Coral Calero · Mario Piattini *Editors*

Green in Software Engineering

Springer

Green in Software Engineering

Coral Calero • Mario Piattini
Editors

Green in Software Engineering

 Springer

Editors
Coral Calero
Department of Information Technologies
    and Systems
University of Castilla-La Mancha
Ciudad Real
Spain

Mario Piattini
Department of Information Technologies
    and Systems
University of Castilla-La Mancha
Ciudad Real
Spain

Preface

Overview

Sustainability is being demanded by our society today; we have become aware of the need to cut down on our energy consumption and reduce our carbon footprint. At an international level, there is a whole host of initiatives trying to tackle these issues, and the main research and development programmes include sizeable amounts of funding for projects seeking to achieve environmentally sound technologies.

Information technology (IT) is a key component in reaching the above goals. The use of IT in obtaining systems that are more ecological (Green *by* IT) has indeed been seen to be significant, contributing to virtual meetings, dematerialization of activities, improvement in logistics, intelligent transport systems, smart grids, more sustainable management of (smart) cities, etc. We must be aware, however, that IT also has a negative impact on the environment (the amount of energy consumed by engineering equipment, processes and services). With this situation as the backdrop, there have been major efforts (Green *in* IT) made to reduce the energy consumption of the 'hard' part of IT (green data centres, green hardware, etc.). Even so, it has not been until recently that research has begun to be undertaken into how to achieve sustainable software (*green in software*) alongside sustainable software engineering (*green in software engineering*).

It is our firm conviction that we, as software researchers and professionals, are under an obligation to make those in positions of responsibility in government and in our organizations aware of the issues involved here. It is all about the vital importance of obtaining models, methods and tools that reduce the environmental impact both of the software life-cycle processes and of the software products that come into being as a result of those processes.

Good books dealing with the issue of Green IT already exist; in this work, we want to make our own small contribution to the attempt to raise the profile of green in software engineering. We want to make sure that it gets the consideration it deserves. To that end, we have brought together the main researchers in the field on this matter.

Organization

The book is composed of 13 chapters, structured in 5 parts that can be used as 'reading paths'.

The first part (Introduction) comprises one chapter written by Coral Calero and Mario Piattini, which introduces the main general concepts related to Green IT, discussing what green *in* software engineering is and how this is different from green *by* software engineering.

The second part (Environments, Processes and Construction) consists of three chapters. Green software development environments is discussed in Chap. 2 by Ankita Raturi, Bill Tomlinson and Debra Richardson. Chapter 3 describes a green software engineering process developed by Stefan Naumann, Eva Kern and Markus Dick. Green software construction, discussed by Fei Li, Soheil Qanbari, Michael Vögler and Schahram Dustdar, is the topic of Chap. 4.

The third part (Economic and Other Qualities) contains Chap. 5, contributed by Patricia Lago, Giuseppe Procaccianti and Héctor Fernández, which proposes using the e^3value technique to model and perform trade-off analysis between alternative green practices, particularly from an economic perspective. Chapter 6 by Juha Taina and Simo Mäkinen, which presents a layered model that gives some background, offers suggestions about measuring how well software supports green software engineering and software engineering for the planet.

The fourth part (Software Development Process) begins with a proposal by Birgit Penzenstadler (in Chap. 7) for incorporating environmental sustainability as an objective in requirements engineering from the very start, by using a reference artefact model. Chapter 8, written by Macario Polo, discusses how different approaches of test design and test execution may have an impact on the consumption of energy. Chapter 9, written by Ignacio García-Rodríguez de Guzmán, Mario Piattini and Ricardo Pérez-Castillo, presents useful techniques, tools and practices for improving software sustainability in existing software systems. In Chap. 10, Coral Calero, Mª Ángeles Moraga, Manuel F. Bertoa and Leticia Duboc show how to include green aspects of a software product within its quality, while Chap. 11, written by Mª Ángeles Moraga and Manuel F. Bertoa, presents the main measures for green in software engineering.

The final part (Practical Issues) begins with Chap. 12, in which Qing Gu, Patricia Lago and Paolo Bozzelli propose a decision-making model for adopting green ICT strategies, while Chap. 13 by Martin Mahaux and Annick Castiaux discusses the participation and open innovation in/for sustainable software engineering.

As the reader will realize, we have tried to follow the structure of the SWEBOK,[1] attempting to cover most of the key areas (KAs) involved in the incorporation of green aspects in software engineering. In Table 1, we summarize in which chapter the content of the corresponding KA is covered (directly or indirectly).

