization that is related to CEP. So, we can move the data object whenever and wherever we want to, and we can execute the methods, thanks to the Coherence Processing pattern. However, being essentially a distributed cache, Coherence is not really at its best when it comes to continuous aggregation, that is, accumulating and filtering data for an extended period of time. It will be quite right to call Coherence processing stateless, and that's logical as it will be quite difficult to replicate and partition stateful object/processing nodes. Yes, Coherence has `EntityAggregators`, which is built-in and customizable, but their main purpose is to reduce the number of entities in the cache, combining and aggregating the identical ones (by their attributes) in order to reduce the cache size and provide the result. Aggregators are constantly updating entities, but not in a time-based aggregation manner. Also, we must keep in mind the limited ways of accessing distributed cache, that is, we do not have adapters in the common sense; that's not the purpose of Coherence.

Event Processing at the same time is very good for time-based continuous aggregation of different streams; aggregations can be incrementally evaluated and grouped by values. We have quite an efficient adapter framework with abilities to implement custom adapters within adapter factories. The downside of the Oracle EDN is that we have no way to make our continuous aggregation parallel. Something that is going on in an isolated JVM for quite some time cannot be really replicated unless we perform the same operations exactly in another JVM. Operations are highly stateful. If we fail, we will have to start all over again, and all the previous results are wasted (if we are processing using CEP alone).

The natural approach here would be the combining benefits of both products in one form of MapReduce patterns (read `http://www.cs.stanford.edu/people/ang//papers/nips06-mapreducemulticore.pdf` and `http://highlyscalable.wordpress.com/2012/02/01/mapreduce-patterns/`; these articles are really good), where roles will be distributed as follows:

- Coherence (Map):
    - Entity aggregation using key affinity/no-entity duplicates
    - Entity indexing
    - Maintaining the highest resilience through Partitioning and backup
    - Supporting partition transactions and preparing results for the final aggregation
    - A handy publishing mechanism

- Oracle Event Processing (Reduce):
    - Continuous aggregation and grouping
    - CQL provides great flexibility for adjustments and declarative modifications
    - Coherence provides some sort of consistent entity views; therefore, if we fail, we can restart our analysis again without losing events


EPN 1
Coherence
EPN 2
Event Sources
Inbound Adapters
Upstream Channels
Aggregating Entities, Indexing, Mapping and Preprocessing
Downstream Channels
Processor
Processor
Processor
Continuous Aggregation and Final Processing (can be the same processor)
Event-Driven Business Component
Consumers

Surely, the presented implementation types (the second one is very schematic; please see the Oracle documentation for details on implementation, `http://docs.oracle.com/cd/E14571_01/doc.1111/e14301/cache.htm`) are not the only ones possible in OFM. Coherence can be highly beneficial for caching Oracle DB data (as any data). Coherence also provides a mechanism called CohQL to query the cache, SQL-style. In other words, we have plenty of means to implement the data grid in our SOA infrastructure around all the runtime frameworks with embedded tools for integration and data manipulation.

Monitoring service activities

Moving down the line of SOA patterns that are used in core technology frameworks, we finally come to Business Activity Monitoring; this term was proposed by Gartner during the earlier years of SOA development. Actually, this term has no relation to any recognized SOA pattern (with the exception of **UI Mediator**, but very remotely), but every Oracle-technology-related book (related or not related to SOA) has a chapter dedicated to BAM.

You can find them a lot in chapter 19 of the book by Lucas Jellema, which has already been recommended. It's quite typical that you will find guidance on BAM somewhere around the final chapters of this book. Quite interestingly, it contradicts the common corporate purchase practice; usually, BAM is purchased first and implemented in the end.

Why is that? It's quite obvious; top managers love the dashboards full of colorful 3D pie charts, but development teams have to go a very long way before something really useful becomes available for visualization, and that has nothing to do with BAM. Those of us who remember the earlier versions of BAM could also be rather hesitant. It was built on the Microsoft platform with lots of bugs and integration problems with core Oracle SOA products, and worked with the IE browser only. It does not always leave happy memories (and we are not discussing the license costs compared to other products here).

