O'REILLY®

Building Software Teams

TEN BEST PRACTICES FOR EFFECTIVE SOFTWARE DEVELOPMENT

Joost Visser

www.ebook3000.com

O'REILLY®

Building Software Teams

Why does poor software quality continue to plague enterprises of all sizes in all industries? Part of the problem lies with the process, rather than individual developers. This practical guide provides ten best practices to help team leaders create an effective working environment through key adjustments to their development process.

As a follow-up to their popular book, *Building Maintainable Software*, consultants with the Software Improvement Group (SIG) offer critical lessons based on their regular assessment of development processes used by hundreds of software teams. Each practice includes examples of goal setting to help you choose the right metrics for your team.

- Achieve development goals by determining meaningful metrics with the Goal-Question-Metric approach
- Translate those goals to a verifiable *Definition of Done*
- Manage code versions for consistent and predictable modification
- Control separate environments for each stage in the development pipeline
- Automate tests as much as possible and steer their guidelines and expectations
- Let the Continuous Integration server do much of the hard work for you
- Automate the process of pushing code through the pipeline
- Define development process standards to improve consistency and simplicity
- Manage dependencies on third party code to keep your software consistent and up to date
- Document only the most necessary and current knowledge

"The guidelines in *Building Software Teams* are wrapped in easy-to-understand metrics that make them more practical. As Peter Drucker said, 'what gets measured gets managed,' and this book understands this well."

—**Bob Belderbos**
Software Developer

Joost Visser is Head of Research at the Software Improvement Group (SIG). Joost also holds a position as professor of Large Scale Software Systems at Radboud University Nijmegen. He has obtained his PhD in Computer Science from the University of Amsterdam and has published over 100 papers on topics such as generic programming, program transformation, green computing, software quality, and software evolution.

PROGRAMMING

US $24.99 **CAN $28.99**
ISBN: 978-1-491-95177-4

Twitter: @oreillymedia
facebook.com/oreilly

Building Software Teams

Ten Best Practices for Effective Software Development

Joost Visser

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

www.ebook3000.com

Building Software Teams
by Joost Visser

Copyright © 2017 Software Improvement Group, B.V. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com/safari*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Nan Barber and Rachel Roumeliotis
Production Editor: Colleen Cole
Copyeditor: Kim Cofer
Proofreader: Jasmine Kwityn
Indexer: WordCo Indexing Services, Inc.
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

December 2016: First Edition

Revision History for the First Edition
2016-12-06: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491951774* for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Building Software Teams*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-491-95177-4

[LSI]

Table of Contents

Preface. vii

1. Introduction. 1
1.1 Software Development as an Observable Process 2
1.2 Software Quality According to the ISO 25010 Standard 4
1.3 The Contribution of Each Developer Matters 4
1.4 Measuring and Benchmarking Development Process Maturity 5
1.5 The Goal-Question-Metric Approach 5
1.6 An Overview of the Development Best Practices in This Book 6

2. Derive Metrics from Your Measurement Goals. 9
2.1 Motivation 10
2.2 How to Apply the Best Practice 11
2.3 Make Assumptions about Your Metrics Explicit 15
2.4 Common Objections to GQM 17

3. Make Definition of Done Explicit. 19
3.1 Motivation 21
3.2 How to Apply the Best Practice 21
3.3 Common Objections to Using Definition of Done 23

4. Control Code Versions and Development Branches. 27
4.1 Motivation 29
4.2 How to Apply the Best Practice 30
4.3 Controlling Versions in Practice 31
4.4 Common Objections to Version Control Metrics 33
4.5 Metrics Overview 34

www.ebook3000.com

5. Control Development, Test, Acceptance, and Production Environments. 37
5.1 Motivation 39
5.2 How to Apply the Best Practice 41
5.3 Measuring the DTAP Street in Practice 42
5.4 Common Objections to DTAP Control Metrics 45
5.5 Metrics Overview 46

6. Automate Tests. 49
6.1 Motivation 50
6.2 How to Apply the Best Practice 52
6.3 Managing Test Automation in Practice 53
6.4 Common Objections to Test Automation Metrics 58
6.5 Metrics Overview 60

7. Use Continuous Integration. 63
7.1 Motivation 64
7.2 How to Apply the Best Practice 65
7.3 Controlling Continuous Integration 66
7.4 Common Objections to Continuous Integration Metrics 68
7.5 Metrics Overview 69

8. Automate Deployment. 71
8.1 Motivation 72
8.2 How to Apply the Best Practice 73
8.3 Measuring the Deployment Process 74
8.4 Common Objections to Deployment Automation Metrics 76
8.5 Metrics Overview 77

9. Standardize the Development Environment. 79
9.1 Motivation 80
9.2 How to Apply the Best Practice 82
9.3 Controlling Standards Using GQM 85
9.4 Common Objections to Standardization 87
9.5 Metrics Overview 88

10. Manage Usage of Third-Party Code. 91
10.1 Motivation 92
10.2 How to Apply the Best Practice 93
10.3 Measuring Your Dependency Management 97
10.4 Common Objections to Third-Party Code Metrics 100
10.5 Metrics Overview 101

11. Document Just Enough. 103
11.1 Motivation 104
11.2 How to Apply the Best Practice 105
11.3 Managing Your Documentation 107
11.4 Common Objections to Documentation 108
11.5 Metrics Overview 109

12. Next Steps. 111
12.1 Applying the Best Practices Requires Persistence 111
12.2 One Practice at a Time 111
12.3 Avoid the Metric Pitfalls 112
12.4 What Is Next? 112

Index. 113

www.ebook3000.com

About the Authors

Joost Visser is Head of Research at the Software Improvement Group (SIG). In this role, he is responsible for the science behind the methods and tools that SIG offers to measure and master software. Joost also holds a position as professor of Large-Scale Software Systems at Radboud University Nijmegen. He has obtained his PhD in Computer Science from the University of Amsterdam and has published over 100 papers on topics such as generic programming, program transformation, green computing, software quality, and software evolution. Joost considers software engineering a socio-technical discipline, and he is convinced that software measurement is essential for development teams and product owners to thrive.

Sylvan Rigal has worked as a software quality consultant at SIG since 2011 and has advised clients on managing their IT since 2008. He helps clients achieve lower software maintenance costs and enhanced security by prioritizing improvements in software design and development processes. He holds an MSc in International Business from Maastricht University, The Netherlands. As an active member of SIG's software security team, Sylvan trains consultants on analyzing software security risks. When he is not assessing the technical health of software, he is training in Brazilian jiu-jitsu, enjoying Amsterdam and its restaurants, or traveling through Asia (approximately in that order).

Gijs Wijnholds joined SIG in 2015 as a software quality consultant in Public Administration. He helps clients get in control of their software projects by advising them on development processes and translating technical risks into strategic decisions. Gijs holds a BSc in Artificial Intelligence from Utrecht University and an MSc degree in Logic from the University of Amsterdam. He is an expert on Haskell and mathematical linguistics.

Zeeger Lubsen started consulting for SIG in 2008 as an all-around expert in software engineering and software quality and is now a senior consultant. Having worked as a web developer during his MSc study at the Delft University of Technology, he found great revelation in learning about how to build high-quality software. In his role as a consultant, he now helps both nontechnical managers and development teams to understand and grasp software. He finds that developing software is a creative and cultural activity, but also one that needs clear and objective guardrails to achieve realistic goals.

Preface

> You can't control what you can't measure.
>
> —Tom DeMarco

Insufficient software quality is a problem of all times and in all industries. We at the Software Improvement Group (SIG) see this time and again in our daily work of source code measurement and code review. That is why we have chosen to share the lessons we have learned with a larger audience. In the companion to this book, *Building Maintainable Software: Ten Guidelines for Future-Proof Code*, we focused on the contributions made by each individual developer. In that book, we discussed ten guidelines that any practicing software developer should master to consistently write maintainable source code. Those guidelines are based on our experience that lack of software maintainability is largely caused by simple issues that occur over and over again. And thus the best way to improve maintainability is to address these simple issues.

But that is not enough. To fully benefit from those guidelines, developers also need to work as a team according to a shared process. That is the focus of this book. This book discusses ten best practices for getting the development process right such that software is produced of consistently high quality. Our best practices do not just point a development team in the right direction. They are accompanied with a set of metrics that helps the team to consistently execute and monitor them.

Getting software development practices right is essential. The right development process facilitates the team and each individual developer to perform at their best. Using these practices, we can avoid inefficiencies in development work. Think of the difficulties of reconciling inconsistencies between contributions (merge conflicts!), chasing nonreproducible bugs, manually rerunning tests, and everything else that distracts us from creating the best possible product.

www.ebook3000.com

The Topic of This Book

This book lays out ten best practices for facilitating a team of software developers and enabling them to develop high-quality code. Having high-quality code is a great asset: it means lower costs for testing and software maintenance, and faster delivery of functionality. Conversely, software that is insecure, unreliable, or difficult to maintain is a source of developer frustration, delays, and software defects.

The practices address shared ways of working in the team, together with the technologies they employ, the processes they have followed, and the work environment they share. Think, for instance, of using *Continuous Integration* together with its required technology (see Chapter 7). Another example is standardization of code style guidelines (see Chapter 9). The best practices in this book are well-known, and many programmers may have heard about them during their education or earlier experience. This book puts those best practices in an overall, lightweight approach to manage software quality. The best practices presented in the following chapters are independent of the choice of programming language and the type of software that is being built.

Why You Should Read This Book

Taken in isolation, each of the best practices in the following chapters are well-known. What is not so well-known, however, is which ones are most important and how to determine whether they are being done in the right way. That is what this book aims to do, setting it apart from other books on software best practices in two ways:

We have selected the ten most important best practices from experience
: From our experience with advising software teams and managers, we know what works in software development and what does not. We also measure and benchmark software maintainability for hundreds of software systems each year, so the effects of specific practices such as Continuous Integration or test automation are very visible to us. We explain the most important best practices in a short and simple manner.

We explain how to measure success toward using these ten best practices
: Knowing a best practice is only the first step toward getting it right. We provide ways to measure how effectively each practice is being applied, and thus to manage its consistent use.

Who Should Read This Book

This book is aimed at those involved in managing and steering the software development process. You may be a team lead, a senior developer, a software architect, or a leader of IT projects or software development (such as a Scrum Master). You may have management responsibilities of your own, or perhaps you are advising/reporting to management.

This book will help you and your team adopt our ten best practices for effectively producing high-quality software.

What You Need to Know to Read This Book

This book is the follow-up to *Building Maintainable Software: Ten Guidelines for Future-Proof Code*. We assume you are familiar with the concepts explained in that book, either from your own experience or because you have read it. Having development experience in a modern object-oriented programming language will certainly make it easier to read this book. But even if you do not have (recent) programming experience, you will be able to understand, apply, and benefit from our best practices.

Many of the best practices we discuss will be familiar to those who use Agile software development methods. Regardless of how much you know about Agile and how much you like it, the practices we present in this book will help you to produce high-quality software more consistently.

What This Book Is Not

This book fulfills a different role than existing process approaches such as Capability Maturity Model Integration (CMMI) and ISO 9001. Those approaches focus on what kinds of process activities are needed to gain a certain process maturity. They provide conceptual frameworks that are applicable to many contexts, defining what is important in general terms. However, they do not focus on software development and they do not provide development teams with immediately actionable guidance. This is what our best practices aim to do.

Process frameworks emphasize documentation and structure. They prescribe certain areas for which you need to design processes and document them, but they do not prescribe which way of working suits you best. This book reasons in another direction. We recommend best practices that can be incorporated directly in your own existing work processes. Our guidelines do not require the use of a formal process framework such as the ones listed earlier.

www.ebook3000.com

While this book does speak about the technology available for equipping a state-of-the-art development environment, this book is neither a recommendation for particular technologies, nor a hands-on user guide for them. In more concrete terms: we do discuss tools and technologies for version control, automated testing, Continuous Integration, static code analysis, and dependency management. While we may name specific tools, such as Jenkins, this book is not meant as an endorsement of any particular technology. It is also not a technical guide for the technology at hand. Instead, this book discusses the role the technology plays in today's best practice software development.

About the Software Improvement Group (SIG)

The *real* author of this book is much more than just the one person mentioned on its cover. The real author is SIG, a software management consulting company. That is, the book consolidates the collective experience and knowledge of the SIG consultants that have been measuring software quality and advising about it since 2000. We run a unique, certified, software analysis laboratory that performs standardized inspections against the ISO 25010 international standard for software product quality.

Apart from assessing the quality of software products in our laboratory, SIG also assesses software development processes of our customers. We do so for two reasons. One is that it helps to put the outcome of a software product quality assessment in perspective. If the outcome of an assessment is lower than desired, this often can be explained by a process that impedes developing high-quality code. Or the other way around, if a team achieves high quality, is this *because of* or *despite* the environment in which they work? The best practices presented in this book are based on our experience in answering these types of questions.

SIG was established in 2000. Its roots can be traced back to the Dutch National Research Institute for Mathematics and Computer Science (*Centrum voor Wiskunde en Informatica* [CWI] in Dutch). After 16 years, we still keep and value our links with the academic software engineering research community. SIG consultants regularly contribute to scientific publications, and several PhD theses have been based on research to develop and improve the SIG quality models.

Related Books

For further elaboration on achieving high-quality software, we recommend several books in the field of software quality and development process measurement:

Building Maintainable Software: Ten Guidelines for Future-Proof Code by Joost Visser (O'Reilly)
: The companion to the current book, written by the same authors. It focuses on 10 guidelines for developers to write maintainable software.

Continuous Delivery: Reliable Software Releases through Build, Test and Deployment Automation by Jez Humble and David Farley (Addison-Wesley)
: This book discusses development process best practices in detail with focus on Continuous Delivery principles.

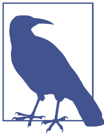
Software Development Metrics by David Nicolette and *Agile Metrics in Action* by Christopher W.H. Davis (both from Manning Publications)
: These books provide examples of using metrics in the software development process.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

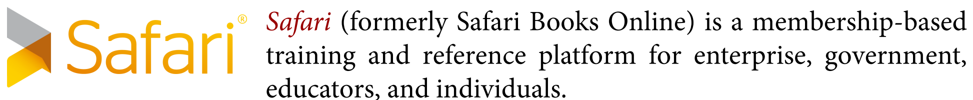
`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.



This element signifies a general note.

This element indicates an important remark

O'Reilly Safari



Safari (formerly Safari Books Online) is a membership-based training and reference platform for enterprise, government, educators, and individuals.

Members have access to thousands of books, training videos, Learning Paths, interactive tutorials, and curated playlists from over 250 publishers, including O'Reilly Media, Harvard Business Review, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Adobe, Focal Press, Cisco Press,

www.ebook3000.com

John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, and Course Technology, among others.

For more information, please visit *http://oreilly.com/safari*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://bit.ly/building_software_teams*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

We would like to thank the following people for helping us to write this book:

- Yiannis Kanellopoulos (SIG), our project manager for overseeing everything
- Soerin Bipat (SIG) for his thorough review
- Ed Louwers (SIG) for his help with the visuals in this book
- All current and former SIG employees that are working and have worked on perfecting models for measuring, benchmarking, and interpreting software quality

We would also like to thank the following people at our publisher, O'Reilly:

- Nan Barber, our text reviewer
- Steve Suehring, our technical reviewer
- Holly Forsyth, our managing editor

www.ebook3000.com

CHAPTER 1

Introduction

> Experience is the name everyone gives to their mistakes.
>
> —Oscar Wilde

Imagine you lived a century and a half ago, and you needed a car. Nothing fancy, just something for the occasional family visit. Obviously, it would be the variant that needs a horse in front of it, and it would be called a "carriage." Given a period in the mid-1800s in which carriage-making was not yet standard practice, your only option would be the local blacksmith. Carriage making was a *craft*, not an industrial process, with a unique result each time (so not with a fully predictable result). In addition, carriages were very expensive relative to disposable income: few people could afford their own carriage.

Fast-forward to 2016. Even if your town still features an artisanal blacksmith, that is not where you get your car(riage). Except for very special hobby cases, cars are manufactured by the tens of thousands on assembly lines, in an *industrial* process. This process is so predictable that two cars of the same model and color can only be distinguished by their license plates. Relative to disposable income, they also became *a lot* cheaper.

This predictability is a result of a mature industry, in which processes are standardized and highly automated. Of course, a comparison with software only goes so far. You do not need to build identical copies of the same software system for different users and the functionality of a car is fixed during the manufacturing process, while software needs to keep changing. But just as each car that comes off the production line must offer the same reliability, performance, safety, and comfort, so must each new version of a software system provide the same reliability, performance, security, and usability.

www.ebook3000.com

And consider what the experience of the car industry has done for its quality and process: designing components with computer simulation models, in a way that they are backward compatible—replaceable—with other versions of the car brand. Testing them individually before integrating them into the product. Testing overall car design in simulated environments, measuring and predicting effects of changes on the road. Doing crash tests, volatility tests, endurance tests. Following a structured process for checking and testing scenarios.

Software development can benefit a lot from these best practices in terms of predictability. For instance, process standardization, automation of the production line, and test automation and simulation. Also, the principle still holds for software that quality cannot simply be added after the product is finished. A bad design or chaotic implementation cannot simply be turned around. In a car assembly line this is evident, but from experience we know that this is just as applicable for software.

In this chapter, we first present *the big picture*: our view on software development and the role that the best practices of this book play in it. We discuss software development as a process in the following section, followed by software quality and ISO/IEC 25010, then the ISO standard that defines key software quality concepts. After that, we introduce the so-called Goal-Question-Metric (GQM) approach as a practical, lightweight way to control software development. Finally, we present a preview of the practices that are discussed in the subsequent chapters.

1.1 Software Development as an Observable Process

At the beginning of this chapter, we argued that software should be developed in a controlled process. This means that we can view software development as a process in the first place. In order to arrive at *output* (working software), organizations clearly need *resources* (developers, tools). Developers receive *inputs* such as requirements and change requests, and work on code within the constraints and *controls* of the organization (time, budget, quality goals).

The best practices presented in this book are about controlling this process. Controlling in this context means ensuring that working software is delivered that meets the requirements and quality goals, within a reasonable time and budget. *To control* is a verb, and rightfully so: in this book, we view process control as a continuous activity. To control software development, we need to observe its characteristics: output, input, resources, and controls. This observing and controlling asks for an active organizational role.

So what are concrete observations and control *actions* in software development, and how does this connect to the best practices that we discuss?

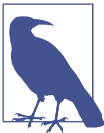
First, the observations. Software development is suited for measured observations. You may think of the following:

- How large is the backlog?
- How many issues have been resolved (over a given amount of time)?
- Which part of the codebase is covered by automated tests?
- What is the performance efficiency of the software delivered (e.g., as measured by transactions per second or number of possible concurrent users)?
- How much of the budget has already been spent?
- How many of the developers have passed a formal certification such as the Oracle Certification Program?

You can see that there is a lot that can be measured in software development. In fact, we believe that anything can be measured. But that does not mean that everything *should* be measured. This is exactly why we introduce GQM in "The Goal-Question-Metric Approach" on page 5, because this approach helps the controlling party (e.g., a development lead) to choose the best metrics, given specific goals.

Consider that measurements are only effective if they help you control something. Requirements, change requests, and time and budget constraints are typically not under the direct control of the development team or its lead. However, developers are in control of the way that development best practices are applied. You may think of control actions such as the following:

- Bring test automation to a higher level.
- Start using a third-party authentication framework.
- Introduce a tool for Continuous Integration.
- Change or improve staff skills by training.



With "process control," we are referring to measures that facilitate development and prevent issues. Resolving issues after software delivery is not a type of process control, but rather damage control.

www.ebook3000.com

1.2 Software Quality According to the ISO 25010 Standard

This book is about best practices for achieving high-quality software. We are using the ISO 25010 standard as a framework for understanding what makes software high quality.[1] This section discusses how ISO 25010 defines software quality and how we use and interpret this in this book.

Software Product Quality in ISO 25010

Up to a certain point, quality is something subjective and undefinable. Luckily, ISO 25010 makes a distinction between experienced (subjective) quality and inherent (product) quality. In ISO 25010 these are defined and separated as two models, called the quality in use model and the software product quality model.

The term quality in use refers to the impact a software system has on its stakeholders or users. The same system may have different *quality in use* for different people, depending on their wishes and behavior. This is subjective quality: it depends on who judges it.

We focus on the *software product quality* model. It deals with software quality characteristics that are inherent to a software system and can be assessed in production. The eight defined characteristics of software product quality are: maintainability, functional suitability, performance efficiency, compatibility, usability, reliability, security, and portability.

These eight characteristics are independent of users and context. That is a good starting point for assessing quality in a reasonably objective manner. For example, source code can be analyzed for security flaws or impediments to performance. And indeed, security weaknesses and performance issues are well suited for source code analysis. Note that it is dependent on the way the system is deployed, whether such findings *manifest* themselves in production.

1.3 The Contribution of Each Developer Matters

From experience we know that the contribution of each developer counts for achieving high-quality software. Following these best practices requires consistency and discipline from each team member. It is not just the most senior developers who are responsible for delivering quality software. And it is not only the junior programmers who should stick to the rules.

1 International Organization for Standardization, "Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models", 2011-03-01.

It also requires setting the right example, in order to avoid the "broken windows effect" where people break the rules and others follow. For example, when one developer takes shortcuts in terms of quality, other developers tend to copy that behavior. Setting the right example is not necessarily about being the most skilled engineer, but more about retaining discipline during development.

Developers can be greatly facilitated to retain that discipline. For example, establish a consistent development process in which technical steps are largely automated or define standards for using third-party code. Also, having metrics makes behavior transparent, such that a team can hold each other accountable for shared responsibilities.

1.4 Measuring and Benchmarking Development Process Maturity

As mentioned in the Preface, we (at SIG) measure both the software products and the software development process of hundreds of software teams each year. To measure the software product, code quality is rated against our Maintainability model. The thresholds for various metrics in the model are derived from code measurement data that we collected over the years.

Similarly, to measure software development processes, we check the activities of a team against a standard set of best practices. We call this a Development Process Assessment (DPA). Over the years, we have learned which best practices are most important and from the collected data we have been able to assign simple maturity levels to each practice. We categorize each practice as "basic," "intermediate," or "advanced," depending on how difficult it is to apply. That categorization is based on our data of how clients have adopted those practices.

The chapters of this book are a reflection of what we have learned to be most important to the working developer. At the end of each chapter, starting at Chapter 3, we provide results of our DPA benchmark. We relate each of the practices to the total benchmark and explain why certain practices are more difficult to implement than others.

1.5 The Goal-Question-Metric Approach

GQM serves as the *linking pin* between the practices: we present them as a combination of goals, questions, and metrics. We believe that anything can be measured. But that does not mean that everything should be measured. Measurements do help in identifying problems and monitoring the progress when working to correct them. Identifying the right metrics begins with asking the right questions. That is why the

www.ebook3000.com

Goal-Question-Metric (or GQM)[2] approach takes a central position in this book. We experience in practice that it is essential to be able to apply GQM's reasoning to make use of the right metrics. It is common to measure too few, too many, or irrelevant things and that is frustrating rather than revealing.

Defining the right measurements is not easy. Also, measurements can be misunderstood or used for the wrong purposes. This is why we devote the next chapter to the topic of how the GQM approach can help a team and a team leader with defining the right metrics to achieve high-quality software.

1.6 An Overview of the Development Best Practices in This Book

In the following chapters, the best practices are presented one by one. The order of the chapters signifies a level of "maturity" of those practices: from development prerequisites to "mature" ideals in software development.

Here is a brief overview of the outline we will follow:

Derive Metrics from Your Measurement Goals (Chapter 2)
: Apply Goal-Question-Metric to choose metrics that help you control your development process.

Make Definition of Done Explicit (Chapter 3)
: Define a specific Definition of Done to be able to tell how far away you are from your goals and to know when you are done.

Control Code Versions and Development Branches (Chapter 4)
: Use a version control system to control and merge different development tracks.

Control Development, Test, Acceptance, and Production Environments (Chapter 5)
: Take control over your Development, Test, Acceptance, and Production environments to achieve higher consistency in the way your software flows through the development pipeline.

Automate Tests (Chapter 6)
: Automated tests enable near-instantaneous feedback on the effectiveness of modifications. Manual tests, in which you provide some input and then observe the output and behavior of the system, do not scale.

2 The GQM approach is described in: Victor R. Basili, Gianluigi Caldiera, H. Dieter Rombach, "The Goal Question Metric Approach," in *Encyclopedia of Software Engineering*, Wiley, 1994.

Use Continuous Integration (Chapter 7)
: Writing tests and automating the build/test cycle makes developers' work easier and repeatable. Combined with automated testing, you have more confidence in the actual behavior of the system.

Automate Deployment (Chapter 8)
: Automate the process of pushing versions into production. This involves automation of tests and deployment steps, which gives more confidence when pushing into production, or rolling back when something goes wrong.

Standardize the Development Environment (Chapter 9)
: Standardize the development environment, because it relieves developers from having to manage manual configurations and makes the behavior of your software more predictable.

Manage Usage of Third-Party Code (Chapter 10)
: Reusing software written by others avoids reinventing the wheel, but needs to be managed.

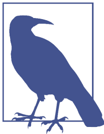
Document Just Enough (Chapter 11)
: The main things you need to document are nonfunctional requirements, high-level design decisions, and your Definition of Done. Documentation needs to be current, concise, and retrievable.

www.ebook3000.com

CHAPTER 2

Derive Metrics from Your Measurement Goals

> Only good questions deserve good answers.
>
> —Oscar Wilde



Best Practice:

- Apply **the Goal-Question-Metric approach** to determine **meaningful metrics** for achieving your goals.
- Make the **assumptions behind your metrics explicit** and **avoid common metric pitfalls**.
- This helps to **manage the development process** and improves the **quality of its outcomes**.