---

[1] SWEBOK V3.0. Guide to the Software Engineering Body of Knowledge. Bourque, P. and Fairley, R.E. (eds.), NJ., IEEE Computer Society. 2014.

Table 1 The book chapters and the SWEBOK KAs

KA number	SWEBOK KA	Chapters numbers
1	**Software Requirements**	**7**
2	**Software Design**	**8**
3	**Software Construction**	**2, 4**
4	**Software Testing**	**8**
5	**Software Maintenance**	**9**
6	**Software Configuration Management**	**–**
7	**Software Engineering Management**	**–**
8	**Software Engineering Process**	**2, 3, 13**
9	**Software Engineering Models and Methods**	**12**
10	**Software Quality**	**6, 10, 11**
11	**Software Engineering Professional Practice**	**12**
12	**Software Engineering Economics**	**5**
13	Computing Foundations	
14	Mathematical Foundations	
15	Engineering Foundations	

We have created a keyword cloud (see Fig. 1) where the most frequently used terms in this book are written in larger letters, thus showing the areas the book focuses on.

Fig. 1 Terms cloud (created with www.wordle.net)

Audience

The audience for this book is software engineering researchers (professors, PhD and postgraduate students, industrial R&D departments, etc.), as well as practitioners (chief information officers, corporate social responsibility professionals, software quality engineers, etc.) who want to know the state of the art as regards green in software engineering.

The reader is assumed to have previous knowledge of software engineering.

Acknowledgements

Ciudad Real, Spain   Coral Calero
May 2014   Mario Piattini

Contents

ix

List of Contributors

Manuel F. Bertoa University of Málaga, Málaga, Spain

Paolo Bozzelli VU University Amsterdam, Amsterdam, The Netherlands

Coral Calero Department of Information Technologies and Systems, University of Castilla-La Mancha, Ciudad Real, Spain

Annick Castiaux University of Namur, Namur, Belgium

Markus Dick Institute for Software Systems, Trier University of Applied Sciences, Trier, Germany

Leticia Duboc Universidade do Estado do Rio de Janeiro, Rio de Janeiro, Brasil

Schahram Dustdar Distributed Systems Group, Vienna University of Technology, Vienna, Austria

Héctor Fernández VU University Amsterdam, Amsterdam, The Netherlands

Qing Gu HU University of Applied Sciences, The Netherlands

Ignacio García-Rodríguez de Guzmán Institute of Information Technologies and Systems, University of Castilla-La Mancha, Ciudad Real, Spain

Eva Kern Institute for Software Systems, Trier University of Applied Sciences, Leuphana University of Lüneburg, Germany

Patricia Lago VU University Amsterdam, Amsterdam, The Netherlands

Fei Li Distributed Systems Group, Vienna University of Technology, Vienna, Austria

Martin Mahaux University of Namur, Namur, Belgium

Simo Mäkinen Department of Computer Science, University of Helsinki, Helsinki, Finland

Mª Ángeles Moraga Department of Information Technologies and Systems, University of Castilla-La Mancha, Ciudad Real, Spain

Stefan Naumann Institute for Software Systems, Trier University of Applied Sciences, Trier, Germany

Birgit Penzenstadler California State University, Long Beach, USA

Ricardo Pérez-Castillo Itestra GmbH, Madrid, Spain

Mario Piattini Department of Information Technologies and Systems, University of Castilla-La Mancha, Ciudad Real, Spain

Macario Polo Department of Information Technologies and Systems, University of Castilla-La Mancha, Ciudad Real, Spain

Giuseppe Procaccianti VU University Amsterdam, Amsterdam, The Netherlands

Soheil Qanbari Distributed Systems Group, Vienna University of Technology, Vienna, Austria

Ankita Raturi University of California, Irvine, Irvine, CA, USA

Debra Richardson University of California, Irvine, Irvine, CA, USA

Juha Taina Faculty of Science, University of Helsinki, Helsinki, Finland

Bill Tomlinson University of California, Irvine, Irvine, CA, USA

Michael Vögler Distributed Systems Group, Vienna University of Technology, Vienna, Austria

Part I
Introduction

Chapter 1
Introduction to Green in Software Engineering

Coral Calero and Mario Piattini

1.1 Introduction

Sustainability is gaining importance worldwide, reinforced by several initiatives with wide media coverage such as the Earth hour[1]; this is a worldwide grassroots movement uniting people to protect the planet, organised by the WWF (World Wide Fund for Nature). Other organisations such as the United Nations (UN) also highlight the importance of reducing energy consumption and our carbon footprint, including this issue in the Millennium Development Goals (MDGs[2]). In Rio+20, the United Nations Conference on Sustainable Development, the world leaders approved an agreement entitled 'The Future We Want', where it is stated that 'We recognize the critical role of technology as well as the importance of promoting innovation, in particular in developing countries. We invite governments, as appropriate, to create enabling frameworks that foster environmentally sound technology, research and development, and innovation, including in support of green economy in the context of sustainable development and poverty eradication...'.