Oracle's early BAM problems and dependence on Microsoft have long gone, and now if you are contemplating using a monitoring tool for your SOA/BI portfolio, the Oracle BAM can be a good alternative to Nagios, for instance. Again, as its main purpose is very straightforward, that is, to construct reports and visualize them in a push-based manner (no manual refreshes should be required), it does not provide any pattern as a solution to any problem; however, it is arguably the best candidate to complete the extremely powerful triad:

- Complex Event Processing on Oracle EPN
- Events/message distribution, aggregation, and reliable replication by Oracle Coherence
- Business events' reporting by Oracle BAM

Frankly, if any of your business cases is similar to the banking, logistic, or telecom examples discussed earlier, this triad with all the available patterns will cover most of your needs (or all in our experience). BAM in this case from the earliest 11*g* build provides a web-based graphical environment for establishing a completely codeless implementation of reports and dashboards to monitor different KPIs, primarily business-related ones. This high usability, together with the fact that there are no complex SOA patterns associated with this product alone, simplifies our task by presenting this functionality in this paragraph, as we do not have room here for providing drag-and-drop cookbook screenshot demonstrations.

If you return to the figure that presents the WLS SOA console in OFM's introductory chapter for a moment, you will see that simplicity starts with the combined installation of SOA Suite, where the BAM server will be installed in one go, with the whole bundle (of course, you can exclude it from the installation).

It runs on its own WebLogic server (in our case, `bam_server1` by default), and it is controlled from the same administration console/OFM control in the same way as other servers are controlled. The Oracle BAM server is a collection of the components Oracle BAM Active Data Cache (Oracle BAM ADC), Oracle BAM Report Cache, Oracle BAM Enterprise Message Sources (EMS), and Oracle BAM Event Engine. All BAM object definitions and reports will be stored in the OFM DB repository, created during the installation. As long as object data is involved, we have two sides of BAM connectivity; naturally, it has APIs for data producers (SOAP and RMI) and data consumers, who will use the BAM DB schema, like with ODI. In fact, Oracle Data Integrator can be both; a data provider for reporting and data consumer of BI data.

When you start `bam_server` and log in to the BAM web application, `http://<your_BAM_host>:<your_BAM_port>/OracleBAM/default.htm`, you will see four main buttons that lead to the console responsible for all aspects of the BAM dashboard's life cycle. The architect's console is the most interesting to us, as its name denotes. The architect's console is responsible for creating BAM data objects on which all our reports will be based. Every type of BAM and OFM integration will start from this console. Let's quickly walk through the possible BAM integration use cases.

Direct integration of BAM and BPEL

The direct integration of BAM and BPEL is the most common use case and also the most frequently demonstrated one in any SOA/OFM cookbook. It is also the most ineffective rule-breaking implementation as it directly violates the Loose Coupling SOA principle. Let's see why. First, let's look at *Chapter 3*, *Building the Core – Enterprise Business Flows*, where we discussed the SCA Service Broker and were particular about declaring a comprehensive process name at the beginning of the BPEL flow. The next step should be to assign the runtime variables, including the process name. If we need to monitor SCA BPEL instances in BAM, it would be best to inject BPEL Sensor (`http://docs.oracle.com/cd/E28280_01/dev.1111/e10224/bp_sensors.htm`). To complete this part, we need to perform the following steps:

1. Check our connection with the BAM server. If we do not have any, it is time to create one: **File** | **New** | **Connections** | **BAM Connections**.
2. Go to the BAM Architect and declare the data object that we need to populate for our report (`http://docs.oracle.com/cd/E21764_01/integration.1111/e10224/bam_data_objects.htm`). Different types of data objects are explained at `http://docs.oracle.com/cd/E25054_01/dev.1111/e10224/bam_adapter.htm#BABHCHCE`.
3. Back in SCA, in the BPEL structure console, locate the node for sensor variables and create a BAM sensor, give it an appropriate name, and set the target variable.