Every development process is aimed at gradual improvement, of the process as well as the product delivered. In order to determine progress, all kinds of measurements and metrics can be used, but not all measurements and metrics are meaningful in the specific context of an organization.

The Goal-Question-Metric approach (hereafter GQM) provides a simple structure for arriving at the right measurements for your context. These measurements allow you to manage the development process by assessing its status and progress, and to know when your goals are reached.

The GQM approach is a simple top-down approach: we start with *goals*, then *questions* that need answering to determine whether goals are being met, and *metrics* that can help answer these questions. So there might be one goal with multiple questions

www.ebook3000.com

with which each question has multiple metrics to answer. The GQM idea itself is very simple. GQM's purpose is to keep questioning whether you are coming closer to your goals.

2.1 Motivation

On the face of it, GQM looks rather trivial. Of course metrics should answer the right questions. Using metrics for managing the development process provides you with facts instead of impressions. Unfortunately, it is fairly common not to use software-related metrics at all. So exactly how useful are metrics? That depends on your goals. This simple reasoning is the main advantage of using GQM: it gives a structured approach toward the right metrics instead of choosing metrics off the cuff.

From practice we know that choosing the right metrics in the right context is not trivial. If software metrics are used at all, they are often used without keeping the goal in mind, leading to negative side effects. With GQM you get the right clarity and methodology in order to avoid metric pitfalls, the four most common of which are as follows:[1]

Metric in a bubble
: A metric is used without the proper context. You should track metrics over time or compare a specific metric to other cases. A metric is only valuable in comparison.

Treating the metric
: The team optimizes the metrics but does not solve the problem that the metric is meant to signal. You should have a clear notion of the problem you address with a metric, and how the metric helps you in solving it.

One-track metric
: A sole metric is used, which may lead to wrong conclusions. For example, measuring the total volume of a codebase, without measuring the degree of duplication, may give a false impression of progress. You should use several metrics that jointly address a goal (i.e., a problem that you want to solve).

Metrics galore
: Too many metrics are used, while many of them do not add value. Typically this leads to a team ignoring the metrics, because they cannot interpret them meaningfully. You should limit the number of metrics you use, for instance, by not adding a metric that correlates too much with one you are already using.

1 Originally published in the following article: Eric Bouwers, Joost Visser, Arie van Deursen, "Getting what you measure," Communications of the ACM, Vol.55, No.7, p.54–59, 2012.

In the next section, we explain how GQM works and how you can define meaningful metrics.

2.2 How to Apply the Best Practice

To help you understand applying the best practice, imagine that you are leading a team to build a new system. Let us assume this concerns a modern web application that is well suited to Agile development. As an example, consider a system providing video streaming services to users over the Web.

Now suppose that development started not too long ago. Before, developers in your organization were accustomed to classic waterfall development (developing based on detailed designs and specifications in long iterations). Now you have decided that for this development Agile is suitable. You suggest dividing work into two teams of seven people. Team members have different expertise and experience, yet all of them have at least half a year of development experience in the system's main programming language. You do not intend to follow the Agile Manifesto to the letter, but want to adhere to basic Agile practices:

- You do development work in fixed-time iterations (sprints).
- You demonstrate functionality after each sprint to user representatives.
- You work with a prioritized backlog of system requirements.
- You assign a Product Owner role, as a representative of "the business" that represents the interests of the users.
- Development work is peer reviewed before it is pushed to the main development line (the trunk).

The team feels that it takes an unusual amount of time to change functionality. Your initial diagnosis is that it may be due to high code complexity (many logical decision paths), so you are interested in tracking the complexity of the codebase.

Goal

Consider that with measurements you want to obtain meaningful values about relevant characteristics. To make sure that you are measuring the right things, a measurement should follow from a goal that it helps to achieve.

www.ebook3000.com

So the first step is to determine a specific goal. Every goal should have five characteristics. This helps you define a goal in a meaningful way. A goal should clearly describe these elements:

Object of examination
: What are we looking at? In this case, that refers to the software that you are managing.

Purpose
: Why are we examining the object?

Focus
: What attribute or characteristic of the object are we interested in? Depending on your measurement purpose, a characteristic could be, for example, performance, maintainability, functional fit, or some other attribute.

Viewpoint
: Who is the primary target? This is about perspective of the goal and may be you as an observer.

Environment
: In which context does the object exist?

This can be written down by filling out a table or in the form of a sentence. By starting to fill out a table you can obtain a sentence that includes all aspects. Table 2-1 shows an example.

Table 2-1. Example for defining a goal for reducing code complexity

Topic	Response
Object of examination	Total codebase
Purpose	To reduce
Focus	Complexity of units
Viewpoint	Development team
Environment	Streaming application development

In sentence form, the goal would be "to analyze the total codebase of the streaming application in order to reduce the complexity of units from the viewpoint of the development team."

Filling out a table may appear to be a tedious task, but it is not that much work and helps to make problems explicit. Therefore, it aids in creating the right questions.

Question

The next step is to formulate questions related to your goal that help achieve that goal. There is no strict bound on the number or type of questions. As an example, consider the questions in Table 2-2.

Table 2-2. Questions in this GQM model

Question	Description
Q1	How is the complexity of units in the codebase distributed?
Q2	Does complexity in units stay the same or change over time?
Q3	Is there a correlation between complexity of units and time to market?

With these questions in mind, we can start to think of the right metrics.

Metric

Consider that metrics can appear in different forms. A metric can be either *objective* (its value is not dependent on who measures it) or subjective (its value *is* dependent on who measures it and tends to involve an opinion or estimation). In addition, it might be *direct* (its value originates directly from observation) or *derived* (its value is calculated or deduced in some way).

Suppose that we use a uniform measure of the complexity of code units (e.g., McCabe complexity[2]). Then we can use a tool to go through the codebase and rank the units in terms of their complexity. That would be an objective, direct measurement that quickly answers Q1. Once we start using the metric over time we can confidently answer Q2. For instance, we can measure the number of units that become more complex relative to the total number of units.

To measure the correlation between unit complexity and time to market we need a measure of time to market. This could be the total time needed from receiving requirements, up to user acceptance of its implementation. That means that you need two observations: when receiving the requirement and after putting code into production. After every release, you could measure the correlation in some form (e.g., by calculating a correlation metric such as the Spearman coefficient of the two measurements[3]). Then you can observe whether there is indeed a relationship and whether that relationship changes over time.

2 McCabe complexity is a standard measure for the complexity of code; it can be interpreted as the minimum number of test cases that you need to write for a unit and it is based on the number of decision statements in which code can take more than one direction.

3 As an example, the Spearman coefficient is a standard measure for correlation, but its mechanics are outside the scope of this book.

www.ebook3000.com

Consider how important it is to make your assumptions explicit. The preceding correlation is a deduction of the metrics of unit complexity and time to market. It is slightly subjective, as it depends on how you define when functionality "hits the market." When explicit, you can consistently interpret the measurements.

Of course, not all requirements are of equal functional and technical difficulty. Therefore you can choose to measure deviations of the estimations that the development team makes when planning implementation of requirements. That has the risk of essentially measuring estimation skills, not code complexity. The relationship between unit complexity and development team estimates could be included as a metric itself, but here we choose to ignore that.

Table 2-3 summarizes the metrics that complete the GQM model.

Table 2-3. Metrics in this GQM model

Metric #	Metric description	Corresponding question
M1	Unit complexity in the codebase	Answers Q1
M2	Complexity of units written for new functionality	Answers Q2 and Q3
M2a	Percentage of code with certain complexity (e.g., volume percentage of units with McCabe complexity of 5)	Answers Q2
M3	Time from specification to acceptance of functionality	Answers Q3

The team lead can now decide upon what and how to measure. In this particular case of complexity in the codebase, the development team may decide to actively reduce that complexity (e.g., by splitting complex units into simpler ones). But if the complexity seems to be uncorrelated with the time to market, Q3 is answered and M3 becomes irrelevant. Such metrics also require careful consideration. For the purpose of testing and maintenance, average complexity in a codebase is much less important than "hotspots" of complexity that are hard to understand and test. For identifying hotspots you would need to define categories of what is, for example, "Good," "OK," "Hard to test," and "Untestable."

Goals may also change: if the system owner turns out to be satisfied with the time to market, there is no urgency to reduce complexity *given this particular goal* and the team may ask itself "why are we measuring again?". Obviously, for the sake of maintainability, there are still good reasons to keep a codebase as simple as possible. In practice it is common to reason back from those metrics to determine whether they actually bring you closer to your goals.

If a metric appears not to help in achieving your goal, remove it. If a specific GQM model becomes obsolete, do not use it until you need it again.

2.3 Make Assumptions about Your Metrics Explicit

Simplifying a metric itself is easier than trying to correct the metric for all possible objections. The most common objection is that the metric does not measure what it promises to measure, because of various circumstances. But that does not render the metric useless. You should make assumptions explicit about the context in which the metric is valid. And of course, circumstances may change, making assumptions invalid. With clear assumptions, this relationship is clear.

Let us discuss some common assumptions underlying software development metrics (these are not necessarily true, but they help you in making your own assumptions explicit):

Assumption: Measurements are comparable

For practical purposes, you will need to assume that measurements have comparable meanings. When comparing data we at least need to assume that it is administered completely (no details are left out), correctly (intended data is recorded), and consistently (data is administered in the same manner/format).

As an example, consider that you want to measure issue resolution time (issues here defined as including both defects and features). You can greatly simplify measurements when you assume that issues are of approximately the same work size. Consider what this means for the process of registering issues.

This assumption forms a good reason to administer issues in approximately equal sizes. To achieve this, issues should somehow reflect how much work they take to solve. This may be a simple classification such as low, medium, or high effort or something more detailed (e.g., hours, story points, etc.).[4]

Then we also need to assume that the classification is scaled such that *scales reflect the same size*. This is to avoid that "low" and "medium" effort are about the same amount of work, but "high" is tenfold that. In this example, the difference between classifications such as "low" and "medium" effort is equal to the difference between "medium" and "high" effort.

Assumption: Trends are more important than precise facts

A typical assumption is that a trend line that compares measurements over time is more meaningful than the precise numbers. The trend will reflect *how* things are changing, instead of precise facts. This implies that outliers should have a small impact.

4 See Chapter 3.

www.ebook3000.com

Assumption: Averages are more important than outliers

Trends focus on average movements. Outliers are sometimes just that, outliers. So this assumption is not always appropriate. An outlier may signify a problem, for example, when it takes extraordinarily long to run certain tests or to solve bugs. However, to simplify your measurements, you may assume that some circumstances are not worth correcting for. Consider measuring issue resolution time. Using business hours between identification and resolution would be more accurate than the number of calendar days between them. However, you may choose to ignore the difference when you assume that the impact is small and that the distribution is consistent.

These assumptions are about the metrics and their underlying behavior. Let us move on to the usage of these metrics.

Find Explanations Instead of Judgments When Metrics Deviate from Expectations

When you find a metric that deviates from the average, resist the urge to judge it as good or bad too soon. It is more important to find explanations for why metrics deviate. That way you can put the metrics into practice for achieving their goal.

Consider the goal in the GQM reasoning that wants to understand how team productivity is influenced. Let us say that the issue resolution time had grown last month. Not necessarily *deteriorated*, but grown. Investigate this difference by asking yourself why this deviation may have occurred:

- What are the most plausible explanations?
- Are there reasons to believe that the assumptions are not valid anymore?
- Is this period different from other periods? Are there special circumstances (e.g., holidays, inexperienced new team members)?

Using Norms with the GQM Approach

In many cases, metrics become useful and supportive only when there is a clear definition of good and bad values—in other words, there has to be a *norm*. You can add norms to the GQM models by defining them for each metric. In the example in this chapter, a norm on complexity could be that the McCabe unit complexity should be below 5, in 75% of the code (Figure 2-1). Once you agree upon a norm, you can visualize a metric as, for example, a distribution of different risk categories, to see what percentage of the units is too complex. Such a norm may also serve as the basis of a Definition of Done (DoD, see Chapter 3).

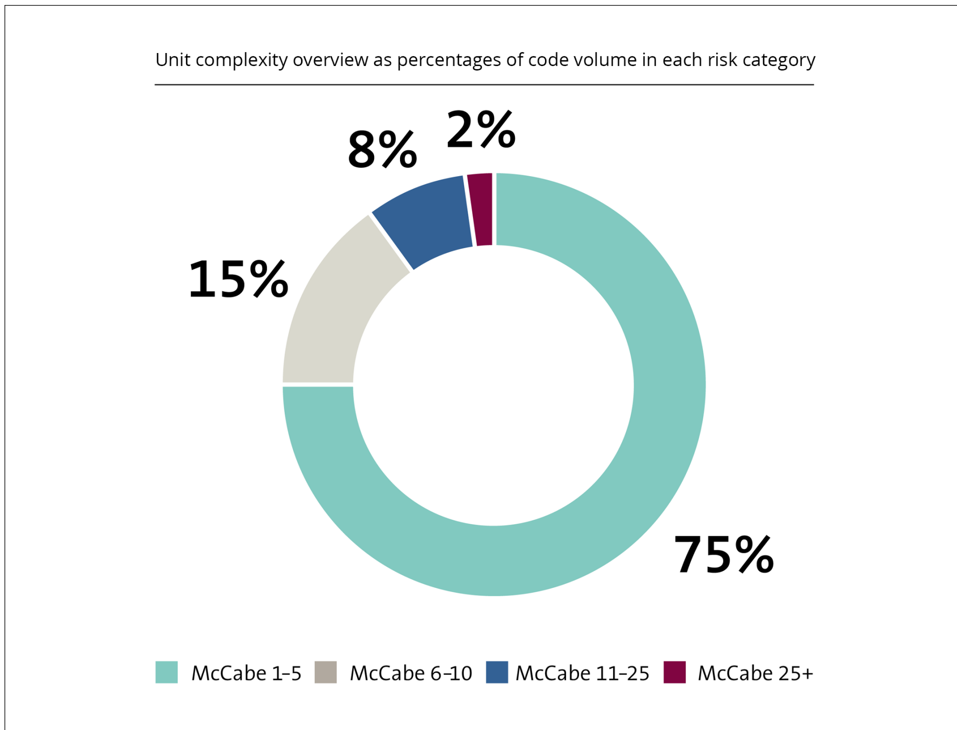


Figure 2-1. Example of code volume distribution of unit complexity

A variation on this is one of shifting norms. Then you define an end goal, such as "the unit test coverage of our system should be 80%" and continuously move the norm to the latest and highest state. That would show whether you are "setting new records."

2.4 Common Objections to GQM

The GQM approach is a powerful way to determine the right metrics for your problem. Often, measurement goals are phrased based on the software metrics that are already available in tooling. With GQM in mind, this is clearly the wrong order of things: metrics should support decision making, instead of metrics defining what is important. In practice this means that you may need to install different tooling for measuring purposes.

Goals come before questions, questions come before metrics. Measure what you *need* to measure instead of measuring what you *can* measure.

www.ebook3000.com

Objection: Yes, Good Metric, but We Cannot Measure This

"That is a good metric, but it we will not be able to measure it correctly enough to rely on it."

Even when the GQM approach is performed to define metrics, in our practice we often encounter objections against measuring in general. Typically the objections are not about the metrics themselves. They tend to take the form of "Yes, it is a useful metric, but we cannot use it because…". In Table 2-4, common objections are discussed.

Table 2-4. Common objections to metrics and possible solutions

Objection	Possible solution
Measurements may be interpreted in the wrong manner by management.	Help management to interpret them and simplify the measurements.
Metrics may act as a wrong incentive (leading to "treating the metric").	Do not make the metric itself a KPI/metric for job evaluation.
Data is registered in an inconsistent/incomplete/incorrect manner.	Automate measurements, help your colleagues to register consistently by defining a step-by-step process.
Goal for measuring is unclear or outdated.	Revisit the GQM model.
This metric is not appropriate or specific for our goals.	Explain the goal and assumptions of the metric (using GQM). Explain whether alternatives (if any) are equally simple to gather and use. Possibly revisit the GQM model.

These objections may all be relevant in a particular situation, but *having* the metric and *using it* are separate things. It may be fine to have an inconsistent metric as long as you acknowledge that it is inconsistent and is for rough trend analysis only. A typical example is "those" team members that are lenient in registering their hours (late, incomplete, or undetailed). If you acknowledge that, you should either focus on administrative discipline to get the metrics done, or accept it as given and look more at "the bigger picture" instead of precise values.

In general, having *a* measurement is just another data point. In that sense we believe it is better than having no measurement at all and fully relying on gut feeling. Metrics may be used or ignored, as not all metrics are "born equal" in terms of quality.

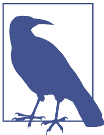
We will use the GQM approach throughout the book. As such, you will see GQM return in all the subsequent chapters.

CHAPTER 3

Make Definition of Done Explicit

I have the simplest tastes. I am always satisfied with the best.

—Oscar Wilde



Best Practice:

- Make a **Definition of Done**, listing all items necessary to reach your development goals.
- Ensure that the Definition of Done is **measurable**.
- This improves the quality of development because you can **track the progress of the product**.

Before you can manage something, you need information about status and change. Are we reaching our development goals? Well, that depends on the status and on what your goals are.

The Definition of Done (hereafter DoD) is a term specific to the Scrum approach of Agile development. It can be an organizational standard or it is made by the development team. The DoD defines when a product is "done": everything that needs to be done to produce software in a releasable (or *shippable*) state to end users. It does not guarantee user acceptance or success in production, but does provide a check toward achieving nonfunctional requirements (such as performance, security, reliability, or other characteristics as defined by the ISO 25010 standard). Because user acceptance criteria are defined by the owner of the system, they are complementary to the DoD. Think of the DoD as a list of the software development activities that add value to the end product, though those activities are not necessarily visible to the user. For example: a minimum amount of unit test coverage, adherence to coding standards, and documentation requirements.

www.ebook3000.com

Revisiting Agile Practices

The DoD is a concept that originates with Agile/Scrum. In this book we adopt this concept for any development process. Therefore, some Agile/Scrum background is useful for those not quite familiar with its concepts.

Short iterations

From experience we know that in software development, major changes made late tend to be more difficult and expensive to implement, compared to when they are identified early. An advantage of Agile is that it forces work into short iterations (typically 2 or 3 weeks). Those iterations are ordered by what is most relevant *now*; that is, functionality or system behavior that adds the most value to the system owner.

The end of each sprint is a mini-evaluation in which the *Product Owner* (hereafter: stakeholder) can decide to continue or halt the project. The possibility to halt a project early reduces the risk of fully investing and committing and then realizing later on that it is doomed to fail. Therefore the (implicit) decision to start a new sprint becomes a trade-off of effort and prioritized benefits.

For management, this is a change in thinking about projects. The way of initial budgeting remains broadly the same, but there is initially only commitment for the amount of effort that will be spent, instead of a clearly defined end point apart from "a valuable system that works."

Agile planning

Given Agile's short iterations, it requires planning functionality in small pieces. The level of formality differs a lot between organizations, but generally a *sprint planning* takes place in which the team prioritizes work together with a stakeholder. The definition of functionality is typically based on *user stories*. A user story is a feature described from a user perspective, which ideally fits into development work for a sprint.

To aid planning, the team defines *story points*. A story point is a rough estimate of effort for implementing functionality: the more points, the more work is needed. As it has only team-specific meaning, it is meant for internal planning and comparison. It can act as a measure of productivity *within the team*, by comparing the average speed with which story points are implemented. That speed is simply called *velocity*. The velocity may be visualized in a *burndown chart*, which is a graph of the amount of work left (in terms of story points) versus time. With story points on the vertical axis, they are "burned down" over time.

To keep an overview of tasks, development teams typically use *kanban* boards. In its simplest form, it is a visible board with different columns for development stages (such as: plan/develop/review/test/release). Team members then place

tasks/issues in those columns with sticky notes to signal in what stage of development they are and who is working on them.

The DoD may change over time, but should only do so in between sprints and only in agreement with the party that is responsible for the product.

3.1 Motivation

This section describes the advantages of using a DoD. Defining an end state in a DoD helps you to manage achieving that end state. Because the DoD concerns nonfunctionals, it puts focus on software quality.

With a DoD You Can Track Progress and Prove Success

In general, a DoD is useful for helping you to manage the nonfunctional aspects of software development. The reason is simple. Knowing the end goal and knowing the current status allows you to see whether you are on track and what needs to be done. This applies to different roles in the organization: developers can assess when their work is done. A project manager can assess whether software quality meets expectations. Acceptance testers can verify nonfunctionals such as performance.

A DoD Focuses on Software Quality

Because a DoD defines when a product is ready from the developer's perspective, it is an aid in assuring the quality level of your software. Consider that when software quality is not defined, it is hard to manage whether software implementations adhere to your quality standards. However, in practice we often see a lack of software quality requirements.

When checking off a DoD, it can be confirmed that development is actually *done*. Done then means that implementation has finished, including coded functionality and nonfunctional work such as unit tests, coding standards, or documentation.

3.2 How to Apply the Best Practice

In order to know both the current situation and assess the end situation, a DoD should comply to the following:

It is assessable
: Preferably its items are quantified as a metric.

www.ebook3000.com

It clarifies its own scope

A DoD normally applies to all development within a sprint but may be specific for a feature. Also, what is "done" is not always defined in the same way. If responsibilities for development, testing, and deployment are clearly separated (e.g., they are different departments), then "done" for the development team may mean "development work is done." If a team mixes these specialties in one team with a shared responsibility (such as in a *DevOps* team), an item may be considered "done" when it is accepted and ready for release, and pushed from an acceptance version into production. In any case, the DoD should define an end state within the scope of responsibilities of the team.

In Chapter 2 we showed an example of a norm for a code complexity measure. Extending it, a specific technical requirement for the DoD may be:

> *In all development for this system, at least 75% of code volume has a McCabe complexity of at most 5. For newly written units in development, at least 75% of units comply.*

A DoD typically includes many different requirements that may be as concise as the team wants it to be. Several topics tend to reoccur in DoD lists. Consider the following (nonexhaustive) elements of a DoD:

Version control

- All source code is committed and merged into the main development line (trunk) or a specified branch.
- Code commits include a code/identifier in a specific format for functionality/issue identifier.
- Commits build without errors.

Proof that code works as intended

- There are no failing unit, integration, and regression tests.
- During code maintenance, corresponding unit tests, integration tests, and regression tests (if applicable) are adjusted accordingly and their code is committed to version control.
- Unit code coverage for a specific task is at least 80% (as a rule of thumb) as defined by number of lines of code affected by unit tests.

Administration for maintenance and planning

- Documentation requirements (dependent on team agreements): code should be mostly self-documenting, contain comments sparingly yet always in a specified format (date-author-issue ID-comment). A separate document describes overall system structure in no more than five pages.
- No code comments are left behind that signal work to be done (i.e., ensure that TODOs are addressed and removed).

- The user requirements (issue/ticket/requirement/user story, etc.) are closed in a traceable way (e.g., in an issue tracking system and/or on the kanban board).

Wrap-up to complete sprint

- New acceptance test cases are created for added functionality.
- A sprint is formally finished when a stakeholder signs off the sprint after a demonstration of functionality ("sprint review").

3.3 Common Objections to Using Definition of Done

This section discusses objections regarding usage of DoD. The most common objections are about the effort required in using a DoD and maintaining it, and the wrong incentives it may give to the team.

Objection: DoD Is Too Much Overhead

"Using a DoD is too much overhead."

A DoD is not a bureaucratic goal: it is an agreement within the team that provides transparency and a way of managing software quality. By extension this is also an agreement with the business owner.

From practice we know that quality must be defined in advance. When quality is not defined at all, it is the first thing that suffers when deadlines need to be met! Shortcuts will be taken in the form of code hacks, workarounds, and concessions in testing.

Quality must not be left to chance. Even if it is not called a "Definition of Done," a team should agree on quality expectations.

If a team is very skilled and quality-conscious, the DoD is not a big deal: it will simply be a representation of the way of working. If a team is more junior and not yet quality-conscious, it will help them develop better quality software by focusing on what is important.

The DoD's level of detail and its application is up to the team. A team might especially favor a summary definition, or apply the DoD as a quality agreement that is only assessed at the end of each sprint. A check at the end of each sprint is a minimum requirement, though; otherwise, the team cannot guarantee quality toward the system stakeholders.

www.ebook3000.com

Objection: DoD Makes the Team Feel Less Responsible

"Formalizing what 'done' means makes the team feel less responsible for overall quality because they just adhere to the DoD without thinking about actual improvement."

Consider that the DoD itself defines an end state of work that you are doing right now, but implies nothing about quality improvement. If a certain (quality) characteristic appears to be lacking (e.g., performance), the DoD should be written in a way to facilitate that. Typically such requirements are nonfunctional in nature.

Also, remember that the DoD is only as fixed as you want it to be (typically fixed for a sprint at least, though). Change its content or its application together with the team if you feel that the DoD does not provide the right incentive to achieve high-quality software. A DoD should be a good representation of which quality aspects are valued by the team, and to which standards they hold each other's work.

Objection: With DoD There Is No Room for Technical Maintenance

"With the DoD there is no room for technical maintenance such as refactoring or upgrading frameworks."

In fact, there is. If "technical maintenance" here means, for example, simplicity of code units, then that could be part of the DoD. If the technical maintenance has a broader scope, like renewal of frameworks, the team should put it on the development backlog and it can then be treated as "regular functionality." Putting technical maintenance on the backlog is an effective custom. It forces the team to reason why the improvements are advantageous for the system's operation or maintainability.