Clean and efficient energy as a societal challenge has also been included by the European Union in Horizon 2020,[3] the biggest EU Research and Innovation programme with nearly €80 billion of funding available from 2014 to 2020. Other initiatives related to environmental sustainability can likewise be found in other countries.

---

[1] http://www.earthhour.org/

[2] http://www.un.org/millenniumgoals/

[3] http://ec.europa.eu/programmes/horizon2020/

C. Calero (✉) • M. Piattini
Department of Information Technologies and Systems, University of Castilla-La Mancha, Ciudad Real, Spain
e-mail: Coral.Calero@uclm.es; Mario.Piattini@uclm.es

Although these initiatives point to ICTs (information and communication technologies) as a key to achieve these goals, we must be aware that ICTs also have a negative impact on the environment. In fact, as noted by [20], when pursuing strategic sustainability, the impact of technology is important from two different points of view at the same time. On the one hand, it helps organisations to tackle environmental issues (using video conferences, dematerialisation, more efficient processes, etc.); on the other hand, technology itself is often responsible for major environmental degradation (amounts of energy consumed by the engineering processes used to manufacture products). This dual aspect of technology means that organisations also face two challenges: they need to have more sustainable processes and they must produce products that contribute to a more sustainable society.

As far as the ICT sector is concerned, it contributes about 2 % of the global CO_2 emissions and is responsible for approximately 8 % of the EU's electricity use, and some 2 % of its carbon emissions come from the ICT equipment and services and household electronic sector. The total electricity consumption of the ICT sector is forecast to increase by almost 60 % from 2007 to 2020 (see Fig. 1.1) due to the increasing number of devices as well as to network expansion [24].

In [38], the authors estimate that present-day systems for business email, productivity and CRM software in the United States require 268, 98 and 7 petajoules (PJ) of primary energy each year, respectively, when the direct energy use and embodied energy of all system components are considered. When combined, the present-day primary energy footprints of these three business software applications add up to as much as 373 PJ per year.
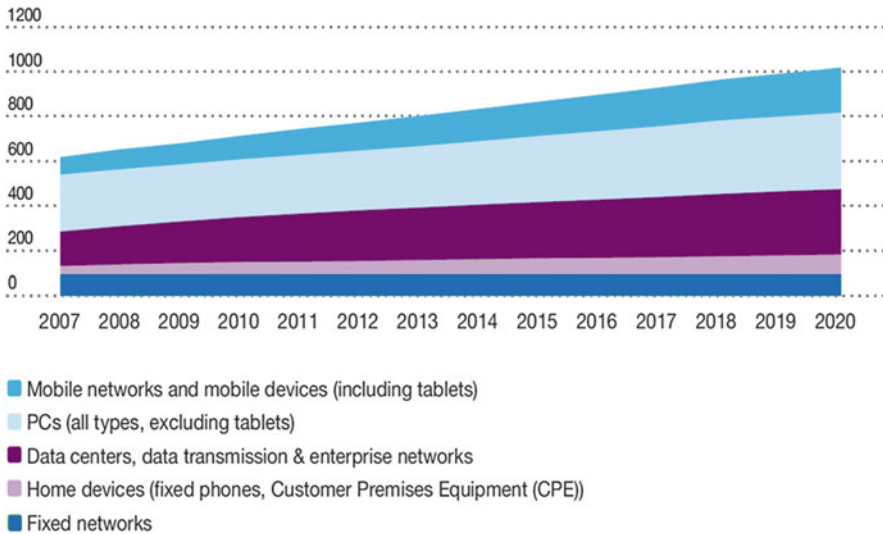

Fig. 1.1 ICT carbon footprint outlook (Mtonnes CO2e) (from [2])

It is therefore essential to control the use of ICTs in order to reduce their impact on sustainability as much as possible. We will focus specifically on software technology, because software is more complex to sell, service and support than hardware, and dollar for dollar, software generates more downstream economic activity than hardware [31], but it has been disregarded in this area until now. Spending on software is growing faster than spending on IT overall—4.8 % a year between 2008 and 2013, compared to 3.3 % for all IT spending [31].