After this, we are ready to configure the sensor's actions. In the **Structure** window, select the sensor's actions and create a new one for BAM when the **Sensor Action** window appears. Then, set the action name and select `BAM DO` created in the BAM architect, and also provide the operations (`Insert`) and keys. XSLT mapping is available at this stage for adjusting BPEL action data sources to `BAM DO`.

Technically speaking, this is it; we are done with the integration. In the next few steps, you will deploy the application, test it, and verify that instances of data objects are pumping into BAM. Finally, in the BAM active studio, you can assemble a beautiful report with a pie chart and everything. The main point here is that we have a direct connection between BPEL and BAM, and although it is perfectly operational, it is not exactly what we want in production, especially for events monitoring.

BAM gives us several other options for data feed, and you can spot some of them right away in the BAM Architect. Before you create a new data object, please look at the drop-down list at the top. In addition to the first option discussed earlier, we have Enterprise Message Sources, External Data Sources, and Alerts. The first one denotes JMS, which is extremely versatile; however, in addition to listening to JMS, we have a synchronous way of getting data to BAM Active Data Cache and BAM Web Services. Thus, we have many ways to get data for dashboards, including a completely vendor-neutral way.

The BAM and JMS connection

So, how can we set up JMS as a BAM data channel? Before creating any sensor, please proceed to the SOA Domain Admin console and create Connection Factory and a new Queue (`BAMSensorQueue`) in SOAJMSModule (in WLS Domain Structure, go to **Services | Messaging | JMS Modules**. Then, create a new JMS System Module Resource, set the JNDI name as `jms/BAMConnectionFactory`, and so on). To create a new sensor for its actions, set **Publish Type** as **JMS Queue**. The steps for creating a sensor variable are similar to the previous example. After deployment and testing using either the SoapUI or EM console, you will find the message in the previously configured JMS. Now, you can return to the BAM Architect and create a new object according to your input (it could be the same as the previous one for test purposes), but now select **Enterprise Message Sources** from the list on the top. Set the correct values in the EMS section: **JMS Message Type** as **Text**, **Operation** as **Insert** (actually, you can perform all the CRUD operations on your objects); then, set the JNDI and queue names as you created them in WLS.

For more administrative details that are not part of the SOA patterns, please refer to `http://docs.oracle.com/cd/E23943_01/dev.1111/e10224/bam_ent_msg_sources.htm`. Here, you will find that Oracle Messaging is supported, and the sources could be JMS providers from IBM, Apache ActiveMQ, and Tibco.

Summarizing all of the preceding points, we can see three main advantages of this connectivity method when compared to the previous one:

- Vendor neutrality (as long as we can use JMS)
- Complete decoupling between the provider and consumer, allowing BAM to balance its processing workload more effectively
- Asynchronous message exchange allows us to use relatively big objects

Advanced XML formatting in EMS will allow you to map virtually any JMS payload to your data object, but here we would like to advise you to avoid excessive transformations and even simplify (flatten or remove the hierarchy) your messages for better performance and reports aggregation. Really, in most of the cases, you need just a bunch of fields to satisfy the imaginative taste of your managers. Surely, you can also log all the failed messages. If an asynchronous exchange pattern is not suitable for your needs, you can employ the `WebService` API.

BAM and the webservice API

BAM provides a set of **web services (WS)** for a synchronous data feed:

- `http://host_name:7001/OracleBAMWS/WebServices/DataObjectOperationsByID?WSDL`
- `http://host_name:7001/OracleBAMWS/Services/DataObject/DataObjectOperations.asmx?WSDL`
- `http://host_name:7001/OracleBAMWS/WebServices/DataObjectOperationsByName?WSDL`

Here, 7001 is a port by default. For testing, you can start with the first one, `DataObjectOperationsByID` (without WSDL), by opening it in your browser. The rules of invocation are fairly simple, but strict; if you do not want to get 401 or 403 as a response, please follow them and also refer to the documentation:

- The names of elements in your payload that you paste into the XML payload should be exactly as the field of your data object
- The SOAP action can be from basic CRUD (see on the top), but formatted according to the namespace as `http://xmlns.oracle.com/bam/insert`
- HTTP authentication must be enabled, and you should provide your OFM username/password

Just press **invoke**, and see what you will get in the BAM Architect in Active Data Cache. You can perform these kinds of tests without a data provider, just by using a WS test page.