You can imagine that such technical maintenance typically is not foreseen by stakeholders and therefore it may not receive priority from a user perspective (because the improvements are invisible). Discuss with the team how they prefer to deal with this. For example, one can agree that a certain percentage of time is reserved for technical maintenance (say, 10%). Within the 90% remainder, user stories can be prioritized. Such a fixed effort can be agreed on with the stakeholders.

This should not relieve developers from having a critical eye toward low-quality code when they see it. For that matter, the *"Boy Scout Rule"* is a great principle. For developers this means that the right opportunity to refactor code is when modifying that code. The result is that code is left behind "cleaner" and of higher quality compared to when the developer found it.

Objection: Changing the DoD May Mean Extra Work

"When the DoD changes, should we revisit all earlier sprints for compliance with the updated requirements?"

The simple answer is: only when the change in DoD is important enough to warrant this. Changes in the DoD affect only what is being delivered from now on. The effects of changes should be considered for the next sprint planning. If the DoD becomes more strict or more specific, the team should agree with the system stakeholder whether the change applies in retrospect. If so, it should be put on the backlog, because reanalyzing code takes time. Then in the sprint planning it will emerge whether applying the changed DoD over older pieces of code has enough priority.

You can assess whether your DoD standards are being met by measuring them. Typically this is done with some kind of quality dashboards that include code quality measurements (such as unit test coverage, coding standards violations, etc.). So clearly, the more specific and quantified a DoD is, the easier it will be to measure them. Remember to walk through the GQM process to determine what kind of metrics help you toward your goals.

www.ebook3000.com

Experience in the Field

In our daily practice, we assess whether development teams have *specific, measurable, attainable, realistic, and testable* (SMART) nonfunctional requirements that they use to build and test their software against. Having SMART nonfunctionals is not the same as having a Definition of Done, but it goes a long way. We observe that it is an advanced practice: in 77% of the cases nonfunctional requirements are either not defined or not quantified. Often, it is the case that a lack of SMART requirements from business stakeholders trickles through to the development team. This makes development hard, because nobody knows what the delivered software should be able to do from a nonfunctional perspective (see Figure 3-1).

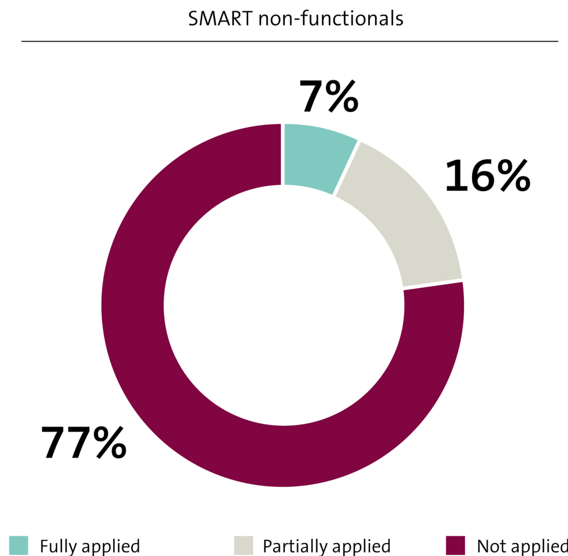
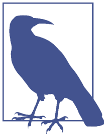


Figure 3-1. Benchmark results on usage of SMART requirements in development teams

One of the key arguments for judging the maturity of a team when it comes to nonfunctional requirements is whether they have a Definition of Done with quantified metrics to assess whether they reach their goals. This is regardless of whether they actually have been required to do so by their stakeholders. To apply this practice partially typically means that a DoD has been defined, yet without quantified metrics.

CHAPTER 4
Control Code Versions and Development Branches

> I was working on the proof of one of my poems all the morning, and took out a comma. In the afternoon I put it back again.
>
> —Oscar Wilde



Best Practice:

- Use a **standard version control system** and keep track of development branches.
- **Integrate your code regularly** and **commit changes both regularly and specifically**.
- This improves the development process because **developers can work in isolation** without drifting apart.

In this chapter we apply the ideas of GQM to version control. Bugs and regressions (recurring defects) occur regularly during development and maintenance of any codebase. To solve those you will need to reanalyze code, and you may need to revert some code or configuration to earlier versions. Consider what happens if there is no version control and all developers can only work on one version. It will be hard to see who is changing what and hard to avoid that developers break each other's code.

Version control is a solution to this. It does what it says: controlling versions, which allows you to divide work among developers. In its most basic form, a version control system tracks changes by labeling them with the person who made the change, and a timestamp of that change. Having the version history available allows you to revert to an earlier version when necessary.

www.ebook3000.com

To better understand what version control systems do, let us start at the beginning. There is one main version of the code that is leading for what is being put into production, referred as the trunk (main line). When developers make changes, they can commit changes and additions to the trunk. When multiple developers work on the trunk, they can get in each other's way. To avoid this, variations can be developed in branches (meaning a new, parallel version of the software). Branches can later be merged into the trunk.

Therefore version control systems generally have integration functionality for merging different versions. In that way, developers can work on isolated parts of the source code. Usually, the merging of source code (which is text-based) proceeds automatically, but when there is a conflict, it needs to be resolved manually.

There can be a central repository that holds the trunk (a centralized version control system). In that case, developers make a copy of that trunk to work on and commit to the central version control system. With *distributed* version control, each developer has a local repository, and changes can be shared among each other to create a version (pushing their own changes and pulling changes of others).

There are several version control systems available, such as *Git*, *Subversion*, and *Mercurial*, each with a slightly different vocabulary and different mechanisms. A notable difference is that Git and Mercurial allow developers to commit changes into local (individual) branches on their own machines, which can later be merged with the main repository. Subversion prefers to always commit changes directly to the central repository.

Although version control has the most visible benefits for source code, you should put *all* versionable parts of the software in version control, including test cases, database configuration, deployment scripts, and the like. That way, you can re-create the different testing and production environments more easily. Documentation can be under version control, and sometimes external libraries (when you cannot rely only on the versioning of the library provider).

Do not put generated code into version control. You do not need to, because the build system will generate that code for you. In case that generated code is actually maintained, it should be versioned and be part of version control. But you should refrain from adjusting generated code as it will lead to problems when you need to regenerate it.

There are situations in which libraries need to be included in version control (e.g., for audits to ensure that known-working versions are used). A more thorough discussion on using third-party code appears in Chapter 10.

4.1 Motivation

Different version control systems share common advantages: they allow the development team to track changes over time, they use branching to allow developers to work independently on variations of the same codebase, and they merge files automatically when versions somehow conflict.

Tracking Changes

Tracking changes with a version control system has the advantage of going back in time. When things go right, there may not be a reason to do so. However, when functionality breaks, comparing old versions is a good tactic to find the cause of the issue (instead of reanalyzing the whole code). So the developer can revert the source code version (in its working copy) to the moment when the bug was introduced. This comparison helps with fixing the bug or simply replacing the new version with the old one. Then, a new (local) version can be merged back into the trunk again.

Version Control Allows Independent Modification

Independent modification in part of the source code avoids conflicts that happen when different pieces are adjusted at the same time. So when two developers need to implement different functionalities but need the common codebase for testing, for example, they can create two isolated branches of the same codebase that they can modify independently. In that way, the code of the one developer will not interfere with the work of the other. When one of the branches functions satisfactorily, it can be pushed back and merged into the main development line. Then the changed functionality becomes available to the whole team.

Version Control Allows Automatic Merging of Versions

Every time developers need to exchange their work, source files need to be merged. Merging source code versions manually is notoriously difficult if they are very different from each other. Luckily, version control can do most of the merging automatically by analyzing the difference between a modified file and the original. It is only when there is no definite way to combine the two files that a problem (merge conflict) occurs. Merge conflicts need to be dealt with manually because they indicate that two persons made different changes to the same line in a file, and you still need to agree on the modification that is most suitable.

www.ebook3000.com

4.2 How to Apply the Best Practice

Consider that changes and additions are much easier to manage when done in small iterations. Each iteration moves the software forward with a small step. Difficulties arise when you want to make large and ambitious changes infrequently.

Therefore, there are two main principles you should adhere to:

- Commit specifically and regularly
- Integrate your code regularly, by building the full product from source code and testing it

We will show how you can use different metrics of the version control system to measure the quality of your development process.

Commit Specifically and Regularly

In many cases, team members will not be aware of what other developers are working on, so it is important to register this when you make changes in version control. So, every commit should be *specific* to one piece of functionality or, when you have an issue tracking system in place, it should be linked to precisely one issue. Commits also should be done on a regular basis: this helps to keep track of the flow of work and of the progress toward the team goal. Moreover, you will avoid merge conflicts by committing your changes to the version control server regularly.

Note that "keeping changes small" implies that you also divide development work into small parts.

A typical indicator for this principle is the commit frequency of team members, and to link the commit messages with the issue tracker. You could further measure the build success rate to determine how quickly issues are solved, and in how many cases the issues need extra work. This could serve as an alternative measure of team velocity.

Integrate Your Code Regularly

Besides committing new code on a regular basis, it is vital to integrate code between different branches of the codebase as often as possible, preferably daily. This is because the later you integrate your code, the more likely it is that there will be merge conflicts or bugs introduced.

There should be a proper balance between the time a programmer operates independently on a branch and the number of merge conflicts. An indication of this is (average) branch lifespan or the relation between branch lifespan and the number of related merge conflicts.

Having long-lived branches (that exist for, say, more than two sprints) poses problems. In general, it makes truly isolated maintenance more difficult, and in some cases unlikely. Long-lived branches heighten the risk of merge conflicts, because the longer they live, the more they tend to divert from the trunk. When branches start to evolve into notably different functionality, they may even become impossible to merge and become forks (independent versions of the software). It is easy to imagine the panic if one fast-paced developer that works independently tries to merge a month of work one day before the release deadline.

4.3 Controlling Versions in Practice

Keeping in mind that effectively managing code variations requires avoiding long-lived branches, you also wish to confirm whether the team works more productively with the help of version control. Therefore, you would expect that issue resolution time should not increase when applying version control. Assuming that the goals are formulated from the viewpoint of the team lead, you can come up with the following GQM model:

- **Goal A**: To manage code variations by preventing long-lived development branches.
  - **Question 1**: How much time passes between commits?
    - **Metric 1a**: Number of developers that did not commit the last 24 hours. This is a simple metric that serves as an indicator of whether long-term branches are developing. Of course, developers could be working on other things, out of office, not committing code. Those explanations are easy. The not-so-obvious cases are the interesting ones. Expect this metric to be fairly stable over time, yet it will almost never be zero.
    - **Metric 1b**: Average branch lifespan. This metric attempts to measure that branch lifespan is limited to avoid the risks of long-lived branches. Expect a downward trend toward the period of time that it takes to implement one work package (task, bug, user story, etc.). In most cases this means resolving the full work package and ideally that work package should be sized in a way that it can be fixed in a day. Therefore it is also dependent on how well issues are registered.
    - **Metric 1c**: Commit frequency. The same reasoning applies here. A high commit frequency signals that work packages are small enough to be done in a short period and that branches are not open indefinitely. Expect a

www.ebook3000.com

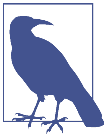
downward trend toward daily commits. Note that in some situations, this metric may be inaccurate due to, for example, bug fixes that require the use of specialized frequent commits, such as in high-security and mission-critical environments.

- **Goal B**: To understand the causes influencing team productivity (by analyzing issue resolution time).
  - **Question 2**: What is the typical issue resolution time for the system?
    - **Metric 2a**: Average issue resolution time, calculated as the total issue resolution time divided by number of issues. Expect a downward trend toward a relatively stable minimum amount of effort where the team can resolve issues efficiently.
    - **Metric 2b**: Percentage of issues that are both raised and solved within the same sprint. A high percentage could signify that bugs are processed immediately instead of postponed, which is desired behavior. Expect an upward trend.
    - **Metric 2c**: Average issue resolution time between different levels of urgency. The distinction is relevant because different levels of urgency may explain different resolution times. Expect a downward trend for each category.
  - **Question 3**: Are enough issues resolved within a reasonable amount of time? In this case, assume 24 hours is quick and 72 hours is acceptable.
    - **Metric 3a**: Percentage of issues resolved within 24 hours. Expect an upward trend until a stable rate has been achieved.
    - **Metric 3b**: Percentage of issues resolved within 72 hours. Expect an upward trend until a stable rate has been achieved.

Although issue resolution time is a common metric, it is important to realize its limitations. The metric could be distorted when issues are being defined in ever-smaller work packages (which makes it seem that efficiency rises), or if team members are closing unsolved issues instead of resolving them to clear the backlog. The latter may be a case of *treating the metric*. This is particularly a problem if there is incentive to make the metric more favorable. This would be especially at risk when team members' performance is evaluated with (mainly) this metric. Therefore make sure that you are not using only one metric (the pitfall *one track metric*), especially when that metric is important in evaluating performance. That can lead to the pitfall *treating the metric*, which causes unintended behavior.

Also note that productivity is not straightforward to define. It definitely is not only about lines of code written. Quality of the written code is a more serious consideration. For considerations on code quality, refer to "Related Books" on page x.

- **Goal B** (*continued*)
  - **Question 4**: How productive are we in terms of functionality implemented?
    - **Metric 4**: Velocity/productivity in terms of story points. Compare the average per sprint with the velocity measured over the system as a whole. Expect an upward trend initially as the team gets up to speed. Expect the trend to move toward a stable rate of "burning" story points each sprint (assuming similar conditions such as team composition). If version control is not present, we expect productivity to be lower, as it causes more overhead to check and merge code adjustments. The story point metric clearly assumes that story points are *estimated consistently* by the team. To use this metric we need to assume that on average a story point reflects approximately the same amount of effort.



Note that the initial measurements can serve as a baseline for later comparison. Often, the trend is more meaningful than the (initial) values.

4.4 Common Objections to Version Control Metrics

When you have a version control system in place and you adhere to the two most important conventions of version control, you are in a better position to measure the effectiveness of your development process. Common objections to the best practice in this chapter is that using different version control systems inhibits analysis or that specific measurements are unfeasible because commits are hard to trace to issues.

Objection: We Use Different Version Control Systems

"We cannot meaningfully measure branch lifespan because one part of the team likes Git while another prefers Subversion."

There seems to be a problem underlying this that is more important than the measurement. Measuring the effectiveness of your version control is unfeasible if you use version control in an inconsistent manner. Using different version control systems increases complexity for merging and partially offsets its advantages. As a team you will need to make a choice for one over the other version control system in order to achieve consistency.

www.ebook3000.com

Objection: Measuring the Recommendations Is Unfeasible (for Example, Whether Commits Are Specific)

"We cannot determine in all cases where specifically commits refer to. We would need to read through all commit messages."

Consider that it is a matter of discipline whether commit messages are specific. In order to find out whether commit messages are specific, you could sample commit messages or ask developers directly. Many version control systems help you to be specific in pushing changes by requiring a commit message, but this can also be enforced technically. This could be done by adding a pre-commit check in the version control system that checks whether the commit message contains a valid ticket identifier.

4.5 Metrics Overview

As a recap, Table 4-1 shows an overview of the metrics discussed in this chapter, with their corresponding goals.

Table 4-1. Summary of metrics and goals in this chapter

Metric # in text	Metric description	Corresponding goal
VC 1a	Number of developers that did not commit the last 24 hours	Preventing long-lived development branches
VC 1b	Average branch lifespan	Preventing long-lived development branches
VC 1c	Commit frequency	Preventing long-lived development branches
VC 2a	Average issue resolution time	Team productivity
VC 2b	Percentage of issues raised and solved within sprint	Team productivity
VC 2c	Average issue resolution time for each level of urgency	Team productivity
VC 3a	Percentage of issues resolved within 1 day	Team productivity
VC 3b	Percentage of issues resolved within 3 days	Team productivity
VC 4	Velocity in terms of story point burn rate	Team productivity

See also Chapter 5 on controlling different environments: development, test, acceptance, and production. This avoids surprises such as failed tests due to unequal testing and deployment environments.

Experience in the Field

We classify version control as a basic practice, as we believe it can be implemented fairly easily and is necessary for proper software development. In practice, proper version control actually seems an intermediate practice: only 41% of development teams fully adhere to version control best practices, as can be seen in Figure 4-1.

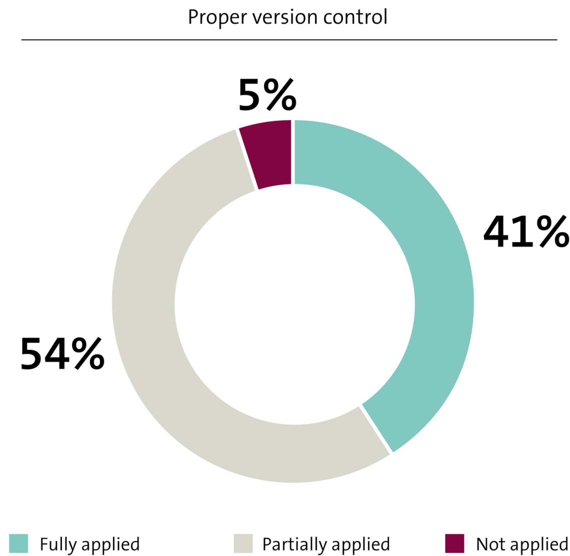


Figure 4-1. Benchmark results on proper usage of version control in development teams

We find that most teams initially apply version control properly, but find it difficult to control the more advanced aspects of the best practice, especially keeping development branches in order.

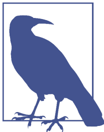
In one particular situation, we saw that a development team had installed a version control system, and that this initially helped them to keep track of the work and keep it all in sync. With business demands the team started growing until eventually, they needed to use branching to be able to simultaneously work on different parts of the codebase. With different styles of coding and long branch lifespans, half a year later they were in trouble: the different branches diverged too much to merge them; merge conflicts were inexorable and backporting notoriously led to regression issues.

www.ebook3000.com

CHAPTER 5

Control Development, Test, Acceptance, and Production Environments

> Testing shows the presence, not the absence of bugs.
>
> —Edsger Dijkstra, *Software Engineering Techniques*



Best Practice:

- **Create separate environments** for various stages in the development pipeline.
- Keep these environments as **similar as possible** and strictly **control the progression of code** from one environment to the next.
- This **improves speed and predictability** of development, because defects found in each environment can be **diagnosed clearly and fixed without disturbing the development flow**.

Imagine that in your development team, a sprint-specific Definition of Done is defined and a version control system is in place. The developers are now able to write code and quickly merge their versions. You notice that development of new features proceeds quickly, but the process of testing, acceptance testing, and production monitoring takes a long time to perform and validate. There is still a fair amount of code that was tested successfully in the test environment that fails during acceptance testing. The team encounters trouble when they need to fix bugs/defects for which they had written the fix long ago.

www.ebook3000.com

These are the kind of issues that may be caused by inconsistencies between different environments: Development, Test, Acceptance, and Production (DTAP in short). Let us briefly review these four environments:

- Development, in which developers modify source code. This environment is optimized for developer productivity.
- Test, in which the system is tested in various ways (e.g., validating whether technical requirements are being met). It is optimized for testing developers' products efficiently and in combination.
- Acceptance, in which it is verified whether user needs are met. It is optimized for mimicking the production environment as realistically as possible.
- Production, where the system is made available to its users. It therefore should be optimized for operation—that is, it needs to be secure, reliable, and perform well.

By controlling this "DTAP street" (the pipeline from development through production) you are in a better position to interpret problems. In particular, by ruling out inconsistent environments as a cause.

Controlling DTAP means defining, agreeing on, and standardizing three main characteristics:

- The configuration of different environments.
- The transfer from one environment to another.
- Responsibilities of each environment—that is, which activities are performed where (notably, whether fixes for issues found in later stages always need to be retested in earlier stages).

Making assumptions about the behavior of different environments causes trouble. Consider an example in which bugs are found in production after acceptance testing. If you can only guess that the acceptance and production environments are "fairly similar" in configurations, version numbers, etc., it will be a lot harder to find the underlying causes of bugs.

Take careful notice: for this best practice to work, the organization will need to be involved. Consider that an IT department could be reluctant to give up control over environments. But this separation of environments is a crucial step. Failure to have a proper separation of environments will jeopardize progress later!

5.1 Motivation

Controlling your DTAP street is useful for at least the following reasons:

- It clarifies responsibilities of the different development phases, avoiding undesirable behavior.
- It allows predicting the required effort of each phase of development and thus planning.
- It allows identifying bottlenecks and problems in the development process early.
- It reduces dependence on key personnel.

Consider that without defining different environments, testing, acceptance, and pushing to production "occurs somewhere, sometime." The DTAP environments are undefined (Figure 5-1).

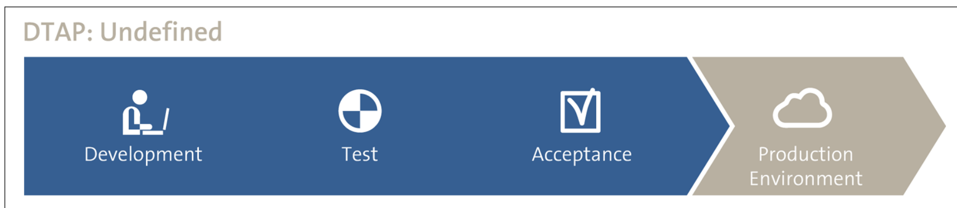


Figure 5-1. Uncontrolled DTAP

In this situation it is rather likely that the environments are set up and configured differently. They might be using the same resources (or not), they might use similar test data, they might be configured properly. Who knows? In fact, we see often that the answer to this question is unclear.

In many organizations the availability of these environments is an issue, especially for acceptance environments. Oftentimes we see that acceptance tests have to be reserved and planned well in advance. Similarly, the test environment is often shared with other systems. Without clear separation of capacity it will then be hard to tell whether performance test results are due to code changes or the environment itself.

In a highly controlled DTAP street (Figure 5-2), the environments' configuration and setup are predictable and consistent with agreements. Notably test, acceptance, and production environments should be as similar as possible. Separation between may be technical (e.g., transfer between different servers) or formal (handover), and it may be physical (different servers) or virtualized (a physical server with different instances).

www.ebook3000.com

Figure 5-2. Controlled DTAP

Controlled DTAP Clarifies Responsibilities Between Development Phases

Different development environments should separate concerns, just like good software should separate concerns in its implementation. Let us discuss the typical boundaries of different environments:

- A separation between development and test environments distinguishes clearly which code is ready for testing and which code is under development. Unit tests are commonly performed locally by developers, but these ought to be repeated in a test environment (typically managed by a Continuous Integration server).
- The separation between test and acceptance environments is needed to avoid time being wasted on verifying the system while the code is insufficiently tested and not ready. The test environment should be as similar to the production environment as possible in order to obtain realistic results.
- The separation between acceptance and production is needed to prevent code going to production that is insufficiently verified; i.e., the system does not behave the way it is supposed to. Typical examples of this are performance or reliability issues because of certain implementation/configuration flaws in the production environment.

Controlled DTAP Allows for Good Predictions

When you have a clear separation between responsibilities in the DTAP street, you can meaningfully measure the time spent in each phase. The more consistent the environments are among each other, the better you can compare those measurements. This is especially useful for making predictions and estimates for the time-to-market of new functionality. Clear separation of environment responsibilities facilitates accurate estimation of the lead times required for each development phase (typically, a division into the four phases of DTAP, but that can be more specific).

Controlled DTAP Reveals Development Bottlenecks and Explains Problems More Easily

When you have an overview of the time it takes to develop new features, you can track the time between the discovery of a bug and its introduction and the time it takes to resolve it. See Figure 5-3 for a simple visualization of such a feedback cycle.

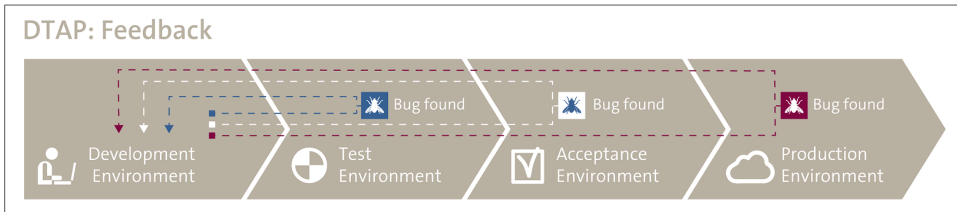


Figure 5-3. DTAP feedback cycles

By measuring you can verify the difference in effort when issues are identified early or late in the development process. Clearly, issues found in production take more time to fix than the ones found in the test environment. Therefore, you want to keep the loop as close as possible. And this is why effective automated testing will make a huge difference: it will identify issues early and effortlessly.

Controlled DTAP Reduces Dependence on Key Personnel

With more consistency between environments, less specialist knowledge is required to fix problems or set up infrastructure. This is generally true for standardization: consistency means less dependence on assumptions and specialist knowledge.

It is of particular interest because different specialties are involved: development, testing (including data, scenarios, scripting), and infrastructure. Consider how much expertise is necessary when none of these is standardized. Not all of this expertise might be part of the development team and therefore there may be dependencies on other teams, other departments, and even other companies (when infrastructure/hosting are outsourced).

5.2 How to Apply the Best Practice

Usually, the production environment is already clearly distinguished from the other environments in terms of organization. This is because often another department (operations) or another organization (hosting provider) is contracted to provide and manage the system's operation. To get the most out of the distinction between environments, you should:

- Separate concerns and restrict access to the different environments: developers should not *by default* be able to, for example, access the production environment.

www.ebook3000.com

For testing and acceptance environments, the distinction is less solid; e.g., when work is performed with cross-functional teams (DevOps).