Sustainability has also become more and more important to business recently. A business that fails to have sustainable development as one of its top priorities could receive considerable public criticism and subsequently lose market legitimacy [20]. According to a global IBM survey in 2008, 47 % of organisations have begun to redesign their business models on the basis of sustainability, treating sustainable development as a new source of innovation, a new opportunity for cutting costs and a new mechanism for gaining competitive advantage. All of this can be summarised under the concept of 'strategic sustainability', introduced by [55]. Most people claim that they will pay more for a green product [13]. In early 2010, the ISO 26000 standard [33] for corporate social responsibility (CSR) was published, providing executives with the directions and measures for demonstrating social responsibilities. In this standard, businesses are required to take a precautionary approach to protecting the environment; the aim is to promote greater environmental responsibility through business practices and encourage the adoption of environment-friendly information technologies. CSR involves the voluntary integration by companies of social and environmental concerns in their business operations, as well as in relationships with their partners [26]. Expectations from corporations are higher than ever. Investors and other stakeholders consider companies in terms of the 'triple bottom line', reflecting financial performance, environmental practices and corporate social responsibility (CSR). The present-day dominant conception of CSR implies that firms voluntarily integrate social and environmental concerns in their operations and interactions with stakeholders [12]. All the CSR definitions consistently refer to five dimensions: voluntariness, stakeholders, social, environmental and economic [17].

In general, the initiatives that foster respect for the environment by means of ICT, IT, software, etc., are called Green or Greening ICT/IT/Software or sometimes sustainability in IT. The problem that arises is that, as in every new discipline, there is no clear map of concepts and definitions. As [8] points out, however, the fact is that Green IT is not only a trend; it is becoming a necessity as more and more organisations are implementing some form of sustainable solutions. These same authors comment that, according to Forrester Research, it is expected that the Green IT services market will grow from $500 million to nearly $5 billion in 2013.

In the next section, we will try to clarify the differences, similarities and relationships between all these concepts.

1.2 Sustainability

The aim of this section is to give a general definition of the word 'sustainability' without actually linking it to any particular context. To do so, we will first summarise the main definitions of sustainability.

Sustainability is a widely used term and refers to the capacity of something to last for a long time. Some more precise definitions are as follows:

- The Collins dictionary [16] defines sustainability as 'the ability to be maintained at a steady level without exhausting natural resources or causing severe ecological damage'.
- A similar definition of 'sustainable' can be found in Merriam-Webster: 'of, relating to, or being a method of harvesting or using a resource so that the resource is not depleted or permanently damaged' [39].
- According to [9], a sustainable world is broadly defined as 'one in which humans can survive without jeopardizing the continued survival of future generations of humans in a healthy environment'.
- In [49], the authors affirm that 'sustainability can be discussed with reference to a concrete system (ecological system, a specific software system, etc.), therefore, global sustainability implies the capacity for endurance given the functioning of all these systems in concert'.
- 'Sustainability is the capacity to endure and, for humans, the potential for long-term maintenance' [47].
- From another perspective, sustainability can be viewed as 'one more central quality attribute in a row with the standard quality attributes of correctness, efficiency, and so forth' [47]. These same authors also defined the term *sustainable development* as that which 'includes the aspect to develop a sustainable product, as well as the aspect to develop a product using a sustainable development process'.
- The Brundtland report from the United Nations (UN) defines sustainable development as the ability to 'meet the needs of the present without compromising the ability of future generations to satisfy their own needs' [62]. According to the UN, sustainable development needs to satisfy the requirements of three dimensions, which are the society, the economy and the environment.
- In [2], the author identifies the same dimensions as the aforementioned UN report for sustainable development: economic development, social development and environmental protection:

  - 'Environmental sustainability ensures that the environment is able to replenish itself at a faster rate than it is destroyed by human actions. For instance, the use of recycled material for IT hardware production helps to conserve natural resources.
  - Social development is concerned about creating a sustainable society which includes social justice or reducing poverty. In general all actions that promote social equity and ethical consumerism.

– The economic pillar ensures that our economic growth maintains a healthy balance with our ecosystem; it integrates environmental and social concerns into business'.

Of all the definitions above, the most widely used is that established by the Brundtland report of the United Nations (UN) [62].

If we take a close look at the definitions, we can observe that there are two fundamental pillars underpinning sustainability: 'The capacity of something to last a long time' and 'the resources used'.

Another aspect that is related to sustainability, and that can be found in the literature, has to do with the topic to which it is applied: information systems, ICT, software, etc.