Finally, you can refer to the figure from the *EDN in SOA the stack – a practitioner's approach* section and replace the Event-Driven Business component with BAM, which is connected to EDN using JMS. This will conclude this block diagram evolution and this section.

SOA as a cloud foundation

We would like to conclude this book in the same way we started it, presenting a roadmap for the implementation of SOA patterns / SOA standards, but now, we will try to link SOA and Cloud patterns in order to see the dependencies, which is important for practical implementation. Exactly as in *Chapter 1, SOA Ecosystem – Interconnected Principles, Patterns, and Frameworks*, we do not intend to show all patterns' relations (at the time of writing this book, in the patterns catalog, we have 39 Design and 13 Compound Cloud patterns), but only those that support the main subject of this book: (Agnostic) Composition Controllers, as enablers of the Composability principle. In the case of cloud, following this particular SOA principle is not enough to fulfil cloud's promises. SOA is just one of the cloud enablers, although an essential one. Other enablers are as follows:

- Virtualization (literally, of any resources, namely, network, OS, service, DB, and so on)
- Grid computing (Oracle covers this)
- Clustering technology (Oracle covers this)

What are these cloud promises? Exactly as in SOA's case, it refers to money but now with a capital "M", and again as 14 years ago, we (or some of us) are caught in the same love-hate cycle. In addition to shortening the delivery cycle and reducing operational costs, heralds of "Mighty Cloud" declared the era of operating income boost through extended business opportunities harvesting, based on Fast Event Processing on Big Data (that is a big opportunity), and so on. Are these promises hollow? Not at all. Are they all achievable? Partly. Can they all be fulfilled today? Hardly (or simply put, no! Sorry). What is the Oracle contribution to it? Well, some say that Oracle is lagging behind the leading Cloud providers. Maybe it seems so, but we must bear in mind that Oracle's cloud approach is probably the most overwhelming and therefore the most complex approach for implementation.

Don't get me wrong; a relatively simple remote file storage is an absolutely valid form of cloud provisioning (please look for PaaS, IaaS, SaaS, and other *aaS Cloud Delivery models in any sources; *The Cloud Computing: Concepts, Technology & Architecture, Thomas Erl* could be a good start). The hosting companies that are reliably providing us with remote computing resources have been around for quite a while, even before the invention of the SOA term.

Now some can call it "Private Cloud", and the line between the cloud and remote datacenter is really thin (both of them can store data remotely, provide computing resources, virtualize them, and are both accessible via the Internet. VPN could apply). So what are the distinctive cloud promises?

- Generally, computing resources today are vast but unevenly loaded. At the same time, speaking about a single enterprise, IT resources (SW and HW) are usually provisioned according to average workload estimates (one/five year forecasts). The Average Order Handling system designed for 25K orders daily will have a hard time during peak loads (100K after an aggressive advertising campaign), so the SLA will be broken with rather unpleasant consequences. It would be nice to have the possibility to delegate the processing dynamically to order service instances available on remote resources.
- Following the logic presented in the preceding point, the management can reconsider the resource allocation estimation model, setting it for the next period not average but minimal requirements for on-premise application farm. The application farm on cloud will be allocated gradually, depending on the current requirements. Internal resources will keep business knowledge in-house, and HW expenses will be replaced by monthly fees with the Pay-As-You-Go option.
- Just scaling our service-bound resources (horizontally mostly, but vertically as well) between on-premise and cloud is just one part of dynamic resource provisioning/allocation. A similar mechanism should exist inside the cloud and between clouds, if one cloud is not enough. This flexible provisioning and ability to maintain redundant implementations considerably improves HA.