- Only push code into the next environment when all tests have succeeded: when a test fails, the code should quickly return to the development environment so that the developers can fix the underlying cause. For consistency, bug fixes should be done in the development environment only. This seems evident, but we often see bug fixes being made in the acceptance environment and then backported (merging to an earlier version). See also Figure 5-3.
- Have the test environments resemble production as much as possible. This includes at least uniformity in versions, (virtualized) configuration of hardware, configured usage of frameworks, libraries, software, and having representative test data and scenarios. A particular point of interest is the actual test data. The ideal, most realistic dataset is an anonymized production copy, but this is often restricted because of security requirements or may not be feasible because of the production database size or other technical issues. For systems in which data integrity is especially important, one solution is to use an older backup that has lost its sensitivity.

5.3 Measuring the DTAP Street in Practice

Suppose that you have taken steps to make your DTAP environments clearly separated. You now want to identify the effect on productivity: the effort that the team needs to implement new features. Therefore, as team lead, you could come up with the following GQM model:

- **Goal A**: To understand productivity by gaining insight into effort spent within the different development DTAP phases.
  - **Question 1**: What is the average time spent in each development phase for a particular feature?
    - **Metric 1**: Time spent on realizing features/issues/user stories divided by each DTAP phase. This could be the average time spent from development to acceptance as measured by a Continuous Integration server (see Chapter 7). The time spent for the last push to production is typically manually registered. At the start you may find much effort concentrated in the acceptance environment. Gradually you may expect a distribution where the time spent in test and acceptance environments lowers, as tests improve and processes run more smoothly. Ideally, the maximum amount of time is spent in development producing new features (instead of fixing bugs that were developed earlier).

- **Question 2**: How many features pass through the testing environment but fail in the acceptance environment?
  - **Metric 2**: Percentage of rejected features/issues/user stories during acceptance testing. You could take the average for each sprint. Expect a downward trend. Investigate acceptance test rejections to see whether human error played a role, or unclear expectations, configurations, or requirements. This metric indicates how well code is developed and tested, but also signals how well the test and acceptance environments are aligned.
- **Question 3**: How long are the feedback cycles on average?
  - **Metric 3**: Time between checking in erroneous code into version control and discovery of issue. The timeline can be determined after analysis of the bug, sifting through version control. Tracing the issue is clearly easier if you follow the version control guideline on doing specific commits with issue IDs (refer back to "Commit Specifically and Regularly" on page 30). Expect this metric to follow the feature phase time, and expect a declining trend.

This model is initially designed to obtain a *baseline*: the initial value from which you can measure improvement. Consider the first metric: the time spent in each phase combined with the duration of feedback cycles gives you information on how well development moves through the DTAP street. In combination these metrics can help you understand where improvements are possible. For example, when issues seem to take a long time to resolve, an underlying problem could be that the team members are unevenly distributed over the different environments. It could be that there are too many developers compared to the number of testers, so that the testers cannot keep up the work, resulting in a longer phase time in the testing environment.

For this, you can use the following GQM model:

- **Goal B**: To understand whether the team is properly staffed/distributed over the DTAP environments.
  - **Question 4**: Is the development capacity balanced with the testing capacity?
    - **Metric 4a**: *Number of work items for testers*. Ideally, the amount of work should be stable and according to the testing capacity; that is, the metric moves toward a stable "standard backlog." If the backlog is rising it seems that testers cannot keep up with the load. That may lead to different causes (e.g., a lack of test capacity, a lack of test automation, or unclarity of requirements).
    - **Metric 4b**: *Workload for different roles (developers, testers, infrastructure specialists)*. Depending on the exact composition of the team, the distribution of the team might be uneven. This can be measured by monitoring

www.ebook3000.com

actual working hours (as present in your time registration). Comparable to the number of work items for testers, a growing number of working (over)hours signals that a backlog is building up. You would have to assume that team members are writing hours consistently and accurately.

— **Question 5**: Are issues *resolved* at a faster pace than they are created?

  — **Metric 5**: The sum of created issues versus resolved issues, averaged per week. This would exclude closed issues that are unresolved. A negative sum signifies that the backlog is shrinking and that the team can manage the issue load. Expect a downward trend that stabilizes at a point where the number of created issues is on average about the same as the number of resolved issues. That signifies that the influx of issues can be managed, even when taking into consideration peaks and troughs. In Figure 5-4, the surfaces signify whether more issues are resolved than created (green) or whether the backlog is growing (red).

Figure 5-4. Created versus resolved issues per day

With this GQM model you can determine whether a bottleneck exists in your staffing: when testing work is piling up, maybe more testers or different/better tests are needed. Or the other way around, when testers are not occupied, you may want to increase the number of developers. Over time you can determine norms that are con-

sidered "business as usual;" for example, a 1:1 relationship between development and testing effort.

Consider that it is advantageous to split functionality into small parts so that they reach the testing and acceptance phases more quickly. In terms of automation and tooling you can also improve the speed and predictability of code through the development pipeline with the help of the following:

- Automate your tests to speed up the two testing phases (see Chapter 6)
- Implementing Continuous Integration to improve the speed and reliability of building, integrating, and testing code modifications (see Chapter 7)
- Automate deployment (see Chapter 8)

These practices are discussed in the following chapters. Notice that test automation is placed before Continuous Integration because an important advantage of using Continuous Integration servers is that they kick-off tests automatically.

5.4 Common Objections to DTAP Control Metrics

Objections against controlling the DTAP street concern perceptions of slowing down development or the idea that it is unnecessarily complex to distinguish test and acceptance environments.

Objection: A Controlled DTAP Street Is Slow

"Controlling DTAP will actually slow down our development process because we need more time to move our code between environments."

There may be some overhead time to move code from the development environment to the test environment and from the test environment to the acceptance environment. But you "win back" that time when bugs arise: analyzing bugs will be faster when you know in what phase they occur, as it gives you information about their causes. Moreover, these transitions are suited for automation.

The pitfall of controlling different environments is to end up with a classical "nothing gets in" kind of pipeline, with formal tollgates or entry criteria (e.g., some sort of rigid interpretation of service management approaches such as ITIL).

www.ebook3000.com

Objection: There Is No Need to Distinguish Test and Acceptance Environments

"We can do all the testing, including acceptance testing, in one environment, so it is unnecessarily complex to create a separate acceptance environment."

An acceptance environment requires an investment to have it resemble the production environment as much as possible. But this has several advantages. First, you can distinguish better between integration issues (essentially of technical nature) and acceptance issues (which typically have a deeper cause). Second, it distinguishes responsibilities in testing. When acceptance testing starts, the technical tests should all have passed, which provides extra certainty that the system will behave well. Then, acceptance tests can make assumptions about the system's technical behavior. This narrows down the scope somewhat for acceptance tests. Of course, to what extent those assumptions hold depends on whether the configuration in the test environment is sufficiently representative of production.

5.5 Metrics Overview

As a recap, Table 5-1 shows an overview of the metrics discussed in this chapter, with their corresponding goals.

Table 5-1. Summary of metrics and goals in this chapter

Metric # in text	Metric description	Corresponding goal
DTAP 1	Average feature phase time	DTAP phase effort
DTAP 2	Percentage of features that fail during acceptance testing	DTAP phase effort
DTAP 3	Feedback cycle time	DTAP phase effort
DTAP 4a	Number of work items for testing	Team distribution and workload
DTAP 4b	Workload of different roles	Team distribution and workload
DTAP 5	Sum created versus resolved issues	Team distribution and workload

With a controlled DTAP street, you are in a good position to shorten feedback cycles and delivery times. The next three chapters are aimed at achieving those goals through automated testing (Chapter 6), Continuous Integration (Chapter 7), and automated deployment (Chapter 8).

Experience in the Field

In our development process assessment work, we check for the existence of separate environments, and we inspect how closely the production circumstances are simulated in the test environment (think data, infrastructure, configurations, etc.).

Having separate environments for development, testing, acceptance, and production is something we consider a basic practice: it is increasingly rare to see organizations in which that separation is not in order.

The difficult part is gaining control over the environments. There are many situations we have seen where test data does not represent production data well. This is not necessarily a problem, but can lead to unforeseen issues in production that you want to catch beforehand. We regularly see situations where development teams are not allowed to replicate all production data, either for privacy or for cost reasons. Those reasons may be valid. We stress that business representatives should make an explicit trade-off and determine whether the risks of nonrepresentative data (e.g., unpredictable reliability or performance) are acceptable.

Those risks vary with the system's usage intentions and requirements. A core banking application certainly needs representative test data for testing reliability, whereas an HR system that is allowed to be offline for a day may not benefit a lot from investing in creating a (mock) dataset.

Our benchmark shows that, even though we classify separation of DTAP as a basic practice, in reality it appears much harder. It appears as an intermediate practice with less than 40% not fully applying it (Figure 5-5).

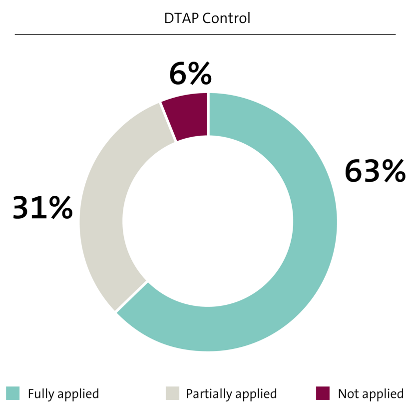
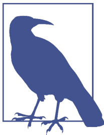


Figure 5-5. Benchmark results on proper control of DTAP in development teams

www.ebook3000.com

CHAPTER 6
Automate Tests

> Beware of bugs in the above code; I have only proved it correct, not tried it.
>
> —Donald Knuth



Best Practice:

- Write **automated** tests for **anything that is worth testing**.
- Agree on **guidelines and expectations for tests** and **keep track of test coverage**.
- Automated tests help to **find and diagnose defects early and with little effort**.

The advantages of automation are easy to recognize. Automated tasks are effortless to run. The more often an automated task is executed, the more you are saving. In addition to saving time and effort for repeating the same tasks, it also removes opportunity for error. Therefore it adds reliability to your development process. This leads to less rework and thus faster development. Of course, to automate something, you need to invest in the automation itself and its maintenance (maintaining test code, script, test data, etc.). After the investment, you save time at each repeat. You should aim to automate as much as feasible in the development pipeline, as that gives developers more time to spend on other, creative tasks. You should also make it easy for yourself by using testing frameworks and tooling as much as possible.

Testing tasks are excellent candidates for automation. Table 6-1 provides a summary of the different testing types.

www.ebook3000.com

Table 6-1. Types of testing

Type	What it tests	Why	Who
Unit test	Behavior of one unit in isolation	Verify that units behave as expected	Developer (preferably of the unit)
Integration test	Joint behavior of multiple parts (units, classes, components) of a system at once	Verify that parts of the system work together	Developer
End-to-end (or system) test	System interaction (with a user or another system)	Verify that system behaves as expected	Developer/tester
Regression test (may be unit/integration/end-to-end test)	Previously erroneous behavior of a unit, class, or system interaction	Ensure that bugs do not reappear	Developer/tester
Acceptance test (may be end-to-end test if automated)	System interaction (with a user or another system)	Confirm the system behaves as required	End-user representative (never the developer)

Note that in Table 6-1, only unit tests are *white-box* tests, in which the inner workings of the system are known to the tester. The other tests operate on a higher level of aggregation. Thereby they are making assumptions about the system's internal logic (*black-box* tests). Different types of testing call for different specialized automation frameworks. Test frameworks should be used consistently and, therefore, the choice of test framework should be a team decision. The way in which (testing) responsibilities are divided may differ per team. For example, writing integration tests is a specialized skill that may or may not reside within the development team, but unit testing is a skill that every developer should master.

6.1 Motivation

Automated tests increase confidence in the reliability of your code. Automated tests find causes of problems early and help to reduce bugs in your code.

Automated Testing Finds Root Causes of Bugs Earlier with Little Effort

Automated tests give more certainty on the root cause of problems because they are executed consistently. Therefore, if a test executes a piece of code at two different points in time yet gives different results, you know that something has changed in the system to cause that outcome. With manual tests you do not have the same amount of certainty.

Because automated tests generally run fast and their effort for execution is negligible, they allow for early identification of problems. This early identification limits the effort it takes to fix problems: when a bug is found later in the development pipeline it will certainly require more effort to fix it. Consider that the (development) capacity to fix problems is costly and scarce when a release deadline approaches.

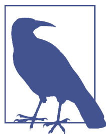
This is why acceptance tests should ideally be automated as much as possible. Functionality visible to users can be tested with frameworks that simulate user behavior and can "walk through" the user interface. An example is scripted user interaction through the screen, for which frameworks may provide a specific scripting language. This is especially useful for simulating browser interaction in web applications.

Automated Testing Reduces the Number of Bugs

Automated tests help your software become "more bug-free" (there is no such thing as *bug-free* software). Take, for example, unit tests and integration tests: they test the technical inner logic of code and the cohesion/integration of that code. Certainty about that inner working of your system avoids introduction of bugs (not all, of course). Therefore, unit tests allow for an effortless double-check of the entire codebase (isolated in units, of course), before turning to the next change.

Writing tests also has two side effects that help reduce bugs:

- By writing tests, developers tend to write code that is more easily testable. The act of designing the tests makes developers rethink the design of the code itself: clean, simple code is testable code. If a developer finds that units are hard to test, it provides a good reason and opportunity to simplify those units. This is typically done by refactoring—for example, separating responsibilities into units, and simplifying code structure. Writing tests thus results in code that is easier to understand and maintain: you easily consider that to be higher quality code. Some development approaches advocate writing a unit test before writing the code that conforms to the test (this approach is popularized as *Test-Driven Development* or TDD). TDD leads to writing methods that have at least one test case and therefore TDD tends to deliver code with fewer errors.
- Tests document the code that is tested. Test code contains assertions about the expected behavior of the system under test. This serves as documentation for the assumptions and expectations of the system: it defines what is correct and incorrect behavior.



Keep in mind that tests only signal problems; they do not solve their root cause. Be aware that a team may acquire a false sense of security when all tests pass. That should not release them from critically reviewing code smells.

www.ebook3000.com

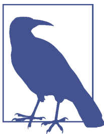
6.2 How to Apply the Best Practice

Based on our experience, we discuss the most important principles for achieving a great level of test automation. Come to clear agreements on tests and implement the right range of them. Make sure that those tests limit themselves to the things you want to test. Then plan and define responsibility for their maintenance:

Agree on guidelines and expectations for tests
: It is helpful to formalize test guidelines as part of the Definition of Done (see also Chapter 3). The principles described in this section further help you define what criteria should be adhered to for developing, configuring, and maintaining tests.

All relevant tests are in place
: The development team must agree on which tests need to be in place and in what order. Generally the order moves from detailed, white-box testing to high-level, black-box testing (see Table 6-1). Unit tests are most critical, as they provide certainty about the technical workings of the system on the lowest level. The behavior of code units is often the root cause of problems. You can therefore consider unit tests to be a primary safeguard against repeated defects (regression). But unit tests are no guarantee. This is because their isolated nature on the unit level does not tell you how the system performs as a whole. So higher-level tests always remain necessary.

  Therefore, the high-level tests such as integration tests and end-to-end tests give more certainty about whether functionality is broken or incorrect. They are more sparse as they combine smaller functionalities into specific scenarios that the system likely encounters. A specific and simple form of end-to-end testing is to confirm that basic system operations are in good order. For example, testing whether the system is responsive to basic interaction or testing whether system configuration and deployment adhere to technical conventions. They are commonly referred to as *smoke* tests or *sanity* tests. Typical examples of frameworks that allow for automated (functional) testing include SOAPUI, Selenium, and FitNesse.



The occurrence of bugs (or unexpected system behavior) is an opportunity to write tests. These tests will verify that the bugs do not reappear. Make sure to evaluate such situations, such that the team learns in terms of, for example, sharper specifications or requiring similar tests in the DoD.

Good tests have sufficient coverage
: A rule of thumb for unit test coverage is to have 80% line coverage. This refers to the percentage of lines of code in your codebase that are executed during those

unit tests. In practice, this roughly translates to a 1:1 relation between the volume of production code and test code. A 100% unit test coverage is unfeasible, because some fragments of code are too trivial to write tests for. For other types of tests, the most important test scenarios must be defined.

Good tests are isolated
: The outcomes of a test should only reflect behavior of the subject that is tested. For example, for unit testing, this means that each test case should test only one functionality. If a test is isolated to a certain functionality, it is easier to conclude where deviations from that functionality are coming from: the cause is restricted to the functionality it is testing. This principle is rather straightforward, but also requires that the code itself has a proper isolation and separation of concerns. If in a unit test the state or behavior of another unit is needed, those should be simulated, not called directly. Otherwise, the test would not be isolated and would test more than one unit. For simulation you may need techniques like *stubbing* (here: a fake object) and *mocking* (here: a fake object simulating behavior). Both are substitutes for actual code but differ in the level of logic they contain. Typically a stub provides a standard answer while a mock tests behavior.

Test maintenance is part of regular maintenance
: Written tests should be as maintainable as the code that it is testing. When code is adjusted in the system, the changes should be reflected in tests, unit tests particularly. Part of regular maintenance is therefore that developers check whether code changes are reflected in modified and new test cases.

6.3 Managing Test Automation in Practice

Imagine that your team has identified a backlog for developing tests. The team is starting up, so to say. Unit test coverage is still low, there are a few integration tests, and no end-to-end tests yet. The team therefore depends largely on feedback from the acceptance tests for identification of bugs. Sometimes those bugs are identified only in production. When (regression) defects are found, they go back to development. The team wants to catch problems earlier. Let us consider the following GQM model formulated from the viewpoint of the team lead. Again, note that initial measurements can serve as a baseline for later comparison. The ideal state would be defined in the DoD and an idea of the future state should appear from a trend line. Even when the ideal level for a metric is unknown, you can tell whether you are improving. Then, changes in the trend line ask for investigation. It is now important for you to understand how you can efficiently and effectively focus efforts on writing tests:

- **Goal A**: Knowing the optimal amount of automated tests (i.e., the point where writing automated tests costs more effort than the extra certainty they offer).

www.ebook3000.com

- **Question 1**: What is the reach/coverage of our automated tests for functional and nonfunctional requirements?
  - **Metric 1a**: *Unit test coverage*. Unit test coverage is a metric that is well suited for a specific standard. The easiest way to measure this would be line coverage, with an industry standard objective of 80%. The coverage may of course be higher if developers identify a need for that. A higher coverage gives more certainty, but in practice 95% is about the maximum coverage percentage that is still meaningful. That is typically because of boilerplate code in systems that is not sensible to test, or trivial code that does not need to be tested all the way.
  - **Metric 1b**: *Number of executed test cases grouped by test type*. Expect an upward trend. Clearly this metric should move upward together with the number of tests developed. However, there may be reasons to exclude certain test cases. Exclusion needs some kind of documented justification (e.g., excluding when test cases are currently not relevant).
- **Question 2**: How much effort are we putting into developing automated tests?
  - **Metric 2a**: *Number of test cases developed per sprint, ordered by test type*. Expect an upward trend until the team gets up to speed and gains experience with test tooling. Because different test cases/scripts are typically produced with different frameworks (e.g., performance, security, GUI tests), they have different productivity. Therefore they should be compared among their own type. After some fundamental tests are in place, expect the number of tests developed to drop over time. However, they will still be maintained. Source for this metric could be, for example, the Continuous Integration server or a manual count.
  - **Metric 2b**: *Average development + maintenance effort for tests ordered by test type*. Expect the *effort* for developing and maintaining tests to drop on average as the team gains experience and as some fundamental tests are in place. Maintenance effort could be a standard ratio of maintenance work or be separately administrated.
- **Question 3**: How much benefit do automated tests offer us?
  - **Metric 3a**: *Total development + maintenance effort for integration/regression/end-to-end tests divided by number of defects found*. Initially, expect a correlation between a higher test development effort and fewer defects found. This would signify that tests become better over time in *preventing* defects. As the system grows naturally, more defects may occur because of more complex interactions. Therefore, the effect between test development effort and declining number of defects will weaken over time. Unit tests may be excluded as should be part of regular development and there-

fore you can assume that time for writing unit tests is not separately registered.

— **Metric 3b**: *Manual test time in acceptance environment*. With an increase in test automation, expect a downward trend in manual test time. It may stabilize at some "minimum amount of manual acceptance testing." Some development teams do achieve a full automation of acceptance testing but your organization might not have that ambition and constellation. In particular, ambitions would need to translate to costs and efforts of automating acceptance tests, and that is rather uncommon. There is an advantage of keeping part of acceptance tests manual: keeping in contact with users with the help of demos after each sprint (sprint reviews) also provides unforeseen feedback on usability, for example.

— **Metric 3c**: *Manual test time in acceptance divided by number of defects found in acceptance*. This metric can help you to determine whether full acceptance test automation is a realistic goal. In general, the correlation is positive: the less time you spend on manual acceptance testing, the fewer defects you will find; the more time you spend, the more you will find. But with a high level of test automation, outcomes of additional manual acceptance tests should not surprise you. At some point you should find that extra effort spent in manual acceptance testing identifies no more bugs.

— **Metric 3d**: *Velocity/productivity in terms of story points per sprint*. You can compare the velocity within a sprint with the average velocity over all development of the system. Initially you may expect an upward trend as the team gets up to speed and moves toward a stable rate of "burning" story points each sprint. We also expect productivity to gain with better tests. The velocity may be adversely affected by improperly implemented version control (because of merging difficulties after adjustments). The story point metric clearly assumes that story points are defined consistently by the team, so that a story point reflects approximately the same amount of effort over time.

Achieving the goal of optimizing the number of automated tests (Goal A) requires careful weighing. The question of whether you are doing enough testing should be answered by trading off risks and effort. With a mission-critical system the tolerance for bugs may be much lower than for an administrative system that does not cause too much trouble if it happens to be offline. A simple criterion to determine whether enough tests have been written is answering the question "does writing a further test really isolate a use case or situation that is not covered by other tests?" If the answer is "no," your test writing efforts may be complete enough for the moment. However, tests need maintenance and fine-tuning for changing circumstances, such as adjust-

www.ebook3000.com

ing the datasets for performance tests when functionality has changed. So testing, just like development, never really stops.

One way of visualizing test coverage is by using a treemap report: a chart that shows how big parts of your codebase are and how much of it is covered by tests (they could be unit tests, but also integration tests). Figure 6-1 shows a simplified visualization.

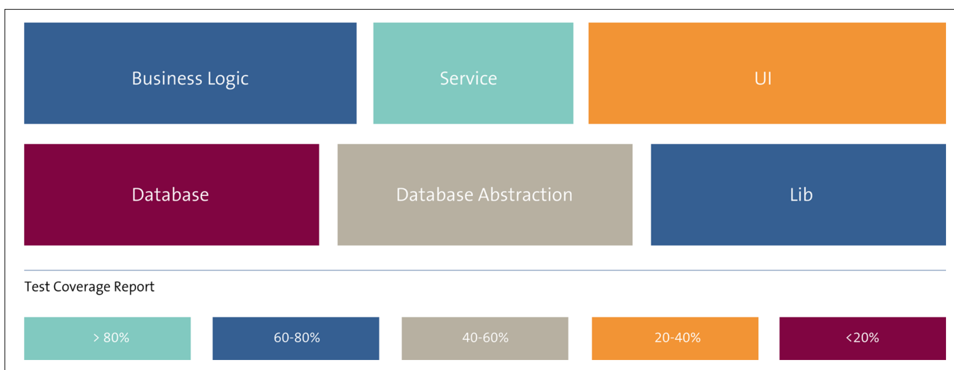


Figure 6-1. An example treemap test coverage report

Although such an image is not ideal for discovering a trend in your test coverage, it gives you insight into the overall test coverage of your system. In Figure 6-1, you can see that the database layer is hardly covered by tests, as well as the user interface part of the system. This does not need to be a problem, but asks for some analysis. Database-related code should normally not include a lot of logic and you should therefore expect tests in the database abstraction layer. Depending on your priorities, you may want to increase these test coverages or you may want to focus on getting, for example, the business logic layer fully covered.

Once you have more insight on the status of your test coverage, you can also better estimate the weak spots in your codebase:

- **Goal B**: Knowing our weak spots in code that automated tests should catch.
  - **Question 4**: Based on test results, how well is our code *guarded* against known defects?
    - **Metric 4a**: *Total of passed and failed tests ordered by test category*. Expect a consistently high ratio of passed versus failed tests. Failed tests are not necessarily a problem. In fact, if tests never fail, they may not be strict enough or otherwise they may not have proper coverage of functionality. A sudden increase of failing tests definitely demands investigation. It could be that newer tests are of lesser quality, but also that test assumptions have changed that have not been taken into account yet.

— **Metric 4b**: *Percentage of failed tests blocking for shipment (has highest urgency)*. Expect a consistent low toward zero. Deviations ask for investigation.

— **Metric 4c**: *Percentage of failed unit tests for each build*. Expect a consistent low percentage. Unit tests should fail sometimes, but they may signify that certain pieces of code are particularly complex or entangled. That may be a good starting point for refactoring efforts.

— **Metric 4d**: *Number of defects found in Test, Acceptance, and Production*. This applies to all defects such as disappointing performance or found security vulnerabilities, but notably regression bugs can be well identified and prevented by tests. Therefore, the ideal number is zero but expect that to never happen. Do expect a downward trend and decreasing percentages between Test, Acceptance, and Production. The later the bugs are found, the more effort they require to solve. They may also signify the complexity of the bugs and give you new information about how tests can be improved.