Taking into account that our focus is on software engineering (SE), Fig. 1.2 summarises the different levels of sustainability that relate organization to information systems and to software engineering.

In the following sections, we will present some definitions for each of the levels in Fig. 1.2. We have worked mainly with papers published in the area of software, software engineering and information systems, because that is what this book focuses on. This means that we will not present an exhaustive study on definitions (i.e. on those beyond the scope of this book), but we believe our work will provide a snapshot of how things are interpreted in the software engineering area.
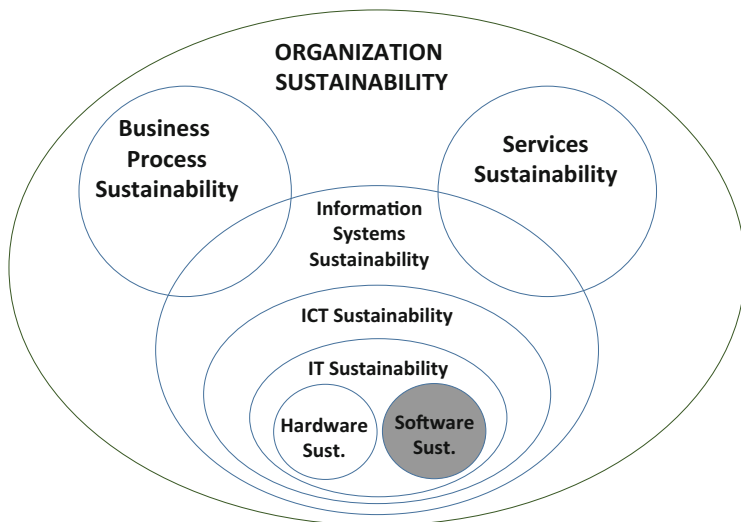
Fig. 1.2 Sustainability levels

1.2.1 IS Sustainability

It must be noted that, in the literature, authors do not differentiate between 'IS sustainability' and 'sustainable IS' (the same applies to the other levels), so we take these concepts as equivalent in this book.

As articulated in the SIGGreen Statement, 'the Information Systems discipline can have a central role in creating an ecologically sustainable society because of the field's five decades of experience in designing, building, deploying, evaluating, managing, and studying information systems to resolve complex problems' [27].

The authors of [64] recommend using the term IS sustainability over IT sustainability, because they consider that the exclusive focus on information technologies is too narrow.

As remarked by [53], it is only through process change and the application of process-centred techniques, such as process analysis, process performance measurement and process improvement, that the transformative power of IS can be fully leveraged in order to create environmentally sustainable organisations and, in turn, an environmentally sustainable society.

Taking this one step further, we contend that IS researchers must consider process-related concepts when theorising about the role of IT in the transformation towards sustainable organisations. This will not only allow us to better understand the transformative power of IS in the context of sustainable development but will also enable us to proceed to more prescriptive, normative research that has a direct impact on the implementation of sustainable, IT-enabled business processes [53].

Although there are some groups working on information systems and environmental friendliness, it is difficult to find suitable IS sustainability concepts. Most of the work being done is about Green IS. In [13], it is considered that the sustainability in IS must take into account aspects such as efficiency systems, forecasting, reporting and awareness, energy-efficient home computing and behaviour modification. Finally, the book focuses on Green Business Process Management consolidating the global state-of-the-art knowledge about how business processes can be managed and improved in the light of sustainability objectives [63].

1.2.2 ICT/IT Sustainability

Donnellan et al. [4] remark that sustainable ICT can develop solutions that offer benefits both internally and across the enterprise:

- Aligning all ICT processes and practices with the core principles of sustainability, which are to reduce, reuse and recycle
- Finding innovative ways to use ICT in business processes to deliver sustainability benefits across the enterprise and beyond

The Ericsson report [24] points to dematerialisation and increased efficiency as the two main ways of aligning ICT with sustainability.

Following the definition provided by [61], IT sustainability is a shorthand for 'global environmental sustainability', a characteristic of the Earth's future, in which certain essential processes persist for a period of time comparable with human lives.

1.2.3 Software Sustainability

There are several areas in which software sustainability needs to be applied: software systems, software products, Web applications, data centres, etc. Various works are in process, but most of this concerns data centres, which consume significantly higher energy than commercial office space [36].

As noted in [10], the way to achieve sustainable software is principally by improving power consumption. Whereas hardware has been constantly improved so as to be energy efficient, software has not. The software development life cycle and related development tools and methodologies rarely, if ever, consider energy efficiency as an objective [11]. Energy efficiency has never been a key requirement in the development of software-intensive technologies, and so there is a very large potential for improving efficiency [59].