In addition to runtime dynamic resources provisioning and (re)allocation, we expect the following:

- Test/development environments' provisioning are always a headache for non-IT companies (actually, for IT as well). We need at least three of them per Prod, where the last one, Operation Readiness Test, must be equal to production. If provided too early, they will waste resources, and if too late, they will affect the quality of our tests. We must have the ability to choose what configuration we want to install (better, and in a friendly way) and access the resources automatically in a matter of minutes (in complex cases, a couple of hours).
- Resource provisioning should be rapid, and relocation (local2cloud or cloud2cloud) should be as simple (to us) as copying a file. This service relocation must be non-disruptive, which means that service consumers should not notice the relocation of production resources.

Talking about service and service runtime environment, we would like to stress the fact that we are *not* discussing the reallocation of a synchronous `*.jar` service file from one `jboss/deploy` folder to another. Let's get back to *Chapter 8*, *Taking Care – Error Handling*, to the figure under the *Error-handling design rules* section; we would like to see the relocation of a task-orchestrated service in the context of all SOA runtime frameworks, and that's not a small thing. Even replication of a VM with a complete OFM installed is a bit bigger than just copy and paste.

> Here, we are discussing cloud patterns in direct relation to the Oracle SOA technology stack. Supporting very important patterns such as Elastic Network Capacity, Elastic Disc Provisioning, Bare-Metal Provisioning, and so on, are out of the scope of this chapter, as they are lying either on Resource Abstraction and Control Layers or the Physical Resource Layer. Cloud ecosystems are based on atomic and compound patterns and mechanisms, which are a bit broader than patterns and act as a technology-centric foundation for the patterns' applications.

How can these cloud undertakings be satisfied? From the bottom to the top, number five is quite difficult to implement even within the same vendor's environment, but virtually impossible between different cloud providers. Yes, vendor lock-in in the case of cloud is one of the biggest risks and the more stateful services we have across our SOA Frameworks, the more difficult it would be to establish Non-Disruptive Resource Relocation. This promise will be fulfilled when we will be able to move our most complex task-orchestrated services running in production from one cloud provider to another in a matter of hours, not months. Alas, at the moment of writing this book, we are far from it.

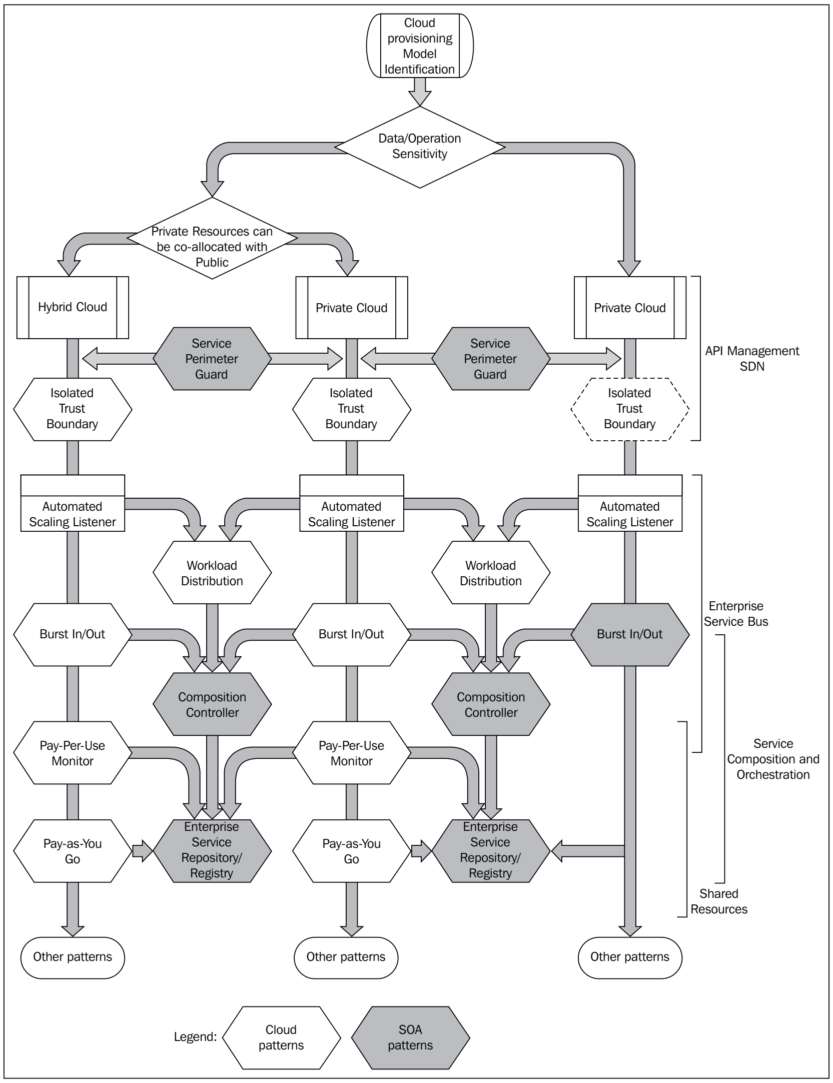
Number four, Rapid Resource Provisioning, is generally tamed by Oracle for DB, Coherence, and EDN environments across all the required frameworks by creating complete HW and SW ecosystems as preintegrated Exadata, Exalogic, and Exalytics with lots of Cloud/SOA supporting appliances (`http://www.oracle.com/us/products/engineered-systems/index.html`). At last, Sun Microsystems together with Oracle made John Gage's prediction true; it was quoted in 1984 that "the network is the computer". In our opinion, it was a truly brilliant decision to combine such an inventive force into one power, but some can say that it was an act of desperation as well. Considering the number of products in the OFM bundle (*Chapter 2*, *An Introduction to Oracle Fusion – a Solid Foundation for Service Inventory*) and the complexity of multilayering a configuration (partly discussed in *Chapter 8*, *Taking Care – Error Handling*); it was just natural to provide a preconfigured that is tuned for best-performance platforms for enterprises with a shortage of skillful IT personnel.