— **Question 5**: How well do we solve defects when they are identified?

— **Metric 5a**: *Average defect resolution time after identification*. Consider the average resolution time for the current sprint and compare with the averages of all sprints. Expect a downward trend over time toward a stable amount of time needed to resolve defects, as tests are becoming more advanced. When comparing trends on system or sprint levels, you can tell whether in the current sprint, solving defects was easier or tougher than normal.

— **Metric 5b**: *Number of reoccurrences of the same or similar bug*. Expect a downward slope with an ideal count of zero. Good tests particularly should avoid regression bugs from reappearing. This assumes mainly that reoccurrences are traceable. See also the following assumptions.

Assumptions Regarding These Metrics

Again we will make some assumptions about the nature of the metrics. These assumptions should allow us to keep the metrics fairly simple and help understand possible explanations of unexpected trend behavior.

For the metrics discussed here, consider the following assumptions:

- Defects cannot be avoided completely but good tests identify failures/regression bugs quickly. This shortens the feedback loop and thus the development time needed to fix a defect.

www.ebook3000.com

- On average, the odds of bugs and defects occurring within a system are roughly the same. We know this is not true in practice, because as a system grows in size and complexity, code tends to become more entangled and therefore there are more possibilities of defects occurring. However, we want to keep this assumption in mind because good tests should still ensure that these bugs are caught early.
- We ignore weekends/holidays for the defect resolution metrics. Refer back to the discussion in "Make Assumptions about Your Metrics Explicit" on page 15.
- Defects (in production) will be reported consistently when they occur. This assumption keeps us from concluding too quickly that a change in the number of defects is caused by a change in user behavior or defect administration.
- Defects are registered in a way that reoccurrences refer to an earlier defect with, for example, a ticket identifier.
- Defects are preferably registered with an estimate of resolution effort. However, it is generally beneficial to leave the defect effort estimates out of consideration, as otherwise we are essentially measuring how good the team is at effort estimation. What we really want to know is how good the team is at resolving defects. If those defects always happen to be tough ones, it is fair to investigate whether those large defects could have been avoided with code improvements and more, better tests.
- Hours of effort are registered in a way specific enough to distinguish different types of development and test activities, for example, as development time for tests, and time for manual acceptance tests.
- Writing unit tests is considered as an integral part of development and therefore not separately registered.

6.4 Common Objections to Test Automation Metrics

In this section, we discuss some common objections with respect to automating tests and measuring their implementation. The objections deal with the visibility of failing tests in production and the trade-off for writing unit tests.

Objection: Failing Tests Have No Noticeable Effects

"In our measurements we see that tests keep failing but they do not have noticeable effects in production."

Situations exist where tests fail consistently while their effects are unclear. When using a metric for the number of failing tests, this seems to distort measurements. Remember that outliers in measurement always warrant an investigation into the causes. Consider two examples:

Suddenly failing regression tests

Tests may fail because of changes in test data, test scripts, external systems' behavior or data, or the test environment itself (tooling, setup), etc. Changes in any of the aforementioned may have been intentional. However, they still ask for analysis of the causes and revision of the tests.

Consistently failing unit tests without noticeable effects

Failing unit tests are a showstopper for going to production! However, we know from experience that this occurs. An explanation can be that the failing unit tests concern code that does not run in production (and thus the test should be adjusted). It is more likely that effects are not noticeable because functionality is rarely used, or because the test is dependent on rare and specific system behavior. In fact, such rare and specific circumstances are typically the cases for which unit tests are insufficiently complete: tests should fail but they do not, because they have not been taken into account during development.

Thus, tests that fail but do not (immediately) have noticeable effects in production can still be highly useful tests. Remember that functionality that is rarely used in production may still be used at any moment and could even become frequently used in the future. Your automated tests should give you confidence for those situations as well.

Objection: Why Invest Effort in Writing Unit Tests for Code That Is Already Working?

"Why and to what extent should we invest in writing extra unit tests for code that already works?"

The best time to write a unit test is when the unit itself is being written, because the reasoning behind the unit is still fresh in the developer's mind. When a very large system has little to no unit test code, it would be a significant investment to start writing unit tests from scratch. This should only be done if its effort is worth the added certainty. The effort is especially worth it for critical, central functionality and when there is reason to believe that units are behaving unpredictably.

A common variation of this objection is that there is no time to write (unit) tests because developers are pushed to deliver functionality before a deadline. This is unfortunately common but very risky. We can be curt about this: *no unit tests means no quality*. If all else fails, the team may choose to write higher-level tests that add confidence on the functionality such as interfaces and end-to-end tests. However, without proper unit testing you will lack confidence knowing whether or not the system arrives at the right output by accident or not.

www.ebook3000.com

Make it a habit for all developers to review (or add) unit tests each time units are changed or added. This practice of leaving code better than you found it is known as the "Boy Scout Rule." Consider that it is especially worth refactoring code during maintenance when unit test coverage is low because the code is hard to test.

6.5 Metrics Overview

As a recap, Table 6-2 shows an overview of the metrics discussed in this chapter, with their corresponding goals.

Table 6-2. Summary of metrics and goals in this chapter

Metric # in text	Metric description	Corresponding goal
AT 1a	Unit test coverage	Optimal amount of testing (coverage)
AT 1b	Executed test cases	Optimal amount of testing (coverage)
AT 2a	Developed test cases	Optimal amount of testing (investment)
AT 2b	Test case development/maintenance effort	Optimal amount of testing (investment)
AT 3a	Development effort versus defects	Optimal amount of testing (effectiveness)
AT 3b	Manual acceptance test time	Optimal amount of testing (effectiveness)
AT 3c	Manual acceptance test time versus defects	Optimal amount of testing (effectiveness)
AT 3d	Velocity	Optimal amount of testing (effectiveness)
AT 4a	Passed versus failed tests	Robustness against bugs
AT 4b	Percentage failed blocking tests	Robustness against bugs
AT 4c	Percentage failed unit tests for each build	Robustness against bugs
AT 4d	Defects found in Test, Acceptance, Production	Robustness against bugs
AT 5a	Average defect resolution time	Defect resolution effectiveness
AT 5b	Reoccurrences of same bug	Defect resolution effectiveness

To fully benefit from automated tests you must integrate test runs in the development pipeline so that they are also kicked off automatically. Notably this concerns unit tests as part of the build cycle. Chapter 7 elaborates on this.

Experience in the Field

Because we believe that testing is paramount to high-quality software, we assess it extensively in our daily work. Notably we benchmark the maturity of an organization's testing strategy and the fitness of tests given the requirements.

We see that having a clear test strategy is already quite difficult: you need to think about the types of tests you perform, how much coverage you require, and then ensure that this strategy is known to everyone involved.

Requirements testing turns out to be an advanced practice, only fully applied by 13% of the teams. Often testing criteria are not defined at all, or they are not SMART. We also see cases where SMART test criteria are defined, test results are available, but they are not looked into until problems occur in production.

This leads to the benchmark result shown in Figure 6-2.

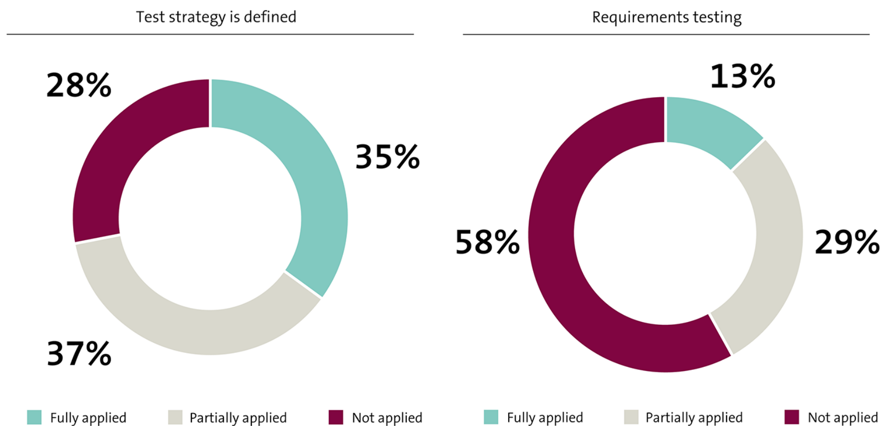


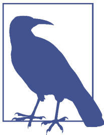
Figure 6-2. Benchmark results on proper (automated) testing in development teams

www.ebook3000.com

CHAPTER 7
Use Continuous Integration

Fall seven times, stand up eight times

—Japanese proverb



Best Practice:

- Achieve Continuous Integration by **setting up a CI server** after you have version control, build automation, and automated tests in place.
- Keep track of the **build time** and **test time**.
- This improves the development process because it **relieves developers of building the system** and it **shortens the feedback loop of issues**.

The term *continuous* in *Continuous Integration* (CI) says it all: continuously integrate (merge) code. CI puts into practice the idea that code integration and building should be done as often as possible. The integration happens at each commit to a CI server that merges the commit into a central repository. Fortunately, most CI servers do more: after each commit, they perform a clean checkout of the branch that was changed, build it, and perform the supplied tests (unit, integration, and regression tests). Running automated tests is not "required" in order to be called CI, but why would you not? Using a CI server is a great opportunity to have your tests run automatically. You can even automate further and run additional scripts after successful builds, to achieve automated deployment. The latter is a topic for Chapter 8.

www.ebook3000.com

7.1 Motivation

Continuous Integration speeds up development because it is fast, has short feedback cycles, and is more reliable than manual integration. Thereby it allows for further automation steps such as "continuous delivery."

CI Is Efficient

With test automation as part of the CI server, developers are relieved from tedious manual integration efforts and integration testing. Because automated tests are fast and effortless, you can frequently test your system. This controls the consequences of human error: instead of fearing that integration goes wrong, you merge and test code as often as possible.

Of course, when merge conflicts happen during a commit, the version control system asks the developer which change should take precedence over the other.

CI Gives Feedback Quickly and Thereby Speeds Up Bug Fixing

CI gives developers quick feedback on whether integration of code succeeds. Quick feedback means identifying problems early, when they are relatively easy to fix. Imagine that your team integrates all code at the end of the week, and it turns out that application builds fail because of integration errors. This can mean a lot of work and code reversions in order to get things to work again.

CI Is More Reliable Than Manual Merging

With CI, the building, merging, and testing process is consistent. As with all automated tasks, this consistency avoids a fair level of human error. A CI server should start processing a commit with setting up a clean checkout. That controls the risk that cached files or a corrupted database will influence the build process.

CI Facilitates Further Automation

Using a CI server paves the way for other kinds of "continuous" development such as "continuous delivery." Continuous delivery shares the goals of automating the development pipeline as much as possible and achieving short iterations of deploying new code. Continuous delivery implies that you can deliver each code modification into production. This demands a high level of certainty on whether the system is working as intended. Therefore, it is a horizon for developers as to what can be achieved by automating tests and deployment.

7.2 How to Apply the Best Practice

Before you can achieve CI, you need three things: version control, automated builds, and (obviously) a CI server. In particular, to get the most out of CI you need automated tests: CI without testing will do nothing more than tell you whether the code can be merged and compiled.

Requirement: Version Control

To integrate code automatically after each commit, you need a version control system. This is also necessary for the CI server to initiate a build on the central code repository. See Chapter 4 for elaboration on the use of version control.

Requirement: Automated Builds

The CI server will need build scripts and configurations (i.e., configuration of code structure and conventions) in order to execute builds. It is important that you are able to synchronize configuration files from developers, not just for Integrated Development Environments (IDEs) but also for databases and any other elements that may be customized for their development environment. In this way, newly introduced dependencies are taken into account when the CI server executes new builds.

Requirement: CI Server

A CI server is a dedicated server that checks out code from version control, uses the build scripts and configurations to execute builds, and performs tests after building. For basic setups and mainstream technologies, most CI servers already have default build tools, so this can save you time. For less common technologies, you may need to adapt your CI server configuration to accommodate the desired build tools.

Requirement: Automated Tests

Strictly speaking, when you have set up the previous parts, you already have Continuous Integration in place. But really you should add automated testing to the mix. A CI server is the ideal tool to perform all necessary unit, integration, and regression tests immediately and report the results to developers.

Additions to the CI Server

Most CI servers allow the execution of additional scripts, depending on the outcome of a build. So you can add deployment scripts on successful builds when you have automated your deployment process (this is the subject of Chapter 8). When the build fails, the CI server notifies you and points you to the failing tests. Typically you will have a quick overview of which unit tests have failed on which lines, and whether this was due to, for example, unexpected values or a compile/syntax error.

www.ebook3000.com

Important Best Practices for CI

Continuous Integration works by virtue of regular commits and quick builds.

If commits are not done on a regular basis—say, at least daily—you can still end up with integration errors that are hard to fix. After all, after a day a lot of time has been spent writing code that may break the build. The person fixing it (not necessarily the one who "broke it") has to go through a lot of code. So commits should be done regularly, as often as possible. This assumes that tasks can be broken down into pieces of work that take less than a day, but that is normally not a problem.

With branching, separate builds and tests become a greater concern. If developers use a new branch for every feature they implement, they should be built and tested through the CI server before they are merged into the master branch. This requirement can be configured in the CI server. In this way, you ensure that developers *must* have a successful build before they merge their new feature into the master branch. This is notably helpful when developers are making feature branches that need extensive testing (which may be the reason to use feature branches in the first place). This sounds trivial, but may be of particular concern when development teams are scattered geographically and/or have different release cycles.

When an application grows really large, it could be that build and test times increase to a point where it becomes impractical to always build after every commit. This should not happen too easily and it is a cue to revisit the design of the system to be able to build smaller parts. However, most CI servers can also cope with this technically. They allow parallel build jobs and dependency caching to speed up your builds. Of course, when changes are layered upon each other and build/test jobs take a long time, there is still a risk that after a certain change, all other changes need revisiting as well.

7.3 Controlling Continuous Integration

With the advantages of CI in speed, reliability, and flexibility, you would expect that your development process is speeding up with fewer issues. Fundamentally you would like to know whether and how productivity is improving. To have more confidence in a successful implementation of CI for your team, consider the following GQM model and metrics.

For the following metrics, the value is again mostly in large deviations, since these signal a problem. In agreement with the team you should determine a norm for which test and build durations are "short" enough.

- **Goal A**: Understanding how CI influences development productivity.
  - **Question 1**: Do our developers receive timely feedback on integration errors?
    - **Metric 1**: *Average feedback time after a commit*. Compared to a situation without CI, the feedback time will be lower, as CI deals with building and testing automatically. Over time, expect the trend line to go up: as the codebase grows, more tests are written and builds take more time. Because codebase volume of systems in maintenance is typically a few percent per year, those times should not raise alarms. The interest of this metric is mostly in large deviations, which warrant an investigation: is new development somehow too complex? Are tests being skipped?
  - **Question 2**: Can we integrate/merge fast enough?
    - **Metric 2a**: *Average build time* (comparing builds of the master branch). Expect a downward trend of this metric if you are comparing with a situation without CI. Expect a slight upward trend line over time as the size of the codebase grows.
    - **Metric 2b**: *Average test run time per type of test*. Expect this metric to be fairly stable, assuming that the different test types (e.g., unit tests, sanity checks, end-to-end tests) are internally consistent (i.e., for each test type, their tests are similar in size, complexity, etc.).

In Question 2, what is "fast" is typically a comparison with the baseline. Primarily, the feedback time depends on server processing time. So if the processing times are out of control, it will influence all of the above metrics. You will then need to speed up the CI process first before gaining benefits of quick feedback. Beware that speeding up build time at the expense of creating a clean environment may lower reliability of test results.

An easy way to see the progress of your CI server is to show the average build and test times per week. We show a simplified example in the following figure, where results are aggregated per week. Figure 7-1 shows an example chart that tracks the build and test time on a weekly basis.

www.ebook3000.com

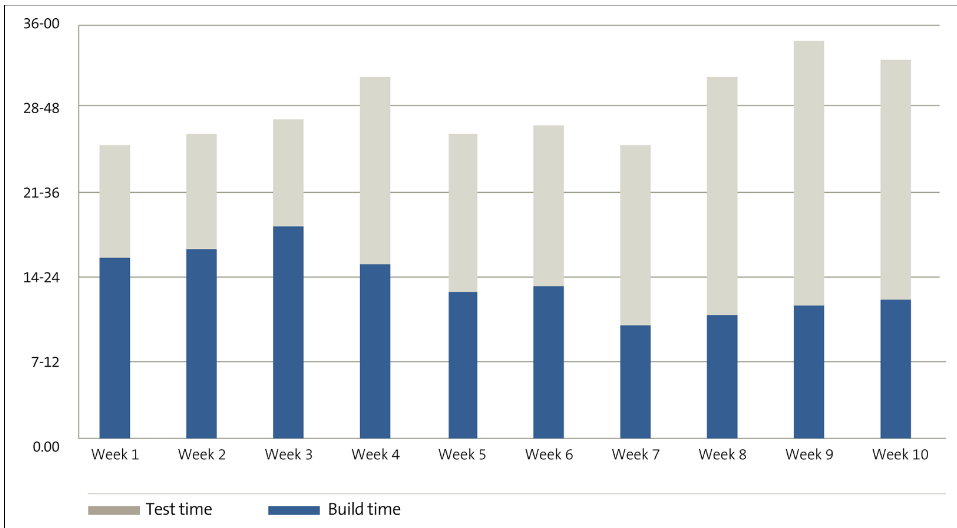


Figure 7-1. Trend line of build and test times

7.4 Common Objections to Continuous Integration Metrics

Typical objections to using CI metrics concern whether its processing time can be controlled and how one should deal with expectations of who will fix the build. They are discussed in this section.

Objection: We Cannot Get Control Over Our Build Time

"My CI server reinstalls all dependencies when a new build is available, so we cannot keep our build time low enough."

We do not wish to be puritans: if reinstalls keep you from doing regular builds, do not reinstall everything. For instance, a database does not have to be reinstalled every time you do a build as long as you clean and refill the database with data that is consistent with other environments. Of course, the ideal is that the test data realistically represents data in the production environment. Or you could opt for a special nightly build that carries out reinstallation of your dependencies. For added certainty it is more important to be sure that builds in the acceptance environment reinstall everything.

Objection: My Colleague Broke the Build, Not Me

"After my commit, the new build was broken, but I am sure my code is perfect. It is only because the code of my colleague in the other room does not align with mine that the build failed."

A broken build is a broken system, so this is an urgent matter. These are the situations that can be solved by having good agreements about what the leading principles of build success are. If you have agreed as a team that the tests determine build success, then your colleague committed code that has passed the test, so your code should be revised. If for some reason the (unit) tests are no longer adequate, then they should have been revised in the latest commit. Who will do the fix is a matter of agreement. Effective teams share responsibilities: if the task seems small, any developer with available time should feel responsible to take it up and fix it.

7.5 Metrics Overview

As a recap, Table 7-1 shows an overview of the metrics discussed in this chapter, with their corresponding goals.

Table 7-1. Summary of metrics and goals in this chapter

Metric # in text	Metric description	Corresponding goal
CI 1	Commit feedback time	Team productivity
CI 2a	Build time	Team productivity
CI 2b	Test run time	Team productivity

When you have CI in place in a stable manner, you have already made great progress toward continuous delivery and automation. In the following chapter, we will look at automated deployment: automating manual configuration of environments and their deployment.

www.ebook3000.com

Experience in the Field

Continuous Integration is one of the first steps toward an automated development pipeline. We check for the proper usage of CI by observing whether there is a CI server used for builds, and whether the server automatically starts a build after each commit. Additionally we check for automated test runs: when there are automated tests, does the CI server automatically perform those when a build has passed?

The results are shown in Figure 7-2. It turns out that having Continuous Integration is an advanced practice when we look at our collected data, although we would classify it as intermediate in difficulty. Combining CI with automated test runs seems to be slightly more difficult, but they correlate strongly: teams that *do* apply CI typically include automated tests as well.

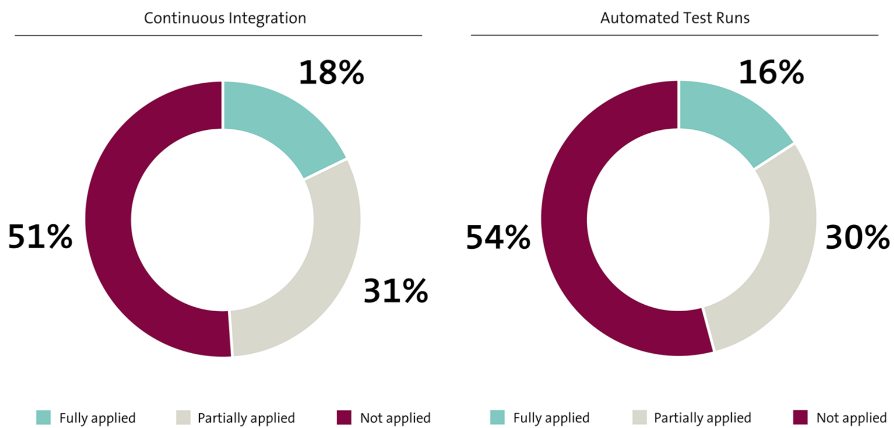
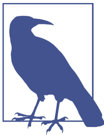


Figure 7-2. Benchmark results on Continuous Integration in development teams

CHAPTER 8

Automate Deployment

> There should be two tasks for a human being to perform to deploy software into a development, test, or production environment: to pick the version and environment and to press the "deploy" button.
>
> —Jez Humble and David Farley in *Continuous Delivery*



Best Practice:

- Use appropriate tools to **automate deployment to various environments**.
- Keep track of **deployment times**.
- This improves the development process because **no time is wasted on manual deployment tasks or errors caused by manual work**.

Once your DTAP street is under control, including automated tests that are kicked off in the development pipeline by a CI server, a further step for automation is *automated deployment*. In its general sense, here we mean automatically transferring code from one environment to the next one. Because this saves time for manual configuration steps, deployment times to new environments may drop from hours to minutes.

Consider the difference with Continuous Integration: CI is about automating builds and the tests that go with it, *within* the environment. Automated deployment is about automation of the deployment to each environment, typically when integration (including tests) has succeeded on the previous environment.

www.ebook3000.com

8.1 Motivation

When comparing automated with manual deployment, automated deployment is more reliable, faster, more flexible, and simplifies root cause analysis.

Automated Deployment Is Reliable

Ideally, deployment automation requires no manual intervention at all. In that case, once a build is finished and fully tested, the code is automatically pushed to the next environment. Such a process is typically controlled by a CI server. Because this is automated and repeatable, it minimizes the amount of (manual) mistakes and errors.

Automated Deployment Is Fast and Efficient

Automated processes run faster than manual ones, and when they need to be executed repeatedly, the initial investment to automate them is quickly earned back. Typically, deployment is a specialization of deployment/infrastructure experts who know their way around different environments. There is always "that one person" who knows how to do it. When this process is automated, such expertise is only needed for significant changes (in the deployment scripts) and root cause analysis of problems.

Automated Deployment Is Flexible

By extension of its reliability and efficiency, automated deployment makes a system more portable (the ability to move a system to another environment). This is relevant because systems may change infrastructure multiple times in their lifecycle (mainly the production environment, that is). Causes are various but often they are based on considerations of scalability (dealing with temporary high loads), transfer of a hosting contract to a new party, cost reductions, and strategic decisions (moving infrastructure to the cloud or outsourcing it altogether).

Automated Deployment Simplifies Root Cause Analysis

Given that automation makes for a reliable (or at least consistent) process, finding the cause of problems is easier. This is because it excludes human error. Causes are then more likely to be found in changed circumstances (e.g., different frameworks, data, configurations) that somehow have rendered the automation process outdated.

8.2 How to Apply the Best Practice

In this section, we discuss specifics on how to achieve automated deployment. The main principle behind this is that you need to think ahead about what steps need to be performed in the pipeline, and what that requires (e.g., in terms of configuration).

You will need at least the following controls:

Define your environments clearly
: Define for each environment its goals and intended usage. An inconsistent or unstable environment may disrupt your deployment process and makes root cause analysis difficult. It is preferable to use virtual environments that are stateless in the sense that they clean themselves up after use. By "reinstalling" them after use, you do not need to make assumptions about their configuration. See Chapter 5 for the separation of different environments.

Define all necessary steps
: What special needs does the system have for deployment? Are different versions run in production? If so, how do they vary? Use such assumptions and invariants for tests that check whether configuration and deployment are set up as intended.

Get your configuration right
: The more uniform development environments are, the better. Script configuration defaults in a uniform manner, so that an environment can be built/rebuilt quickly without human intervention. Your tooling should include provisioning functionality that checks (third-party code) dependencies and installs their appropriate versions.

Make sure you can do rollbacks early
: In case of unexpected issues in production, such as performance issues or security bugs, you need to be able to quickly revert to an earlier version of your system. Also allow for a rollback during the process—for example, when the deployment process freezes and you want to return to the begin state.