As remarked by [21], software plays a major role, both as part of the problem and as part of the solution. The behaviour of the software has significant influence on whether the energy-saving features built into the platform are effective [56].

In [49], it is said that 'The term Sustainable Software can be interpreted in two ways: (20) the software code being sustainable, agnostic of purpose, or (24) the software purpose being to support sustainability goals. Therefore, in our context, sustainable software is energy-efficient, minimizes the environmental impact of the processes it supports, and has a positive impact on social and/or economic sustainability. These impacts can occur direct (energy), indirect (mitigated by service) or as rebound effect [30]'.

According to [18], sustainable software is 'software, whose impacts on economy, society, human beings, and environment that result from development, deployment, and usage of the software are minimal and/or which have a positive effect on sustainable development'.

These authors subsequently use the same definition for the concept of green and sustainable software. They therefore define green and sustainable software as 'software, whose direct and indirect negative impacts on economy, society, human beings, and environment that result from development, deployment, and usage of the software are minimal and/or which has a positive effect on sustainable development' [46]. They consider that direct impacts are related to resources and energy consumption during the production and use of software, while indirect impacts are effects from the software product usage, together with other processes and long-term systemic effects.

One of the most complete definitions is the one proposed by [34], which considers that green and sustainable software is software whose:

- 'Direct and indirect consumption of natural resources, which arise out of deployment and utilization, are monitored, continuously measured, evaluated and optimized already in the development process
- Appropriation and utilization aftermath can be continuously evaluated and optimized
- Development and production processes cyclically evaluate and minimize their direct and indirect consumption of natural resources and energy'

Another related term is sustainable computing. It is used to transfer the political concept of sustainability to computer systems, including material components (hardware) as well as informational ones (software); it includes development as well as consumption processes [40].

As commented at the beginning of Sect. 1.2, the literature contains some definitions of sustainable (or sustainability), while others refer to the term *green* (or *greenability*).

This phenomenon is especially noteworthy in the case of software, because various authors such as [46] and [34] use both terms synonymously. We believe that this approach is faulty and that it ought to be avoided, since we are talking about two different concepts, as will be seen in due course.

What does seem true, however, is that software sustainability, although still in its early stages, is a very important research topic that will be of great importance in the next few years. That said, general work on its significance is needed. The goal of that work would be to raise awareness on the part of all those involved with software: the companies that develop software, those who buy it and also the people who use it.

1.2.3.1 Software Engineering Sustainability

One part of the software sustainability is the software engineering sustainability. Within the context of software engineering, not many proposals have tackled the concept of sustainability [50]. In a recent update of this work, the authors observed that the number of proposals has increased considerably over the last 2 years [49]. This fact serves to demonstrate that there is an ever-growing concern to tackle sustainability in the context of software engineering.

Sustainability should generally be taken into account from the very first stages of software development. That is not always feasible, since it is not easy to change how developers work. Moreover, there is little guidance on how software engineering can contribute to improving the sustainability of the systems under development [48]. In this work, the authors consider five dimensions of sustainability that are important for the analysis of software systems:

- Individual sustainability: This refers to the maintenance of the private good of individual human capital. Health, education, skills, knowledge, leadership and access to services constitute human capital [52]. For software engineering (SE),

we have to ask: 'How can software be created and maintained in a way that enables developers to be satisfied with their job over a long period of time?'

- Social sustainability: This means maintaining social capital and preserving the solidarity of societal communities. Social capital is investments and services that create the basic framework for society [52]. For SE, we ask: 'What effects do software systems have on society (e.g. communication, interaction, government)?'
- Economic sustainability: This aims to maintain assets. Assets include not only capital but also added value. This requires a definition of income as the "amount one can consume during a period and still be as well off at the end of the period, as it devolves on consuming added value (interest), rather than capital" [52]. For SE, the question is: 'How can software systems be created so that the stakeholders' long-term investments are as safe as possible from economic risks?'
- Environmental sustainability: This seeks to improve human welfare by protecting natural resources such as water, land, air, minerals and ecosystem services; hence, much is converted to manufactured or economic capital. Environment includes the sources of raw materials used for human needs, as well as ensuring that sink capacities recycling human wastes are not exceeded [39]. For SE, we pose the question: 'How does software affect the environment during, inter alia, development and maintenance?'
- Technical sustainability: From a point of view of (software) systems engineering, there is another dimension that has to be considered. Technical sustainability has the central objective of long-time usage of systems and their adequate evolution with changing surrounding conditions and respective requirements. For SE: How can software be created so that it can easily adapt to future change?