It comes at a cost, of course, but here is an opportunity to deploy these engineered systems in cloud data centers and provide multitenant access with the Pay-As-You-Go option. Oracle sends a clear message; investments in the Global Cloud Infrastructure will be expanded in the coming years (November 2013). These intentions are supported by the following facts:

- The Cloud Multitenant Environment Compound pattern is based on resource sharing, pooling, and reservation and Application Server side Pooling and Reservation are quite well covered by WLS clusters and work managers. Isolated Trust Boundaries is another pattern that supports this environment, and it can partly rely on the Oracle Service (API) Gateway, partly because its main purpose is to provide AAA operations and other SOA Security patterns discussed in *Chapter 7*, *Gotcha! Implementing Security Layers*, at the service level, but not the network. Most importantly, Oracle DB 12*c* Enterprise Edition has now introduced a new multitenant architecture, supporting DB resources' sharing and consolidation.
- In addition to its own SDN development efforts, in January 2014, Oracle acquired Corente for getting into software-defined networking. Corente's products include Cloud Services Exchange, which establishes trusted network services between public or private cloud data centers and any location over any IP network (according to a press release). This is a very strong move towards a complete implementation of the Isolated Trust Boundaries pattern and establishment of flexible and high-performing VM-2-VM networks. Technically, it concludes all the necessary prerequisites for complete SOA Platform virtualization.

To ensure that the implementation of the cloud is right on the money (see the next figure), SOA, as one of the enablers, must heavily contribute to Autonomy, Abstraction, and Loose Coupling of the application resources that are deployed on the cloud platform. As mentioned, task-orchestrated services have to be redundantly predeployed for automatic scaling, as the runtime non-disruptive relocation could be rather complicated. However, if we can assemble a complex composition at runtime using Composition controllers (Service Brokers) from *Chapter 3*, *Building the Core – Enterprise Business Flows*, and *Chapter 4*, *From Traditional Integration to Composition – Enterprise Business Services*, this task will be quite attainable. Individual composition members, entity and utility, and even reasonably sized atomic task services can be provisioned dynamically from clouds and between clouds. Composition controller, which is positioned on-premise and combined with the HW Load balancer can effectively distribute the service workload between local and cloud resources; it is implemented in the cloud and can be part of cloud balancing and Burst In/Out compound patterns for SCA/OSB resources (still, HW balancers should be considered first for atomic resources).