Use deployment-specialized tooling
: Tooling will help you supervise and configure deployment in a consistent manner (e.g., Chef). When portability to other environments has special interest, tooling that "containerizes" your application is especially useful (e.g., Docker). Such tooling can package deployment configurations, required software components, and other dependencies in a way that testing in different environments can be done consistently. This would benefit you if you intend to use, for example, different operating systems or different kinds of servers.

www.ebook3000.com

8.3 Measuring the Deployment Process

With deployment automation, you would expect that deployment times between environments will decrease. By extension you can also expect overall delivery time to decrease (from development to production). With added reliability you would also expect a decrease in the frequency and time to analyze deployment errors. Of main interest is then knowing whether deployment automation is helping you to achieve improved productivity and reliability. Then consider the following GQM model:

- **Goal A**: Understand the impact of automated deployment on development effort and reliability of the deployment process.
  - **Question 1**: How much time are we gaining by automating deployment?
    - **Metric 1a**: Difference between (former) average time spent and current time spent on deployment from acceptance (after a manual or automatic "go") to production. Expect an upward trend in the beginning because of the investment in writing deployment scripts, testing them, and setting up tooling. Then you should expect a downward trend as the scripts and process become more solid and dependable. If the savings are marginal, consider where most manual work is still required (e.g., troubleshooting, fine-tuning, error analysis) and focus on their automation.
    - **Metric 1b**: Amount of time it takes for code to move from development to production (as measured by the CI server/deployment tooling). When all tests (including acceptance tests) are automated, the gains in time will be apparent when you compare this metric with a process in which each step needs manual intervention. Expect a downward trend over time as the team gains experience with automating and less manual work is needed.
    - **Metric 1c**: Percentage of overall development time spent on deployment. By automating, expect the percentage to decrease over time. If absolute delivery time decreases together with this percentage (thus, the relative effort for deployment for development as a whole), you can assume you have implemented automated deployment well.

Consider a situation in which the average deployment time (the baseline) has been identified at 8 hours of effort and you want deployment time on average to stay below the baseline. A trend report may then look like Figure 8-1.

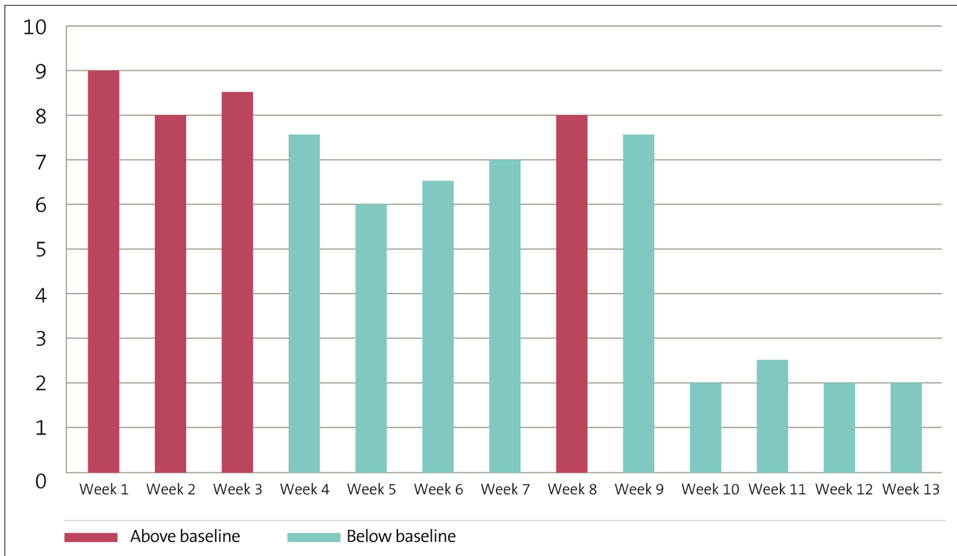


Figure 8-1. Deployment time after automation (in week 10, deployment scripts are ready and only minor fixes need to be performed)

- **Goal A** (*continued*):
  - **Question 2**: Is our deployment process more reliable after automating it?
    - **Metric 2a**: Number of deployment issues. Expect this number to drop. If you went from manual to automated deployment, typically you may expect a significant decrease in the number of bugs arising during deployment, simply because human error is much less of an issue. If this change is not visible, your deployment process may not be stable yet. That asks for issue analysis. It is most probable that the deployment scripts are yet unstable (faulty) or insufficiently tested. Deeper issues could be that the production environment is inconsistent over time or is inconsistent compared to test environments. You would probably have identified that before because it will probably distort other measurements as well.
    - **Metric 2b**: Amount of time spent on solving deployment bugs. Expect a downward trend line. This metric should follow the trend of the previous metric: having fewer issues means less time fixing them. If this is not the case, then bugs apparently are harder to fix when they appear. Then you should investigate why that is the case.

www.ebook3000.com

8.4 Common Objections to Deployment Automation Metrics

Common objections to deployment automation (and its measurement) are that it creates unnecessary overhead, that automated deployment to production is prohibited, or that the developer's technology of choice cannot be automatically deployed.

Objection: Single Platform Deployment Does Not Need Automation

"We only deploy on a single platform so there is no gain in automating our deployment process."

While generally it is easier to deploy on a single platform than on several platforms, the benefits of deployment automation still hold. Automated deployment will still be faster than manual deployment, it makes issue analysis easier (more structured), and allows for flexibility such as switching deployment platforms. Just like code itself, platforms requirements will change over time, though not that often. Consider deployment platform updates, new operating systems, switches to mobile applications, cloud hosting, 32-bit versus 64-bit systems, and so on.

Objection: Time Spent on Fixing Deployment Issues Is Increasing

"After automating deployment, we now spend more time on fixing deployment bugs!"

Automating deployment requires an elaborate investment. Time spent on fixing deployment issues is initially part of the deal. Gain experience by doing it more often. That experience will ease the process and solidify the scripts. This is one of the reasons why releases should be done regularly and frequently, given that a sufficient width and depth of tests add certainty on the system's behavior.

Objection: We Are Not Allowed to Deploy in Production By Ourselves

"For security reasons, we cannot deploy new releases of our software without permission of our user representatives."

This objection concerns the last step in the development pipeline in which code is pushed to production. However, the advantages of automating the pushes between development, test, and acceptance environments should still hold. When your team and the user representatives gain positive experience with the effects of deployment automation, eventually this last push may also be (allowed to be) more automated.

In practice we see that some large organizations use acceptance and production environments that are controlled by different parties or departments. These environments typically require strict requests for changes, due to concerns that changes hurt their KPIs such as uptime and number of incidents. Although deployment automation and

test automation should help manage those feared consequences, facing such a situation may be out of the span of control of the development team.

Objection: No Need to Automate Because of Infrequent Releases

"We only release a few times a year so there is little to gain by automating deployment."

Clearly, this should not be the case when you are practicing Agile development. We often hear this argument with legacy systems, though. Those are the archetypes of "anti-continuous-deployment." This situation is likely due to a combination of technology/system properties, available expertise, and constrained resources. The more infrequent your deployments, the more likely that they are difficult, requiring expertise, and thus expensive. Due to these costs, the business may not be willing to pay ("releases are really expensive, which is why we only have three of them"), which reinforces the cycle.

The "right solution" is to do it more often, to "diminish the pain of the experience." But when technical or organizational constraints force you to release only a few times a year, you may have bigger problems to solve in your development process.

Objection: Automating Deployment Is Too Costly

"It is very hard to get all the people and resources together to automate the deployment."

It could be the case that in your development you are dealing with multiple disparate systems that require manual steps for deployment. This is a particular type of complexity in which each system may require specific knowledge or skills. Automation of those steps and therefore of the process can seem costly in terms of resources. But it should also be evident that this complexity can benefit greatly from automation. Also, most systems can be automated at least partially. Those parts should be your initial focus to reap the immediate benefits.

8.5 Metrics Overview

As a recap, Table 8-1 shows an overview of the metrics discussed in this chapter, with their corresponding goals.

Table 8-1. Summary of metrics and goals in this chapter

Metric # in text	Metric description	Corresponding goal
AD 1a	Deployment time acceptance to production compared with baseline	Deployment effort and reliability
AD 1b	Deployment time to production	Deployment effort and reliability
AD 1c	Percentage of time spent on deployment	Deployment effort and reliability
AD 2a	Number of deployment issues	Deployment effort and reliability
AD 2b	Time spent on fixing deployment issues	Deployment effort and reliability

www.ebook3000.com

This is the last chapter on automation of the development process. After you have automated testing, integration, and deployment, you are right on track to deliver software (almost) continuously. In the subsequent chapters, we will look at best practices in software development from an organizational perspective. This includes practices for standardization (Chapter 9), managing usage of third-party code (Chapter 10), and proper yet minimal documentation (Chapter 11).

Experience in the Field

We consider deployment automation as a two-step process in practice: first we assess the level of automation for moving between environments, especially production. This can range from completely manually copying files from a local machine into a production environment (basic practice), semi-automatic deployment using a few deployment scripts (intermediate practice), up to complete automation where any commit triggers a build, a full test run, and a deployment given that all tests have passed. The latter is a very advanced and rare practice, though.

Then we also consider existence of patching and/or updating policies. This serves as a risk control of deployment reliability.

In the resulting graphs shown in Figure 8-2, we see that these two practices both get ranked as intermediate.

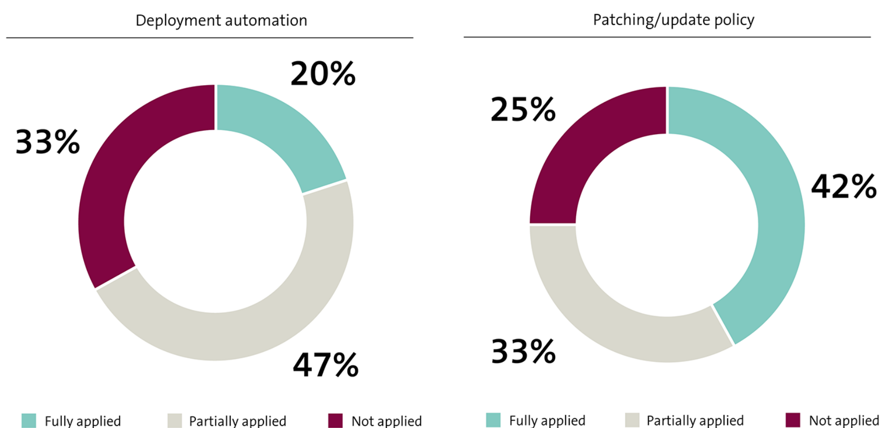


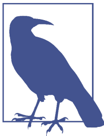
Figure 8-2. Benchmark results on deployment automation in development teams

Compared with deployment automation, a policy is relatively easy to implement but still ranks as an intermediate practice. One reason that a patching policy is also applied more often is that it is often part of contractual obligations with suppliers/clients, while deployment automation is almost never a contractual obligation.

CHAPTER 9

Standardize the Development Environment

> To accomplish anything whatsoever one must have standards. None have yet accomplished anything without them.
>
> —Mozi



Best Practice:

- Define and agree on standards in the development process.
- Ensure that developers **adhere to standards** and that **standards reflect the goals of the team**.
- This improves the development process because standards **help enforce best practices** and standards **simplify both the development process and code**.

This chapter deals with standardization in the development process. Standards make the results of development work more predictable. This predictability leads to a "hygienic development environment," in which developers work in a consistent manner. By this standardization, we mean four things:

Standardized tooling and technologies
: Agreeing on default tools and technologies, and not using a complex technology stack for solving a simple problem.

Process standards
: Agreeing on steps that developers need to perform during development.

Standardized coding style
: Defining a coding style (e.g., conventions for naming or indentation) and enforcing it.

www.ebook3000.com

Code quality control
: Defining and continuously measuring code quality in terms of complexity, readability, and testability of code.

Arriving at standards that are the best fit for your development team or organization is not trivial: if you put three developers in a room, they will usually offer three different ways to solve a certain problem, all three according to best practice. The team should ultimately agree on what is "best." Settling the issue is up to a quality champion, which typically is a team lead or (software) architect.

The following section explains the benefits of having these standards defined.

9.1 Motivation

Development standards make software more predictable, which eases maintenance. Standards also help in enforcing best practices, simplifying development and avoiding discussion overhead.

As a result, standards also ease comparison of metrics. Comparison allows for trend analysis and may be about comparing developers, teams, systems, or other factors. When the development process is consistent, there is less "noise" in observations that would otherwise make metrics hard to compare.

Development Standards Lead to Predictable Software Development

Standards lead to consistency and consistency is good for predictability. Therefore, agreeing on standards is a useful way to achieve predictable behavior and products (the software). The most important advantages for maintenance are:

- It lowers the effort to understand how other developers have made decisions about design and implementation.
- It makes it more likely that code and tests can be reused by others. Not by copying and pasting, of course!

Development Standards Help Enforce Best Practices

Standards should reflect best practices. With the exclusion of externally set standards (e.g., legal requirements), standards appear when a user community apparently prefers to do things a certain way. Therefore, standards reflect what commonly works well in practice.

Thus indirectly, standards help to enforce development best practices. For example, with consistent naming conventions, the relationships between different pieces of code (be it methods, classes, components) are clear. This helps dividing code into the right abstractions and allows, for example, for proper code encapsulation.[1]

A naming convention is a common type of standard. Identifiers that are very long or hard to understand are confusing for other developers, such as (a real example) `AddPaymentOrderToPayOrderPaymentGroupTaskHelper`. If those identifiers are named in a consistent manner, it serves as a system's documentation of itself, because they show how functionality is organized technically.

This consistency is important. Standards are only effective when they are applied consistently. And tooling can help you with that (by facilitating or enforcing). A common example is to have a CI server automatically check code commits for certain characteristics (such as an issue identifier) and based on the result, allow or deny the commit.

Development Standards Simplify Both the Development Process and Code

Standardization decreases the number of specific decisions that developers need to make. Variations in implementations complicate maintenance, such as using different technologies to solve a similar problem. Using a single technology per domain ensures that developers can understand each other's code, and learn from each other. This improves code reusability.

For large software development efforts it is common to have separate development teams working rather isolated from each other. Without standards, different teams may use a different development process and implement "custom solutions" that cannot be reused or they may duplicate functionality already present elsewhere.

Development Standards Decrease Discussion Overhead

Using standards limits discussion in development teams to the *exceptions* outside the standard. This is because with good standards, their context and application are understood. This eases understanding and communicating what is going on in source code. This also means that code is easier to *transfer* to new developers or other development teams.

1 Encapsulation here has a general meaning: being able to control access to a piece of code (e.g., an object) in order to isolate effects (e.g., for maintenance).

www.ebook3000.com

9.2 How to Apply the Best Practice

Without standards, questions arise such as *"What technology should I use?"* or *"How should I deal with version control?"* With development process standards, these are defined and agreed upon. The following best practices are in place to obtain clear standards.

Standardize Tooling and Technologies—and Document It

Standardization of tooling and technologies should include a summary description of their context. Context information should list a few simple situations with their solution. That leaves less room for developers to misunderstand in which situation (and how) they should use which technology.

Agreeing on standards verbally may not be enough to have a lasting effect. Therefore, centralize documentation of these standards (e.g., in a wiki or document). This allows for access from different teams that may be physically separated.

A clear standard for tooling and technologies includes the following:

- Prescribing a simple technology stack—for example, agreeing to use only one back-end programming language for implementing business logic.
- Scoping of technologies and frameworks—for example, using multiple specialized tools instead of using all features of one framework whose features might not be mature yet. Typically, areas such as testing, security, and performance warrant specialized tooling. This tends to require some deliberation, especially given cost trade-offs (licensing/open source). See Chapter 10 for elaboration.
- Default configurations of technologies or tooling (e.g., IDE, test suite, plug-ins). To achieve consistency, it is beneficial if default technologies and configurations are pushed over the internal network.

Chapter 11 elaborates on documentation efforts.

Considerations for Combinations of Technologies

Standards should be applicable and suitable to your organization and development team. The following are typical considerations for choosing combinations of programming languages and frameworks:

Assimilation
: Is it likely that one of the technologies will evolve quickly with advanced capabilities? Such that it integrates (assimilates) qualities from the other technology?

Compatibility
: How compatible are the different technologies? Do they require custom code to work together?

Time frame
: Are the technology's benefits really needed right now, or are you anticipating future demands?

Maturity
: How long has the technology been around? How viable is usage of the technology?

Independence
: Can different technologies be maintained independently or are they dependent on each other?

Specialized skills
: Do the different technologies require essentially different technical skills? Is it likely that one developer can work with the complete stack or is specific training needed (or even different staff)?

Testability
: Will the usage of different technologies significantly complicate testing?

Nonfunctionals
: Are there foreseeable effects on non-functionals, such as performance or security, when using multiple technologies? This is likely to be the case when you foresee that testability is negatively influenced when using certain combinations.

Defining Process Standards

Process standards describe the steps that should be followed during the development process. Think of standards such as:

- "Every commit should include an issue identifier."
- "Code should have a sufficient unit test coverage before it is committed."
- "Every implemented feature should be peer reviewed on code quality by a senior developer."

These examples could be enforced with pre-commit hooks that perform a check on the code commit before pushing it to the CI server. Peer review could be enforced by requiring authorization from a developer other than the one doing the commit. The aim of the peer review should be to check for adherence with standards, understandability, and general feel of code quality (is the code well tested, simple enough, isolated, etc?).

www.ebook3000.com

Though these are only a few examples, any of the best practices in this book can be standardized. When they are actions that developers must perform, you could also put them in your Definition of Done (see Chapter 3).

Coding Style

Most IDEs already have default style checking features that give warnings or even errors when your code formatting is not compliant with the predefined style configuration in the IDE. It is a good idea to use such a style checker, if only to be consistent in your code formatting. Do make sure that your style guidelines are not being suppressed. You would rather have an overview of violations than assuming that all code is according to standards. Commonly we see `CHECKSTYLE` tags such as the following:

```
// CHECKSTYLE:OFF
piece of code that violates coding style standards
// CHECKSTYLE:ON
```

Here, the style checker is bypassed by enclosing the violating code in between suppressing tags. This is something you should definitely avoid! There are several ways to facilitate this technically. For instance, pre-commit hooks can be configured to check for these tags or to ignore suppress messages.

Another good way to standardize coding style is to adhere to conventions and defaults rather than specific configurations. So, instead of writing lengthy mapping tables to relate the objects in your data access layer to the corresponding database table, you assume the convention that an object like "Book" will be mapped to a "book" table by default. This principle is known as *convention over configuration*, because you typically adhere to the convention and only violate it when exceptions occur. In general that leads to less configuration and less code. Using the conventions of the technology you use also improves transferability of the software, for example when hiring new developers. Conventions can be understood by newcomers to the team even when they originate from a completely different environment. Tooling and technologies help you in adhering to conventions with default presets.

Code Quality Control

What makes high-quality code? With maintainability as a quality aspect, some basic measurements for quality are system size in lines of code, lines of code per unit, the amount of duplication, and complexity of decision paths within units.

There are numerous static code analysis tools available for determining and enforcing code quality. Most of them can measure a few simple metrics such as size, duplication, and complexity of code, but do require some level of configuration. Combined with an IDE that includes built-in style checks, developers will get quick feedback on their code, which speeds up code improvement. Usually, style checks are performed immediately while coding, and code quality control measurements are executed in

the CI server. Quality control checks can also be checked before a release, but by adding preconditions on a build in the CI server, you can do quality checks automatically. You can even ensure that all code has passed quality control before it is built for integration testing. This is rather strict, but practice shows that this is the best way to obtain high-quality code because it cannot be ignored.

Perform code quality checks before a build is performed. In the most strict form of control you can choose not to build when the code is not up to code quality standards.

9.3 Controlling Standards Using GQM

Suppose that you have agreed upon standards for tooling and technologies and that now you want to put those into practice. The IDEs that developers use support code style checks such as duplication, unit test coverage, and complexity of units/methods. You know why it is useful to measure those software characteristics. But if standards for complexity and test coverage are enforced (e.g., checked by the CI server before a commit triggers a new build), you may want to know whether the standards are in fact appropriate. Therefore, consider the following GQM model:

- **Goal A**: To understand the effect of our development standards by monitoring adherence to those standards.
  - **Question 1**: Do developers adhere to process and coding standards?
    - **Metric 1a**: Number of process standard violations as a percentage of the number of commits. A violation here could be the omission of a peer review before committing code. You would expect this percentage to be low, such as 5%, but not 0%. If there are never any violations, then the standard may be too conservative. When there are many violations you will hear complaints soon enough, about the standards breaking the flow of work or being unreasonably strict. Both cases are worth investigating. The standard itself may be unfeasible, or the developers may lack certain skills necessary to comply with the standard.
    - **Metric 1b**: Number of coding style violations as a percentage of the number of commits. Expect a downward trend as developers learn the standard. If the standard is enforced by tooling, this can be near zero, because coding style can be corrected in real time.
    - **Metric 1c**: Number of code quality violations as a percentage of the number of commits. Expect this percentage to gradually decrease as the code quality increases.

www.ebook3000.com

The reason to normalize the number of code quality violations is that the number of violations alone may not be meaningful enough to the team: imagine a productivity peak in which the team produces much more code than usual. In that case the number of code quality violations will surely rise, even though the relative number of violations might decrease. We capture this in Figures 9-1 and 9-2. Although the absolute number of violations did not decrease dramatically, compared to the increasing number of commits we see that the relative number of violations decreases more quickly.

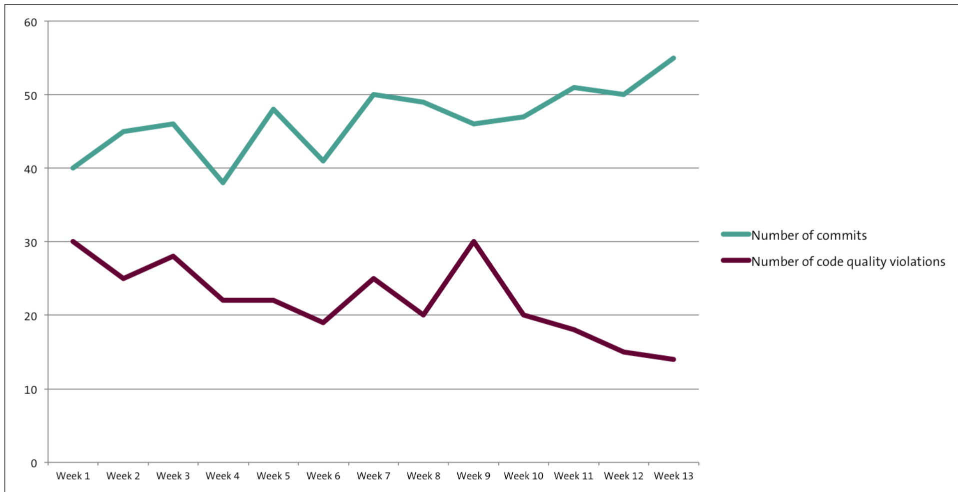


Figure 9-1. Absolute number of violations versus the number of commits

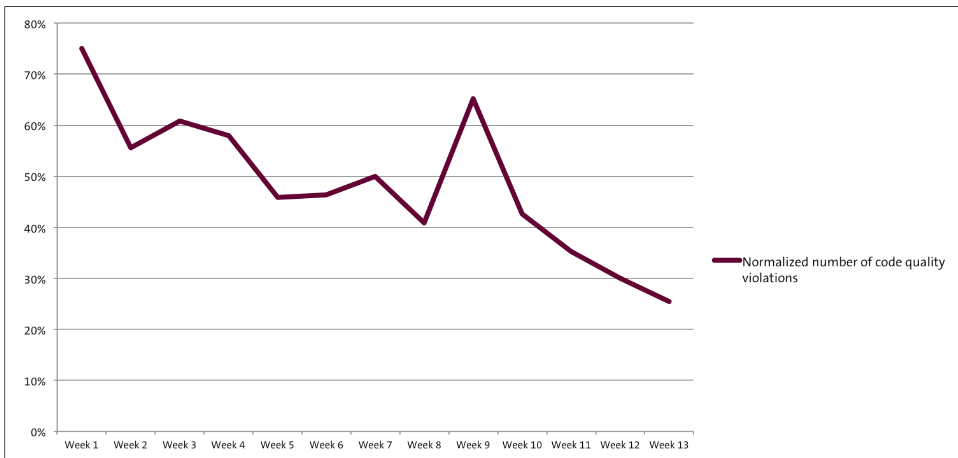


Figure 9-2. Normalized number of violations (as percentage of the number of commits)

In other words, it has become easier to adhere to code quality standards. This GQM model could be enforced automatically, if you wish to. In practice we see that this

works mainly when the team understands and agrees with the standards. Clearly, standards should not be too lenient, otherwise they do not lead to improvement. There should be a middle way to achieve this. You can measure perceptions manually with the following GQM model:

- **Goal B**: To understand the quality of our development standards by taking inventory of developers' opinions of them.
  - **Question 2**: Do developers think that the standards are fair?
    - **Metric 2**: Opinion per developer about the fairness of the standards, on a scale (say, from 1 to 10). The outcome of this measurement should be discussed with the team. When developers have widely different opinions, this may be caused by difference in experience. When the measurement shows that everyone agrees that the standards are unreasonable, you should distinguish between the parts that are unreasonable and the parts that you can keep. Then you can focus on tackling the standards that are unfair.
  - **Question 3**: Do developers find it easy or hard to adhere to the standards?
    - **Metric 3**: Opinion per developer about the effort to adhere to standards, on a scale (say, from 1 to 10). Again, different opinions may reflect difference in experience. The important observations come from the trend line. Maybe some developers do not find it easier to adhere to standards over time. Then consider pairing up more experienced developers with less experienced ones, which is a good practice anyway. For instance, you could define in your process standards that the commits of junior developers with less than 2 years of experience should be peer reviewed by a developer with at least 5 years of development experience. Or you could organize regular plenary sessions in which the top violations are discussed in terms of their causes and solutions. Then, everyone can learn from each other and code quality will increase.

It is important to monitor your standards periodically (say, quarterly), because standards should change (however slightly) over time to reflect new gains in experience and technology.

9.4 Common Objections to Standardization

Typical objections to standardization are that certain standards do not apply. You may choose to accept certain violations, as long as they are not being covered up. Keep in mind that standards are a reflection of how the team should work or wants to work. That means that the objections of single developers will not always make sense toward team goals.

www.ebook3000.com

Objection: We Cannot Work Like This!