There are many definitions of sustainable software engineering. We present some of these in Table 1.1. It is clear that there are many more works which use the term *sustainable software engineering*.

1.3   From Sustainability to Greenability

As detected in several definitions, sustainability is generally considered from three dimensions (the social, the economic and the environmental) provided by the UN [62].

If we apply the definition to our context, the third dimension, the one related to the technical aspects, is the one that we call the 'green' dimension. Figure 1.3 shows this in diagram form.

Taking this distinction as a basis, in the next section we will show the definitions of green applied to each one of the levels in Fig. 1.2. As happened in the case of sustainability, in the literature authors use the terms *green* and *greenability*

Table 1.1 Sustainable

Reference	Term	Definition
[3]	Sustainable software engineering	Sustainable software engineering aims to create reliable, long-lasting software that meets the needs of users while reducing environmental impacts; its goal is to create better software so we will not have to compromise future generations' opportunities
[37]	Sustainable software engineering	Sustainable software engineering aims to create reliable, long-lasting software that meets the needs of users while reducing the negative impact on the economy, society and the environment
[33]	Sustainable software engineering	Sustainable software engineering is the art of defining and developing software products in a way so that the negative and positive impacts on sustainability that result and/or are expected to result from the software product over its whole life cycle are continuously assessed, documented and optimised
[58]	Sustainable software engineering	Sustainable software engineering is the development that balances rapid releases and long-term sustainability, whereas sustainability is meant as the ability to react rapidly to any change in the business or technical environment
[19]	Green and sustainable software engineering	Green and sustainable software engineering is the art of developing green and sustainable software with a green and sustainable software engineering process. Therefore, it is the art of defining and developing software products in a way, so that the negative and positive impacts on sustainable development that result and/or are expected to result from the software product over its whole life cycle are continuously assessed, documented and used for a further optimisation of the software product
[35]	Green and sustainable software engineering	The objective of green and sustainable software engineering is the enhancement of software engineering, which targets 1. The direct and indirect consumption of natural resources and energy 2. As well as the aftermath that are caused by software systems during their entire life cycle, the goal being to monitor, continuously measure, evaluate and optimise these facts
[31]	Software engineering for sustainability	The aim of software engineering for sustainability (SE4S) is to make use of methods and tools in order to achieve this notion of sustainable software

(e.g. Green IS and IS Greenability) synonymously; we will do the same, presenting definitions found for both concepts.

Fig. 1.3 Sustainability dimensions

1.3.1   Green IS

At the top level, we found the Green IS concept. Chen et al. [14] unite the terms *Green IT* and *Green IS* and suggest that 'Green IS & IT refers to IS & IT products (e.g., software that manages an organization's overall emissions) and practices (e.g., disposal of IT equipment in an environmentally-friendly way) that aims to achieve pollution prevention, product stewardship, or sustainable development'.

The authors of [64] define Green IS as inclusive of Green IT, extended with people, processes, software and information technologies to support individual, organisational or societal goals (Fig. 1.4).

The Green Book [29] focuses on Green IS projects, programmes and initiatives as potential influences on the sustainability of organisations and communities under threat from climate change and other aspects of environmental degradation.

1.3.2   Green ICT/IT

The literature provides us with a variety of definitions of the concept of Green IT. The term *Green IT* refers to the relationship between IT and energy efficiency [11].

In [6], the authors state that Green IT means using technology efficiently while taking into account the triple bottom line: 'economic viability, social responsibility and environmental impact'.

Fig. 1.4 Green IS and
Green IT
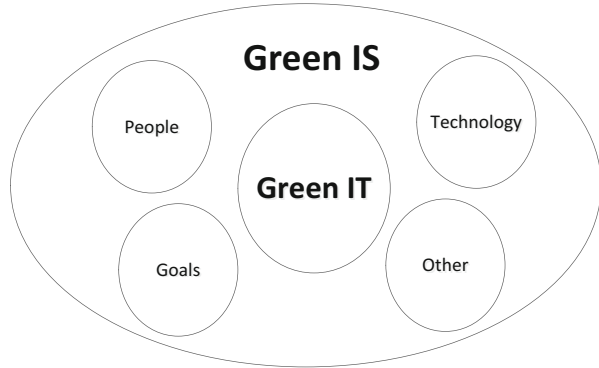
The same author in [5] presents eco-computing and green computing as synonyms of Green IT, defining them as a set of best practices for the optimal use of computing resources. Green practices in technology can cover several phases of the product or service life cycle: from acquisition to recycling and final disposal.