The implementation of Coherence will not require a dedicated Composition Controller as a processing pattern can help you reach the required level of grid distribution between the cloud and local infrastructure. Service Perimeter Guard, which is essential for all cloud delivery models, will help you establish the Isolated Trust Boundaries for multitenant environments. The last one is conditional for a private cloud, but it must be planned carefully anyway, as you should establish cross-domain (or department) separation for the same enterprise. By the way, the Oracle API gateway is quite good for throttling and balancing the workload as well, but again, HW LB is more suitable for balancing.



Cloud and SOA Patterns implementation roadmap

Dynamic Service Brokering, Composition Controlling, and Service Resource metering would not be possible without Enterprise Repository, and the realization of Oracle SR for a Fusion application hosted on Oracle Cloud (`http://fusionappsoer.oracle.com`) should be the starting point for any integration/interoperability activities between Oracle Cloud and your local SOA platform. There you can obtain a preconfigured Cloud Service WSDL (for instance for the order, select ADF Service for the Type, and search for the orders. Select **Purchase Order**, and look at the bottom of the **Detail** tab for WSDL). Well, we got WSDL and all XSDs. We also have out-of-the-box security policies (`wss_username_token_over_ssl_client_policy`) that we can extend or create on our own, and we have full SAML support. So, now we can go to JDeveloper and create any complex process using this information in Partner Links. This would be the fastest way to establish service interoperability on Oracle Cloud and later employ them in on-premise Dynamic Composition controllers.

Summary

Complex Event Processing on Event-Driven Networks, In-Memory grid, and parallel processing with the following entity aggregation are not isolated standalone technologies that can be evaluated versus / instead of a service-oriented approach but parts and supporting blocks of SOA in a broad prospective. Similar to the cloud methodology and mechanisms, the Cloud Service / Resource Broker is not someone who is trying to upsell you the remote hosting services, which are provided by the company two blocks away.

SOA and Cloud are quite interconnected, complementing each other with atomic and transportable units of work at one side (genuine SOA) and virtually unlimited distributed computing resources (Service Grid and Canonical Resources SOA patterns, which are provided by cloud) at another.

With recent developments, Oracle can practically provide all the necessary building blocks for all implementation tiers and frameworks, both on-premise and offsite. Complex silo-like applications, such as Salesforce, are now available on cloud. They can be easily integrated with the local Oracle SOA Suite, or clouds from other vendors can be linked to Oracle Cloud Farm with not much effort. This approach will allow you to boost your processing right away without making any significant investments in your local infrastructure, paying as you go. For those who already have corporate data centers with service infrastructures of any level of maturity, a more gradual pattern-based approach would be wiser. Just remember where we started; each pattern is the answer to a particular, repeatable, and identifiable problem. Identify your problem first.

Index

Thank you for buying
Applied SOA Patterns on the Oracle Platform

About Packt Publishing

Packt, pronounced 'packed', published its first book "Mastering phpMyAdmin for Effective MySQL Management" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike.
For more information, please visit our website: `www.packtpub.com`.

About Packt Enterprise

In 2010, Packt launched two new brands, Packt Enterprise and Packt Open Source, in order to continue its focus on specialization. This book is part of the Packt Enterprise brand, home to books published on enterprise software – software created by major vendors, including (but not limited to) IBM, Microsoft and Oracle, often for use in other corporations. Its titles will offer information relevant to a range of users of this software, including administrators, developers, architects, and end users.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

[PACKT] enterprise
PUBLISHING
professional expertise distilled

Professional Expertise Distilled
Oracle SOA Governance 11g Implementation
Successfully implement SOA governance using Oracle SOA Governance Suite 11g with the help of practical examples and real-world use cases
Luis Augusto Weir
Andrew Bell
[PACKT] enterprise

Oracle SOA Governance 11*g* Implementation

ISBN: 978-1-84968-908-3 Paperback: 440 pages

Successfully implement SOA governance using Oracle SOA Governance Suite 11*g* with the help of practical examples and real-world use cases

1. Understand SOA governance including its key concepts, goals, and objectives, and how to implement these using the Oracle SOA Governance Suite.