"Our code keeps violating our standards. We cannot work like this!"

We do not expect that standards are a perfect fit right away, or that they should forever be held on to. Developers should understand the reasoning behind and the need for standards and what they intend to achieve. Resistance to standards may invite developers to circumvent them in clever ways. Hiding or not knowing violations is a bigger problem than knowing where violations occur. This may mean going back to the goals of the standards and possibly loosening them. Be careful when doing so, because having certain standards may serve a particular purpose. For instance, the standards you introduce may be high, but they reflect a quality level you want to have in the future. Then you should be explicit about this fact and have the team agree that they do not have to do it right first, as long as they understand this is something the team should work toward.

Objection: Can You Standardize on Multiple Technologies?

"Can we standardize on multiple technologies, such as using two programming languages for frontend code?"

Try to stick with one technology stack that is consistent, ideally with a single programming language. Simplicity is an important determinant notably for maintainability and security. Complex relations between technologies require a lot of testing and are at risk of breaking. There may be good reasons to use multiple technologies, to benefit from the specialties and strengths of each technology. For common considerations, see "Considerations for Combinations of Technologies" on page 82.

Objection: Organization Unfit for Standardization

"Our organization/project is not fit for standardization. We cannot get it through."

This is essentially a management issue. If formal standardization is not possible, an intermediate solution is to verbally agree on standards; for example, agreeing within your team to apply a standard within (part of) a system. If a "bad" standard applies consistently to a whole organization, it is generally better to adhere to it consistently than to ignore it with custom solutions. Deviating from an organization-wide standard tends to lead to problems in maintenance and failing interfaces, in addition to a lot of miscommunication and trouble with other departments.

9.5 Metrics Overview

As a recap, Table 9-1 shows an overview of the metrics discussed in this chapter, with their corresponding goals.

Table 9-1. Summary of metrics and goals in this chapter

Metric # in text	Metric description	Corresponding goal
ST 1a	Number of process standard violations	Adherence to standards
ST 1b	Number of coding style violations	Adherence to standards
ST 1c	Number of code quality violations	Adherence to standards
ST 2	Developer opinion about fairness of standards	Quality of standards
ST 3	Developer opinion about effort to adhere to standards	Quality of standards

In the next two chapters, we will further refine the hygiene of the development process by looking at two other kinds of standards: the usage of third-party code (Chapter 10) and documentation (Chapter 11).

Experience in the Field

The topic of standardization is very broad, and we could not claim that we assess all of its aspects of all the development teams we encounter. In process assessments, the question of standardization mostly relates to coding and building aspects about the use of a fit IDE, the use of issue tracking, and the use of coding standards and/or quality control. For all these practices, we observe whether they are used, but also whether they are used by the whole team. For example, issue tracking is often not used consistently. In many situations, issues are not linked to actual development work, making it harder to see the benefits of administering them.

The most difficult practice in standardization is consistent code quality control. Even if teams use standard coding style checks (say, for automatically formatting or completing their code), code metrics may be hard to implement. This makes the code quality practice an advanced one, whereas choosing a fit IDE and applying issue tracking are considered basic.

The results from our benchmark are shown in Figure 9-3.

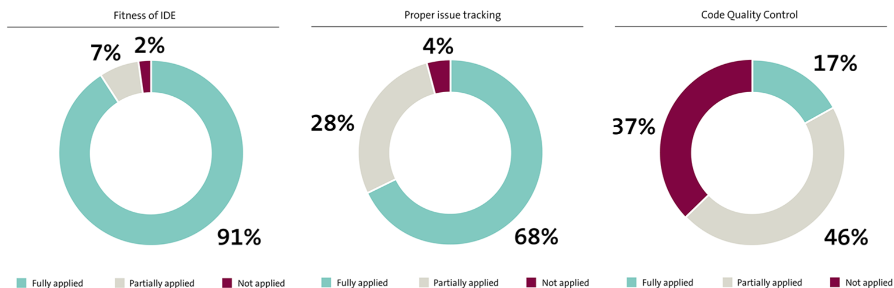


Figure 9-3. Benchmark results on standardization in development teams

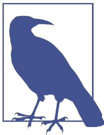
www.ebook3000.com

CHAPTER 10

Manage Usage of Third-Party Code

Oscar Wilde: "I wish I had said that." Whistler: "You will, Oscar; you will."

—James McNeill Whistler



Best Practice:

- **Manage the usage** of third-party code well.
- **Determine the specific advantages of using an external codebase**, and **keep third-party code up-to-date**.
- This improves the development process because **using third-party code saves times and effort**, and proper third-party code management **makes the system's own behavior more predictable**.

Third-party code is code that you have not written yourself. It comes in several variants: sometimes in the form of a single library, sometimes in the form of a complete framework. The code can be open source and maintained by a (volunteer) community, it can be a paid product that derives from open source code, or it can be a paid product using only proprietary code.

There is a lot of third-party code and there are good reasons to use it. However, managing the use of third-party code requires a policy based on the right requirements for your team and/or organization.

www.ebook3000.com

10.1 Motivation

While using third-party code may feel like being out of control (you are not the writer or maintainer of the code), it comes with a lot of benefits. For one, it saves you from having to "reinvent the wheel," and so saves you development effort. Usually, third-party code has also passed some form of quality control.

Using third-party code is, however, not trivial: external code can become outdated, cause security concerns, and thereby it may require additional maintenance. In complex cases the number of dependencies of your own codebase can explode. The decision to use third-party code is essentially a risk versus reward decision: the risk of security concerns or added maintenance versus the reward of reduced development time. So the usage of third-party code should be managed to ensure predictable behavior of that code.

Using Third-Party Code Saves Time and Effort

Using third-party code saves the development team time and effort in reinventing the wheel. It relieves them from writing custom code for which standard solutions are already present.

This is relevant as many applications share the same type of behavior. Consider basic needs for almost all systems: UI interaction, database access, manipulation of common data structures, administrating settings, or security measures. Using third-party libraries is especially helpful for such generic functionality, as it avoids unnecessary over-engineering. Widely known examples include web application frameworks such as Spring for Java, Django for Python, and AngularJS for JavaScript. For testing, there are the JUnit framework and its corresponding ports for other languages. For database communication there are also numerous frameworks available, such as NHibernate and the Entity Framework. So there is plenty of third-party code to choose from, saving you time and effort in the end.

Third-Party Code Has at Least Base-Level Quality

Third-party code is widely available as open source code next to commercial alternatives. All variations provide some form of quality control. Free open source solutions are maintained by the community: when that community is sufficiently large and active, they maintain and control the source code of the product. Although this is not a quality guarantee, the popularity of open source products suggests that many more benefit from its usage. When many users disagree with the quality or content of that product, typically this will lead to someone fixing the issues or someone making a development fork (a separately maintained version).

Typically, within paid subscriptions of open source derivatives, you have the possibility to request support in installing/using the products. In some cases you can communicate with the developers to report bugs or request improvements.

The commercial variations are hopefully quality controlled by the developers/organization itself. They clearly have the incentive to provide proper quality because they depend on satisfied clients and clients may expect proper quality. What is more, they constitute a professional organization that has some level of maturity, and so applies certain standards before a new version is released.

Managing Usage of Third-Party Code Makes a System's Behavior More Predictable

Having control over external dependencies makes the behavior of those dependencies more predictable. By extension, this also applies to your system as a whole: to know the expected behavior of third-party code helps to predict the behavior of your own system. With a clear overview of external dependencies, developers do not need to make assumptions about which library versions are used where. This gives a fair indication of how third-party libraries and frameworks affect the behavior of your own system.

It is rather common that different versions of a library are used within a system. This introduces unnecessary maintenance complexity. There may be, for example, conflicting behavior of libraries that perform the same functions. Therefore, the starting point for managing third-party code is to have an overview of its usage. That is, gaining insight on the specific frameworks and libraries and the versions that are used.

10.2 How to Apply the Best Practice

Making decisions on how to use third-party code is essentially a matter of standardization. It includes decisions on a general approach (policy, for which you should list general requirements) and specific choices (e.g., listing advantages and disadvantages for a particular library). Requirements include an update policy and maintenance requirements. We elaborate on these points in this section.

Determine the Specific Maintainability Advantages of Using an External Codebase

Using third-party code for general functionality is a best practice. That is, insofar as it decreases the actual maintenance burden on your own source code. Using third-party code is a good choice when it offers much functionality that can be delegated away from the system while it requires reasonable effort to adapt your system to it. For utility functionalities this is generally straightforward and those are a great opportunity. One should also use libraries for functionality that is complex but widely available as

www.ebook3000.com

a library. This is especially true for general security functionality (such as the algorithms for setting up secure connections between systems). They should never be written yourself, as it is hard to guarantee the security of that code.

In order to determine the quality of a library or framework, answer or give estimates for the following questions:

Replacing functionality
: Is the functionality that you are trying to implement utility functionality that is widely used by other systems? Is that functionality complex and specialized, i.e., is coding it yourself error-sensitive?

  - *Expectations:* Is it likely that functionality in the library will be expanded soon such that you can use it also to replace other functionality that you are coding yourself now?

Maintenance
: Does the specific third-party code have a reasonably active community with frequent updates? How widely is it used (e.g., number of downloads, number of forum topics, or mentions on popular developer forums)?

  - *Experience/knowledge:* Does the development team already have experience with the third-party code? Is knowledge readily available—is the code well-documented, either in the form of a book or online tutorials/forums?

Compatibility/reliability
: Is the third-party code compatible with other technologies used in the system and the deployment environment?

  - *Trustworthiness:* Has the source code been audited with good results? This is especially relevant for security functionality.
  - *Licensing:* Is the source code licensed in a way that it is compatible with your form of licensing?
  - *Intrusiveness:* Can the third-party code be used in such a way that your own code is relatively loosely coupled to the framework you use? Will upgrading not break your own code because of coupling? Every "yes" to these questions implies an advantage and argument for using that particular library/framework.

Keep Third-Party Code Up-to-Date

Updates are needed to stay up to speed with bug fixes and improved functionality. However, without a policy this tends to lag behind. Checking and updating libraries costs time and the advantages may not seem evident. Therefore, before creating a policy the following must be known:

Effort
: The amount of work required to perform updates each check/update cycle. By all means, automate as much as possible with tooling.

Priority
: What has highest priority in the system's development when it comes to updates? Typically, security has high priority and thereby policies tend to prescribe to always update libraries immediately when security flaws are patched. Consider that such a security emergency is especially hard to solve when libraries have been lagging behind major versions.

Then the policy should define how/when to check, and how/when to update. Note that the policy execution can be managed and automated with CI/dependency management tooling. (Examples of such tooling are Nexus, JFrog, Maven, and Gradle.)

How to check
: Automatically scan recency of updates daily or manually check it (e.g., at the start of a release cycle).

When to update
: Updating immediately when updates are available, or bundling all update work in a certain release cycle.

Update to what exactly
: Updating to the newest versions, even if that is a beta version, or to the latest stable version.

  - *Staying behind:* You may choose to consistently wait a certain amount of time before updating, for example, to see whether the community experiences updating problems. You might therefore choose to always stay one version behind the latest stable release.
  - *Not updating at all:* You may choose to remain at the current version and not update, for example, when updates introduce a notable instability concern with respect to your own codebase.

Note that libraries can become unsupported because users have moved to an alternative. Unsupported libraries run risks for compatibility (interacting with other tech-

www.ebook3000.com

nologies that are updated in the meantime) and security (because new flaws are not being fixed).

Ensure Quick Response to Dependency Updates

Regardless of your update strategy, you should be able to detect and perform library updates quickly. To this end, the intrusiveness of a library or framework is very relevant. If your codebase is tightly coupled with library code, it becomes harder to perform updates because you have to fix a lot of your own code to make sure it works as it did before. This is yet another reason to take unit tests seriously: when you update a library and several unit tests fail, there is a good chance that the update caused it.

If you decide that you want to keep up with certain versions of libraries, do the work as soon as you can. Postponing this increases the effort required to adjust your own code when the next update arrives.

Do Not Let Developers Change Library Source Code

Developers should be able to change and update libraries with minimal effort. Therefore, agree with developers that they do not make changes to the source code of a third-party library. If developers do that, the library code has become part of your own codebase and that defeats the purpose of third-party code. Updates of changed libraries are especially cumbersome and can easily lead to bugs. It requires developers to analyze exactly what has been changed in the library code and how that impacts the adjusted code. If a library does not perfectly fit the functionality you need (but it solves part of a difficult problem), it is easier to use it anyway and write custom code around it.

For large or complex functionality, it is well worth it to consider adjusting functionality to fit it to third-party code, instead of building a custom solution (or adjusting third-party libraries).

Manage the Usage and Versions of Libraries and Frameworks Centrally

To keep libraries up-to-date, you need an overview of what versions are used. Dependency management tooling can help with this. To facilitate library updates, a best practice is to use a central repository with all used libraries. The process *can* be fully automated. In that case, when a developer creates a new build, a dependency management tool retrieves and imports the latest versions of the required libraries.

You can also manually document the usage of types/versions of libraries and frameworks centrally, but in practice this is a huge maintenance issue. If the list cannot be fully relied upon, it loses its effectiveness completely.

10.3 Measuring Your Dependency Management

Suppose that you have a dependency management tool in place that automatically checks for and notifies you about new versions of libraries that the development team uses. You want to make sure that the team keeps updating their dependencies, but you also want to make sure that updates do not cost too much effort. Consider the following GQM model, to give you insight into the problems of the team and for checking whether your dependency management is done right:

- **Goal A**: To understand how the team manages the use of third-party code by assessing the effort required to keep the system up-to-date with external dependencies.
  - **Question 1**: How much time does the team spend on updating external dependencies and fixing potential bugs introduced to the system?
    - **Metric 1a**: Number of bugs found after updating a library. This metric will not be useful directly, but will provide useful insight over time. This is because it tells you something about how the team uses specific libraries: some will introduce more bugs than others. Expect the trend line to stabilize per library. If some library updates require a lot of work, you can investigate whether the team is, for example, behind with updating. A cause could be that issues are accumulating because developers do not have the time for updates or because the library is altered very often. Or possibly, the library is not very suitable for the system, in which case you should consider switching to another library.
    - **Metric 1b**: Number of versions the team is behind per library. This number may indicate trouble with updating when the team is working with libraries that have already had two major updates, for example. It could be that the team postpones it because they think it is too much work or that they cannot allocate time. Or they may have run into technical issues trying to update. In any case, when the metric signals trouble, you should find out what the problem is.

If you use this model to assess your library usage, you may discover that you need to update your standards for using third-party code, or that you are better off by switching to other libraries.

A good way to gain insight into your usage of third-party code is to create a chart of your dependencies that shows how far behind you are with them. For instance, the

www.ebook3000.com

chart in Figure 10-1 indicates the status of support for a list of libraries. For each library or framework the latest version is shown, together with the version that is currently in use. The colors show how far behind you are: blue means the framework is stable, while gray indicates that you should update to a newer version. Burgundy indicates an unsupported framework, meaning that you either should update as soon as possible, or consider switching to another framework.

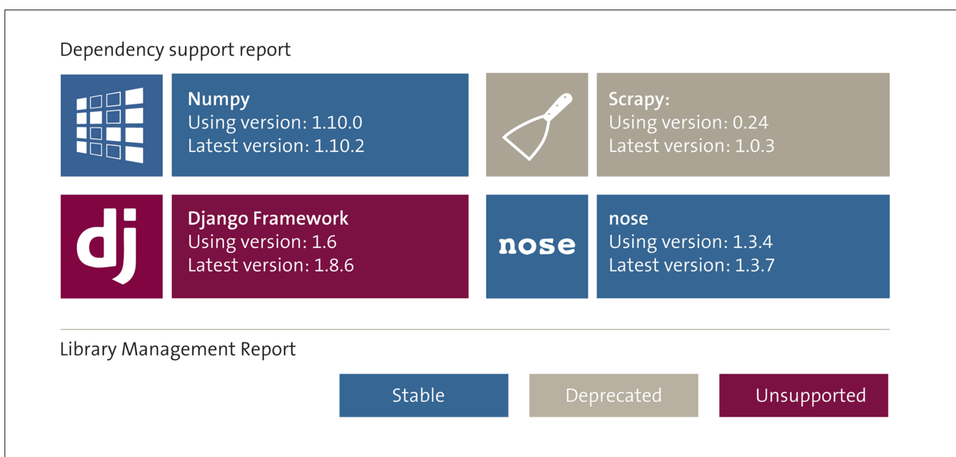


Figure 10-1. An example of library freshness

Suppose now that you need to select a new library for encryption functionality. This is a definite example of functionality you should never try to write yourself because of its complexity and impact of dysfunction. You may have found a few open source libraries, but want to select the right one. These considerations can be answered in a measurable manner:

- **Goal B**: To select the most suitable library by measuring its community activity and trustworthiness.
  - **Question 2**: For which of these libraries are issues fixed the fastest?
    - **Metric 2**: Per library, the issue resolution time. Consider this metric as an indicator for the level of support you can expect from the maintainers. That is, when you report a bug or an issue, how fast the maintainers will respond and solve the bug or issue. Of course, a lower issue resolution time is usually better. A codebase that releases very often can probably also fix bugs very fast.
  - **Question 3**: Which of the libraries is most actively maintained?
    - **Metric 3a**: Number of contributions per week, per library (assuming that the libraries are on version control that is publicly accessible so that you

can see the activity. Here, the number of contributions is a signal of maintenance activity).

— **Metric 3b**: Number of contributors per library. A higher number of contributors indicates an active community. Having several contributors work on one library also makes it more likely that some form of quality control is used. Be cautious, however, of too many contributors as this could also signify a distorted library. Too many contributors may also lead to forking when the contributors disagree about the contents.

This GQM model can help you in deciding which third-party codebase is most suitable for you. Remember that finding the best library is a trade-off: you should neither choose a library when you do not have the capacity to keep up with its pace, nor a library that is stagnant, as this may leave bugs unfixed for a long time.

When you have decided to use an external component, you may want to track the time you save on custom implementation and the time you spend on adapting to a framework or library. In this way, you gain insight into the benefits of specific third-party code. So you could use the following GQM model for this:

- **Goal C**: To determine the time gain of a specific framework by measuring the time spent on adapting to the framework versus the time saved on implementing custom functionality.
  - — **Question 4**: How much time did we save by implementing functionality using the framework?
    - — **Metric 4**: For each functionality that is using the framework, the estimated time it would take to implement it from scratch minus the time it took to implement using the framework. This metric gives a raw estimate of the time gain in using the framework. Therefore you should expect this value to at least be positive, otherwise it would mean that building it from scratch is faster!
  - — **Question 5**: How much time is spent on learning about and staying up-to-date with the framework?
    - — **Metric 5**: Time spent specifically on updates, or studying the framework. This time should be relatively high when you start using the framework (unless developers already know it, of course), and should gradually drop. The point is that it contributes negatively to the time you save by using the framework.

When you know how much time goes into using the framework itself and the time you save by delegating custom implementations, it is easy to predict your savings: just determine the break-even point and decide if it works for you. If you notice that it is very hard to save time on it, think about the other reasons for using the framework. If

www.ebook3000.com

there are no convincing reasons, then decide whether to keep using it or to switch to another framework.

10.4 Common Objections to Third-Party Code Metrics

Controlling the usage of third-party code is important from a maintainability perspective. However, common objections are concerns of their trustworthiness, maintenance benefits, and inability to update.

Objection: Safety and Dependability of Third-Party Libraries

"We cannot know whether third-party libraries are dependable and safe. Should we test them?"

As libraries are imported pieces of code, you cannot unit test them directly the way you do with normal production code. Also, once you have determined that the library is dependable and adequate, you should avoid efforts to test it yourself and trust it is working. However, testing is possible in the following ways:

- You could use the unit test code of the library itself to find bugs that may arise because your system has uncommon requirements or input. However, this mainly enables you to find bugs that the community should have found as well. Proper documentation of the library should clarify what it can and cannot do.
- If you have further concerns on a library, you could test it while abstracting the library behind an interface. This does cost you extra work and may lead to unnecessary code, so you should only do this when you have particular concerns about the library. It does, however, give you the possibility of easily switching to another library.

Objection: We Cannot Update a Particular Library

"We cannot update a certain library—doing so leads to trouble/regression in another system."

If the usage of a certain library leads to problems in another system, there is a deeper problem at work. The library may have changed significantly in a way that it is no longer compatible. In most cases, failing unit tests should signal what kind of functionality is in trouble. There could also be other reasons: young frameworks usually introduce a lot of breaking changes. If you do not have the capacity to keep up with such changes, this is all the more reason to switch libraries.

Objection: Does Third-Party Code Actually Lead to Maintenance Benefits?

"How can we determine whether using third-party code leads to benefits in maintenance?

As third-party code is not built by yourself, you do not know exactly how much effort was spent building it. But it is clear that for highly complex functionality it is much easier to import it than it is to code, test, and maintain it yourself. You do need to make sure that your system remains compatible with that library (which requires some testing and maintenance) and that effects of library changes are isolated in your code.

The important consideration is whether you are using your time to write code that makes your system unique and useful to your goals, or you are using that time to implement functionality that is already available in the form of a framework or library. It turns out in practice that a lot of functionality is very common among systems, so that most of the time there are already solutions readily available.

There are lots of ways in which you determine maintenance benefits: you can use stories or function points to determine how much work you save on implementing those while at the same time considering the investment you make to understand and use a third-party component. You can use the metrics provided in the previous section to gain insight into these times.

10.5 Metrics Overview

As a recap, Table 10-1 shows an overview of the metrics discussed in this chapter, with their corresponding goals.

Table 10-1. Summary of metrics and goals in this chapter

Metric # in text	Metric description	Corresponding goal
TPC 1a	Number of bugs found after updating a library	Third-party code effort
TPC 1b	Number of versions behind on a library	Third-party code effort
TPC 2	Issue resolution time per library-related issue	Library selection
TPC 3a	Number of contributions per week per library	Library selection
TPC 3b	Number of contributors per library	Library selection
TPC 4	Difference between time effort for implementing functionality from scratch and using a framework	Framework effectiveness
TPC 5	Time invested in studying or updating a framework	Framework effectiveness

www.ebook3000.com

This is the penultimate chapter that has dealt with standardization issues. The following and last best practice chapter covers (code) documentation.

Experience in the Field

Managing third-party code is hard in practice. In process assessments, we check for the existence of a library management strategy. Then, we assess whether the strategy is actually adhered to: are updates executed regularly and in line with the strategy? How many unsupported libraries are used? Is tooling supporting the library management strategy?

It turns out that this is hard for many development teams we encounter. Usually there is some kind of consensus about how and when to update libraries or frameworks, but this is not set in stone. We also see that developers have difficulty keeping third-party libraries up-to-date. Not only because they do not have the time to perform an update, but because they may need to change their own code as well and that time is not accounted for. The result is often that the used versions increasingly fall behind, until they become unsupported. Typically, the more versions you lag behind, the harder it is to get back on track. This requires discipline but teams would also be hugely supported by tooling telling them of version updates (or updating them centrally and automatically).

So it is unsurprising that this practice is an advanced one. So why this particular spread in the data? The answer is that completely doing it right is extremely difficult, but adhering to at least some basic principles of library management is easy. The distribution can be seen in Figure 10-2.

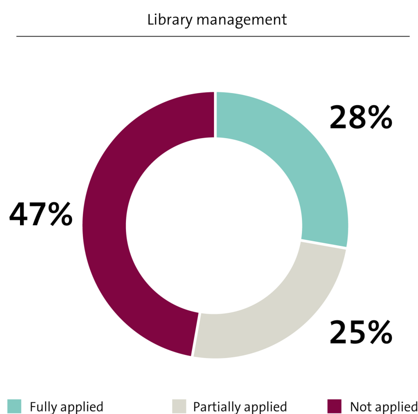
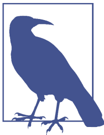


Figure 10-2. Benchmark results on third-party code management in development teams

CHAPTER 11
Document Just Enough

Je n'ai fait celle-ci plus longue que parce que je n'ai pas eu le loisir de la faire plus courte. (The only reason why I have written this long is because I did not have the time to shorten it.)

—Blaise Pascal



Best Practice:

- Document **just enough** for the development team to have a common understanding of design decisions and nonfunctional requirements.
- Keep documentation **current, concise, and accessible**.
- This improves the development process because documentation **retains knowledge** and **helps to avoid discussion**.

Some of the previous chapters have already dealt with documenting in some way, such as the Definition of Done, coding style standards, and third-party code policies. Because documentation is a form of knowledge transfer, consider that for all developers the following should also be clear:

Design decisions
: By this we mean the fundamental choices that are made about the subdivision of the system into components, but also how the system is implemented, how input is handled, and how data is processed. In this sense, design decisions cover high-level architectural choices as well as low-level design patterns. They are based on assumptions about the circumstances in which the system will run (for instance, requirements and criteria).

www.ebook3000.com

Requirements and criteria
: Functional requirements and nonfunctional criteria describe what the system should deliver when, and how.

Testing approach
: Decisions about what, when, and how to test.

Code documentation
: Code is ideally self-documenting, but for complex pieces of code you can supply reference implementations, and when supplying an API, you can use automatically generated API specifications.

We explain why good documentation is important, and how you can arrive at good documentation.

11.1 Motivation

Documentation is important for your development team, in order to have a common understanding of how the system is structured and how it should behave. Technical documentation is also important to retain knowledge of the system, whether it is for new developers or for the external world (in case others are to maintain the system as well). Documentation also helps you to see whether you are reaching your goals.

Documentation Retains Knowledge and Helps to Avoid Discussion

Documentation of your design, assumptions, and requirements is a means of knowledge transfer, in case new developers join the team or when your codebase becomes available to external developers. When introducing new developers, they should be able to get acquainted quickly with the system and its design.

When you have documented why certain design decisions were made, and explain the rationale behind these decisions, this helps avoid discussions later. That way developers can consider, for example, whether assumptions made in the past are still valid. When the code that is written is aimed to be exposed to the outside world with APIs, you require an API specification and greatly help others with a reference implementation.

Documentation Helps You to Know Whether You Are Reaching Your Goals

With an overview of (non)functional requirements you can assess whether the system is successful. Of course, requirements change over time, but they remain the standard by which you can measure system success.

11.2 How to Apply the Best Practice

Good documentation is concise, current, and complete, and these properties should be enforced by quality control.

Keep Documentation Current, Concise, and Accessible

Documentation is only useful if you can depend on a reasonable amount of currency. Therefore, good documentation is concise and is actively kept up-to-date (thus requires maintenance effort during development).

Documentation tends to be complete only up to a certain level of detail and therefore can remain concise. Indeed, it is not necessary to document all implementation details, as source code (structure) and tests should be self-explanatory. High-level design decisions such as technology choices should be documented in order to be reviewable when circumstances change. Such documents should in general not consist of more than 10 pages for any subject. If it does, it usually contains a high level of detail. This can be useful in some cases, but make sure these are documents to refer to from other, shorter documents. The general rule of thumb is that a document that is not a reference manual can be read and understood within an hour.

If documentation is unretrievable or scattered, it cannot be used properly. Therefore it is a prerequisite that documentation is stored and maintained in some central location that allows for access to those who need it (for proper security that should be on a need-to-know basis). This can be part of a central version control system, a wiki (that contains version control management), or similar other repositories.

Do not think of documentation as a "Write Once, Read Never" device. It should be actively maintained simultaneously with the system to be valuable.

Required Elements of System Documentation

All design choices should include basic assumptions underlying those choices. Those assumptions are based on the circumstances in which the system is available. For example, when defining a performance requirement one may assume a certain number of concurrent users and intensity of usage. Once those assumptions are documented, they can be reviewed, challenged, and improved when the circumstances change.

The form that documentation is presented in is up to the team. Typically, tests, use cases, and scripts describe well what the system should do and what it should not do. Consider that there are different things you want to know about a system as a devel-

www.ebook3000.com

oper if you are building on it. Therefore, in whatever form, system documentation should include the following:

Goals and functionality
: The goals of the system in terms of functionality and value for users and the organization as well as what functionality and behavior the system should provide to achieve those goals.

Internal architecture
: Division of the system in functional components. A well-designed source code organization should be able to show this division.

Deployment
: The way a system behaves can largely be influenced by the environment in which it runs. Therefore its deployment architecture and its assumptions should somehow be documented; e.g., visualized and described.

Standardization
: Agreements on decision choices for the technology stack, technology-specific usage, and the internal (code) architecture. For more on standardization, see Chapter 9.

Owner and process
: To ensure that the documentation is and stays current, each type of documentation needs an owner and a process understood by the maintenance team that they adjust documentation once assumptions, design choices and /or circumstances change significantly. For code, tests, and scripts, the one that is adjusting it tends to take responsibility naturally to update it (be it a document, comments, test cases, etc.).

Quality control is enforced to keep the nonfunctional requirements in check

After defining nonfunctional requirements, development should be actively controlled for adherence to those requirements. Preferably, this is an automated process (part of the development/test pipeline), but it should at least appear as part of the Definition of Done. In case of an automated process of quality control, the nonfunctional requirements also require a testing strategy. For more on quality control, refer back to Chapter 9.

11.3 Managing Your Documentation

Assessing your documentation is a delicate issue. With too much documentation, the maintenance burden becomes high, knowledge is harder to find, and often large documents are neglected. With too little, you are at risk of not knowing enough about the system, which may lead to making wrong decisions and assumptions. However, the code should be clear enough to document itself to a large extent. Perceptions of what is good documentation may also depend on experience. More experienced developers generally need less documentation to understand the technical intricacies of a system.

With the following GQM model we focus on the maintenance burden and the quality of documentation.

- **Goal A**: To understand the maintenance burden of documentation.
  - **Question 1**: How much time is being spent on maintaining documentation?
    - **Metric 1**: Number of hours spent on documenting. This assumes that developers can and do write hours specific for administration. This should neither be zero, nor a significant investment. Expect documentation investment to be high in beginning stages of system development, reducing to a stable figure. The risk notably is in arriving at "too much documentation to maintain" after which maintenance becomes neglected. What is "high" should therefore be discussed with the team based on this metric.
- **Goal B**: To understand the quality of our documentation (completeness, conciseness, and currency) by taking inventory of developers' opinions of them (after all, they are the ones using it).
  - **Question 2**: What do developers think about the documentation's quality?
    - **Metric 2a**: Opinion per developer about the quality of documentation overall.
    - **Metric 2b**: Opinion per developer about the documentation's completeness for their purpose.
    - **Metric 2c**: Opinion per developer about the documentation's conciseness for their purpose.
    - **Metric 2d**: Opinion per developer about the documentation's currency for their purpose.

www.ebook3000.com

These opinions can be rated on a scale (e.g., from 1 to 10). The outcome of these measurements should be discussed with the team. Just as with standards, when developers have widely different opinions, this may be caused by difference in experience. When the measurement shows that everyone agrees that the documentation is not of high quality, you know that improvements are necessary. This assessment could be repeated every six months, for example.

There is a clear trade-off between completeness and conciseness. There is also a relationship between the maintenance burden and currency. Finding the right balance is dependent on team preference and development experience.

11.4 Common Objections to Documentation

Common objections to documentation are that the team does not get the time to write documentation, that there is disagreement in the team, or that knowledge about the system is too scattered to write documentation.

Objection: No Time to Write Documentation

"We do not get the time to write documentation from the business!"

Consider that maintaining documentation is not a task in itself. It is part of regular development, when software changes thoroughly or when the software design has intricacies that are hard to understand quickly (for new developers). Documentation is not necessary for each modification, but lack of it may especially hurt future maintenance when modification concerns high-level design decisions. Consider also that the requirements or changes as defined by the business may themselves serve as documentation.

Objection: Disagreement in the Team

"My development team does not agree on the internal architecture of the system."

Documentation is something that the whole team should agree on. When there is friction between team members, you should appoint a quality champion within the team with the task of pulling the rest of the team along. Typically a lead/senior developer with a software architect's role can make the final decisions on architectural issues, in consultation with the team. If this is still difficult, consider having a workshop to evaluate the possibilities of revising the architecture or to explore the current issues.

Objection: Knowledge about System Is Scattered

"Knowledge about the system is too scattered to achieve consistent documentation."

Consider that you cannot document all knowledge. Start by documenting what you know now and write down the assumptions that you make under the current circumstances. This does require active maintenance when the situation changes or new knowledge is gained. This maintenance should be part of a standard (e.g., part of the DoD). That way, it becomes a shared responsibility for the team. In practice it works well if one person is assigned an authority responsibility to remind others to retain discipline to maintain documentation.

11.5 Metrics Overview

As a recap, Table 11-1 shows an overview of the metrics discussed in this chapter, with their corresponding goals.

Table 11-1. Summary of metrics and goals in this chapter

Metric # in text	Metric description	Corresponding goal
DOC 1	Number of hours spent on documentation	Documentation maintenance burden
DOC 2a	Developers' opinions on documentation quality overall	Documentation quality
DOC 2b	Developers' opinions on documentation completeness	Documentation quality
DOC 2c	Developers' opinions on documentation conciseness	Documentation quality
DOC 2d	Developers' opinions on documentation currency	Documentation quality

Because documentation should describe common understanding, this chapter is closely related to the topics of standardization (Chapter 9) and the usage of third-party code (Chapter 10).

www.ebook3000.com

Experience in the Field

When we assess the quality and extent of documentation of a system, we assess how balanced that documentation is: not too much detail, not too little. Typically we expect to see a general, high-level design document explaining the functional and technical architecture of a system as well as some details on its implementation. This commonly is in the form of a wiki. We take the assumption of a new developer that will be maintaining a system with little former experience with it. How well would that developer be helped with the available documentation?

The results show a fairly proportional distribution: about one third in our benchmark has been evaluated between having proper documentation (i.e., most areas are covered for maintenance), improper documentation ("not applied"), and a group in between, in which some key areas about the software system are missing or too comprehensive to be useful. See the details of this distribution in Figure 11-1.

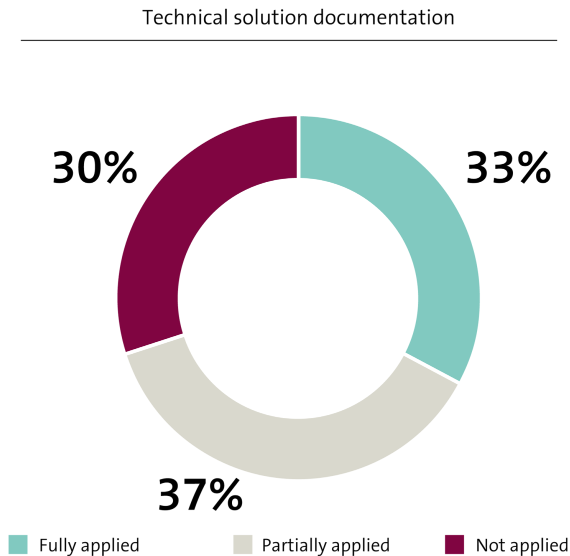


Figure 11-1. Benchmark results on documentation in development teams

CHAPTER 12
Next Steps

Le secret d'ennuyer est celui de tout dire.
(The secret of being boring is to tell everything.)

—Voltaire

Of course, there are more best practices for software development than those discussed in this book. At this point you will know the essential ingredients of a mature development process. The power of these practices and their metrics is that you can make improvement visible and manageable.

12.1 Applying the Best Practices Requires Persistence

We know, it is a bit of a downer. The best practices in this book will only work for disciplined and persistent teams. Persistence requires discipline and a belief that what you are doing is right. Discipline is not about being in the office at the same time every day or having a clean desk. It is about working consistently, standing your ground, and being able to say "for this issue we will not make an exception."

Persistence also implies patience. It takes time to master best practices and that will require an investment in time.

12.2 One Practice at a Time

We have presented our ten best practices in an order that reflects the typical order in which you want to accomplish them. For instance, Continuous Integration (Chapter 7) cannot possibly be effective without version control (Chapter 4) and to benefit fully from it, requires automated testing (Chapter 6). Of course, often they appear together. For example, when defining the DoD (Chapter 3), this is part of documenta-

www.ebook3000.com

tion (Chapter 11). In fact, all best practices in this book are standards themselves (Chapter 9).

It is more effective to implement one best practice in a complete and dependable way, instead of doing multiple ones partially. Partial implementation of best practices does not demonstrate their full benefits and that may be discouraging: it seems like you are investing all the time, and not reaping the advertised benefits.

12.3 Avoid the Metric Pitfalls

We know, the metric pitfalls ("Motivation" on page 10) are hard to avoid and their effects may not be visible immediately. Using the Goal-Question-Metric approach in a rigorous manner (Chapter 2) helps to define the right metrics for your purposes. Of particular interest are metrics that provide "perverse" incentives (that is, the opposite of what you are trying to achieve), leading your team to treat or misinterpret metrics in a manner that is not helping achieve the team goals. To find out whether this is the case, you will need to able to discuss this openly with the team. This may lead you to change or remove metrics. After all, if the metrics do not help you, change them!

12.4 What Is Next?

We hope that this book is helping you to improve the quality of your software, by applying the mentioned best practices. And that you have found or will find proper metrics to help you to confidently follow progress.

If you want to hone your individual coding skills, we refer you to our companion book *Building Maintainable Software: Ten Guidelines for Future-Proof Code*. For other process-related issues, consult the books listed in "Related Books" on page x.

Index

A

acceptance environment
- characteristics of, 38
- feature failures that passed in test environment, 43
- production environment boundary and, 40
- test environment boundary and, 40
- test environment vs., 46

Agile/Scrum methodology, 11
- (see also Scrum)
- Agile planning, 20
- DoD's origins in, 20
- short iterations, 20

assumptions, about metrics, 15-17

automated builds, 65

automated deployment, 71-78
- best practice application, 73
- CI vs., 71
- common objections to metrics, 76
- field experience, 78
- measurement of process, 74-75
- measuring reliability gained by, 75
- measuring time gained by, 74
- metrics overview, 77
- motivation for, 72

automated testing, 49-61
- assumptions about metrics, 57
- best practice application, 52-53
- CI and, 63, 65
- common objections to metrics, 58
- field experience, 61
- for detecting root causes of bugs, 50
- managing in practice, 53-58
- measuring benefits of, 54
- measuring development effort for, 54
- metrics overview, 60
- motivation for, 50
- optimal amount of, 53
- unit tests for code that already works, 59
- when failed tests have no noticeable effects in production, 58

automation, CI as facilitator of, 64

automobile industry, 1

averages, outliers vs., 16

B

baseline, 43

best practices
- automated deployment, 73
- automated testing, 52-53
- CI, 65
- controlled DTAP, 41
- development standards, 80, 82-85
- DoD, 21-23
- for documentation, 105
- GQM approach, 11-14
- implementing one practice at a time, 111
- metric pitfalls and, 112
- overview of, 6
- persistence in application of, 111
- third-party code, 93-97
- version control systems, 30-31

black-box tests, 50

branches, long-lived, 31

branching, 66

broken windows effect, 5

bugs, testing for (see automated testing)

builds

www.ebook3000.com

automated, 65
average time for, 68
CI and, 66
burndown chart, 20

C

car industry, 1
changes, tracking, 29
CI (see Continuous Integration)
classification, scaling of, 15
code integration, regularity of, 30
code quality control, 84, 89
code, testable, 51
codebase, estimating weak spots in, 56
coding style, 84
commit messages, version control systems and, 34
commits
CI and, 66
process standards and, 83
specificity/regularity of, 30
time between, 31
consistency, development standards and, 80
continuous delivery, 64
Continuous Integration (CI), 63-70
automated deployment vs., 71
best practice application, 65
common objections to metrics, 68
controlling, 66
dedicated server for, 65
field experience, 70
metrics overview, 69
motivation for, 64
control
defined, 2
measurement and, vii, 2
controlled DTAP, 37-47
best practice application, 41
common objections to metrics, 45
development bottleneck discovery with, 41
field experience, 47
GQM model for measuring in practice, 42-45
lead time predictions and, 40
metrics overview, 46
motivation for, 39
reduction of dependence on specialist knowledge, 41
separation of concerns between development phases, 40
speed of DTAP street, 45
test/acceptance environment separation, 46
convention over configuration principle, 84
cost, automated deployment and, 77
coverage, by good tests, 52
craft, 1

D

decision-making, metrics and, 17
Definition of Done (DoD), 24
Agile/Scrum background for, 20
best practice application, 21-23
common objections to using, 23-25
elements of, 22
extra work with, 25
field experience with, 26
making explicit, 19-26
motivation for using, 21
overhead and, 23
proving success with, 21
software quality assurance, 21
team responsibility and, 24
technical maintenance and, 24
tracking progress with, 21
DeMarco, Tom, vii
dependency management
reinstalls, 68
third-party code and, 97-100
deployment issues, 76
deployment, automated (see automated deployment)
derived metrics, 13
design decisions
documentation and, 105
documentation of, 103
developers, importance of contributions, 4
development best practices (see best practices)
development bottlenecks, 41
development capacity, test capacity balance with, 43
development environment
characteristics of, 38
test environment boundaries and, 40
Development Process Assessment (DPA), 5
development standards, 79-89
best practice application, 82-85
best practices enforced by, 80

choosing combinations of technologies, 82
code quality control, 84
coding style, 84
common objections to, 87
controlling using GQM, 85-87
defining process standards, 83
developers' opinion inventory, 87
discussion overhead and, 81
documentation and, 82
field experience, 89
metrics overview, 88
motivation for, 80
on multiple technologies, 88
organizational fit, 88
predictability and, 80
simplification of process/code, 81
standardization of tooling/technologies, 82
violations and, 88
Development, Test, Acceptance, and Production (DTAP)
controlled (see controlled DTAP)
controlled vs. uncontrolled, 39
Dijkstra, Edsger, 37
direct metrics, 13
distributed version control, 28
documentation, 103-110
best practice application, 105
common objections to, 108
discussion minimization, 104
field experience, 110
goal assessment and, 104
keeping current, concise, accessible, 105
knowledge retention, 104
level of detail for, 105, 110
maintenance burden of, 107
managing, 107-108
metrics overview, 109
motivation for, 104
required elements, 105
scattered knowledge about system and, 109
team disagreements on, 108
testing and, 51
time limitations, 108
DoD (see Definition of Done)
DPA (Development Process Assessment), 5
DTAP (Development, Test, Acceptance, and Production)
controlled (see controlled DTAP)
controlled vs. uncontrolled, 39

E

end-to-end tests, 52
environments (see also development environment; production environment; test environment; acceptance environment)
inconsistencies between, 38
separation of concerns between development phases, 40
expectations, deviations of metrics from, 16
experienced (subjective) quality, 4
explanations, judgments vs., 16
external dependencies, third-party code and, 97

F

Farley, David, 71
feedback
average cycle length, 43
CI and, 64
integration errors and, 67

G

generated code, version control and, 28
Git, 28
goal assessment, documentation as means of, 104
Goal-Question-Metric (GQM) approach, 5
as means of choosing metrics, 3
automated deployment, 74-75
automated testing, 53-58
best practice application with, 11-14
common objections to, 17
controlling development standards with, 85-87
deriving metrics from measurement goals, 9-18
documentation management, 107
for measuring DTAP street in practice, 42-45
for third-party code dependency management, 97-100
goal definition with, 11
making assumptions about metrics explicit, 15-17
metrics in, 13
motivation for using, 10
questions in, 13
using norms with, 16
version control and, 27-35

www.ebook3000.com

version control systems, 31-33
goals
definition for GQM, 11
deriving metrics from, 9-18
(see also Goal-Question-Metric approach)
five characteristics of, 12
GQM approach (see Goal-Question-Metric approach)

H

high-level tests, 52
Humble, Jez, 71

I

inconsistencies
between environments, 38
in metrics, 18
industrial process, 1
inherent (product) quality, 4
integration
speed of, 67
tests, 52
ISO 25010, x
quality in use model, 4
software product quality model, 4
software quality according to, 4
isolation of tests, 53
issue resolution time, 32, 44
iterations, short, 20

J

judgments, explanations vs., 16

K

kanban boards, 20
Knuth, Donald, 49

L

lead time predictions, 40
legacy systems, 77

M

maintenance
development standards and, 81
documentation and, 107
of tests, 53
third-party code and, 93, 93, 101
manual merging, CI vs., 64
maturity of team, 26
McCabe complexity, 13
measured observations, 3
measurement, 3
(see also metrics)
control and, vii, 3
scaling of classification, 15
Mercurial, 28
merging
CI vs. manual, 64
speed of, 67
version control systems and, 29
metric in a bubble, 10
metric pitfalls, 10, 112
metrics
assumptions about, 15-17
averages vs. outliers, 16
common objections to/solutions for, 18
common pitfalls in selecting/applying, 10
comparable meanings of, 15
decision-making and, 17
deriving from measurement goals, 9-18
(see also Goal-Question-Metric approach)
finding explanations for deviations from expectations, 16
trends vs. facts, 15
metrics galore, 10
mocking, 53
modification, independent, 29
Mozi, 79

N

norms, GQM and, 16

O

objective metrics, 13
one-track metric, 10, 32
open-source third-party code, 92, 98
outliers, averages vs., 16
overhead
discussion, 81
DoD and, 23

P

Pascal, Blaise, 103

patching policies, 78
permission, for deployment, 76
personnel, key
  automated deployment and, 72
  controlled DTAP and, 41
perverse incentives, 112
planning, Agile, 20
portability, automated deployment and, 72
precision, trends vs., 15
predictability
  development standards and, 80
  process and, 1
predictions, lead time, 40
process control
  as continuous activity, 2
  defined, 3
process frameworks, ix
process standards, 83
product (inherent) quality, 4
Product owner, 20
production environment
  acceptance environment boundary and, 40
  characteristics of, 38
  importance of resembling test environment, 42
productivity, functionality implemented as measure of, 33
progress, tracking with DoD, 21

Q

quality
  DoD and, 21
  experienced vs. inherent, 4
  importance of defining, 23
  ISO 25010 standard, 4
  unit tests and, 59
quality champion, 108
quality control
  code standards and, 84, 89
  documentation and, 106
  third-party code, 92
questions, in GQM approach, 13

R

regressions, 27
reinstalls, 68
releases, infrequent, 77
reliability, automated deployment and, 75
responsibility(-ies) of team, 24
rollbacks, automated deployment and, 73
root cause analysis
  automated deployment and, 72
  of bugs, 50

S

sanity tests, 52
  (see also end-to-end tests)
scaling of classification, 15
scope, DoD and, 22
Scrum, 19
  (see also Agile/Scrum methodology)
short iterations, 20
SIG (Software Improvement Group), x
single platform deployment, 76
SMART nonfunctional requirements, 26
SMART test criteria, 61
smoke tests, 52
  (see also end-to-end tests)
software development
  as observable process, 2
  best practices overview, 6
  importance of each developer's contribution, 4
  measuring/benchmarking development process maturity, 5
  separation of concerns between development phases, 40
Software Improvement Group (SIG), x
software quality, 21
  (see also quality)
  according to ISO 25010 standard, 4
  DoD and, 21
source code, independent modification of, 29
source files, merging of, 29
specialist knowledge
  automated deployment and, 72
  controlled DTAP and, 41
sprint planning, 20
stakeholder, 20
standardization
  of development (see development standards)
  third-party code and, 93
static code analysis, 84
story point, 20
stubbing, 53
style checkers, 84
style of coding, 84

www.ebook3000.com

subjective (experienced) quality, 4
subjective metrics, 13
Subversion, 28
success, proving with DoD, 21

T

TDD (Test Driven Development), 51
team maturity, 26
technical maintenance, 24
technology choices, documentation of, 105
test capacity, development capacity and, 43
Test Driven Development (TDD), 51
test environment
    acceptance environment boundary, 40
    acceptance environment failures, 43
    acceptance environment vs., 46
    characteristics of, 38
    development environment boundaries, 40
    importance of resembling production environment, 42
test failures, 42
test maintenance, 53
testing approach
    automated (see automated testing)
    black-box tests, 50
    end-to-end tests, 52
    high-level tests, 52
    sanity tests, 52
    smoke tests, 52
    third-party code, 100
    unit tests (see unit tests)
third-party code
    base-level quality of, 92
    best practice application, 93-97
    central repository for, 96
    change of library source code by developers, 96
    common objections to metrics, 100
    dependency management measurement, 97-100
    dependency updates, 96
    determining maintainability advantages of, 93
    field experience, 102
    maintenance benefit determination, 101
    management of usage, 91-102
    metrics overview, 101
    motivation for using, 92-93
    predictability of system behavior, 93
    safety/dependability, 100
    time/effort savings with, 92
    update management, 95
    update problems, 100
tooling
    automated deployment and, 73
    standardization of, 82
treating the metric, 10, 32
treemap report, 56
trends, precise numbers vs., 15

U

unit tests, 50, 52
    for code that is already working, 59
    quality and, 59
    third-party code and, 96
unsupported libraries, 95
updates
    deployment automation and, 78
    third-party code, 95
user acceptance, DoD and, 19
user stories, 20

V

velocity, of a development team, 20
version control systems
    automatic merging, 29
    basics, 28
    best practice application, 30-31
    CI and, 65
    code integration and, 30
    commit messages and, 34
    commits and, 30
    common objects to metrics for, 33
    field experience, 35
    GQM and, 27-35
    in practice, 31-33
    inconsistent use of, 33
    metrics overview, 34
    motivation for using, 29
    recommendation measurement, 34
versions, automatic merging of, 29
violations, development standards and, 85-87, 88
Voltaire, 111

W

Whistler, James McNeill, 91

white-box tests, 50

Wilde, Oscar, 1, 27, 91

www.ebook3000.com

Colophon

The animals on the cover of *Building Software Teams* are three species of pipit: water, Richard's, and tawny.

Pipits are slender songbirds who prefer open country. The genus is widespread, occurring across most of the world with the exception of the driest deserts, rainforests, and mainland Antarctica. Molecular studies of the pipits suggest that the genus arose in East Asia around seven million years ago and that the genus had spread to the Americas, Africa, and Europe between five and six million years ago.

The plumage of the pipit is generally drab and brown, buff, or faded white. The undersides are usually darker than the top, and there is a variable amount of barring and streaking on the back, wings, and breast. The mottled brown colors provide camouflage against the soil and stones of their environment. The pipit feeds primarily on insects, larvae, and plant matter (especially berries), by picking items from the ground or from low-lying vegetation as it walks. Pipits are monogamous and territorial. The nest tends to be situated on the side of a steep bank or in a hollow and is made from surrounding vegetation by the female. Females lay two clutches a year, consisting of 4 to 5 eggs, which are incubated for 15 to 16 days. The male and female both forage for their young and tend to feed them larger and slower arthropods that are easy to catch, in order to obtain the most food they can in the shortest amount of time.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world. To learn more about how you can help, go to *animals.oreilly.com*.

The cover image is from Lydekker's *Royal Natural History*. The cover fonts are URW Typewriter and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.