2. Execute an SOA maturity assessment in order to capture the SOA governance challenges specific to your organization.

Quick answers to common problems
Oracle SOA Suite 11g Developer's Cookbook
Over 65 high-level recipes for extending your Oracle SOA applications and enhancing your skills with expert tips and tricks for developers
RUBICON RED
Strategy. Governance. Education.
Antony Reynolds
Matt Wright
[PACKT] enterprise

Oracle SOA Suite 11*g* Developer's Cookbook

ISBN: 978-1-84968-388-3 Paperback: 346 pages

Over 65 high-level recipes for extending your Oracle SOA applications and enhancing your skills with expert tips and tricks for developers

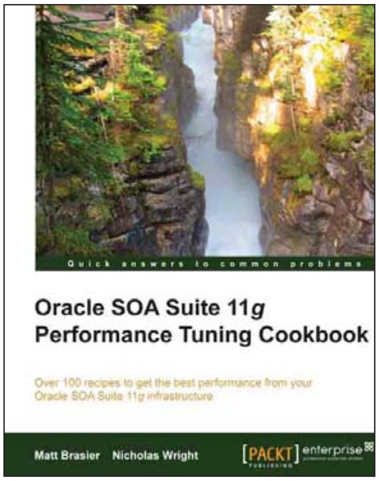
1. Extend and enhance the tricks in your Oracle SOA Suite developer arsenal with expert tips and best practices.

2. Get to grips with Java integration, OSB message patterns, SOA Clusters, and much more in this book and eBook.

3. A practical Cookbook packed with recipes for achieving the most important SOA Suite tasks for developers.

Please check **www.PacktPub.com** for information on our titles

[PACKT] enterprise
professional expertise distilled
PUBLISHING

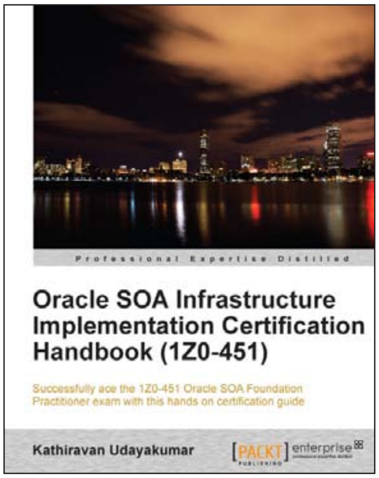

Quick answers to common problems
Oracle SOA Suite 11g
Performance Tuning Cookbook
Over 100 recipes to get the best performance from your Oracle SOA Suite 11g infrastructure
Matt Brasier Nicholas Wright
[PACKT] enterprise

Oracle SOA Suite 11*g* Performance Tuning Cookbook

ISBN: 978-1-84968-884-0 Paperback: 328 pages

Over 100 recipes to get the best performance from your Oracle SOA Suite 11*g* infrastructure

1. Tune the Java Virtual Machine to get the best out of the underlying platform.
2. Learn how to monitor and profile your Oracle SOA Suite applications.
3. Discover how to design and deploy your application for high-performance scenarios.
4. Identify and resolve performance bottlenecks in your Oracle SOA Suite infrastructure.


Professional Expertise Distilled
Oracle SOA Infrastructure
Implementation Certification
Handbook (1Z0-451)
Successfully ace the 1Z0-451 Oracle SOA Foundation Practitioner exam with this hands on certification guide
Kathiravan Udayakumar
[PACKT] enterprise

Oracle SOA Infrastructure Implementation Certification Handbook (1Z0-451)

ISBN: 978-1-84968-340-1 Paperback: 372 pages

Successfully ace the 1ZO-451 Oracle SOA Foundation Practitioner exam with this hands on certification guide

1. Successfully clear the first stepping stone towards becoming an Oracle Service Oriented Architecture Infrastructure Implementation Certified Expert.
2. The only book available to guide you through the prescribed syllabus for the 1Z0-451 Oracle SOA Foundation Practitioner exam.

Please check **www.PacktPub.com** for information on our titles