In [45], the author considers that Green IT 'refers to the study and practice of designing, manufacturing, and using computer hardware, software, and communication systems efficiently and effectively with no or minimal impact on the environment'. In his opinion, Green IT is also about 'using IT to support, assist, and leverage other environmental initiatives and to help in creating green awareness'. The author refined the definition in [44] in the following manner: 'Green IT is the study and practice of designing, manufacturing, and using computers, servers, monitors, printers, storage devices, and networking and communications systems efficiently and effectively with minimal impact on the environment. It includes environmental sustainability, the economics of energy efficiency, and the total cost of ownership, which incorporates the cost of disposal and recycling. Green IT is also about the application of IT to create energy-efficient, environmentally sustainable business processes and practices'.

Very similar definitions are provided in the following pieces of work:

- In [22], 'the aim of Green IT is to produce as little waste as possible during the whole IT lifecycle (development, operation and disposal)'.
- In [35], 'Green IT considers the optimizing the resource and energy consumption of ICT itself, induced during the whole life cycle, and tries to optimize it'.
- In [28], Green IT 'denotes all activities and efforts incorporating ecologically friendly technologies and processes into the entire life cycle of information and communication technology'.
- In [15], the authors use the term *green computing*. This term 'refers to environmentally sustainable computing which studies and practices virtually all computing efficiently and effectively with little or no impact on the environment'. The green computing term is the same as Green IT [4].
- In [42], 'Green IT is a systematic application of environmental sustainability criteria to the design, production, sourcing, use and disposal of the IT technical

infrastructure as well as within the human and managerial components of the IT infrastructure in order to reduce IT, business process and supply chain related emissions and waste and improve energy efficiency'.

- In [7], the authors consider Green IT and green computing as synonymous, defined as the study and practice of designing, manufacturing, using and disposing of computers, servers and associated subsystems efficiently and effectively with minimal or no impact on the environment. Green IT thus encompasses hardware assets, software assets, tools, strategies and practices that help improve and foster environmental sustainability.

- In [54], the authors consider that the definition of Green IT is broad, as it can be applied to situations where IT enables greenhouse gas emission reductions and to situations where IT enables structural changes that lead to changes in broader societal patterns, which takes us closer to the low-carbon society and leads to further emission reductions.

A different definition is provided in [1]. In this work, the author believes that Green IT can be described by dividing IT-related issues into four different fields:

- 'Field 1 concerns the IT product itself and the energy and environmental impact they cause and in particular the products people use on a daily basis. This field is important to gain (sic) credibility to Green IT solutions because it is difficult to take Green IT seriously if the products needed have not been undertaken by the process of diminishing environmental impact.
- Field 2 is about transportation, communication and virtual mobility. This field is pictured as two separate parts, which represents the transportation of goods and the transportation of people.
- Field 3 is about community planning on all levels, ranging from whole regions, cities and small towns down to the personal household planning level.
- Field 4 handles the production and consumption patterns. IT opens possibilities to measure environmental impact on production and consumption and following a product or a service throughout its entire lifecycle enables control over the total environmental effect'.

In [41], the authors explain that Green IT initiatives can range from those that focus on reducing IT infrastructure's carbon footprint to those that transform a business. Green IT can be deployed to support a variety of sustainability initiatives, such as those to measure carbon footprints, monitor the environmental impact of business practices, reduce waste in business processes, lower resource consumption or increase energy efficiency and reduce greenhouse gas emissions.

From our point of view, one of the definitions that best expresses how the term *Green IT* is tackled in the literature, at the same time as being more thorough and precise, is the definition provided in [48], which encapsulates all the definitions in [15, 22, 28, 35].

For a deeper insight into Green IT, we recommend the book *Harnessing Green IT* [43], in which the idea is to give a holistic perspective on Green IT by discussing its various facets and showing how to embrace them strategically.

As remarked by [22], however, over a long time, the topics of Green IT involved only research dealing with hardware. It is clear that, independently of the efforts made until now, software is also part of IT and must be taken into account when talking about Green IT.

Apart from the previous definitions of general Green IT, there is an important aspect to be taken into account, which is related to the difference between Green *in* IT and Green *by* IT. The next section introduces this difference, along with the definitions found between them.

1.3.2.1   Green by IT Versus Green in IT

The main difference between Green in IT and Green by IT is the role played by the IT and the focus of the greenness. As indicated by [61], the difference depends on considering IT as a producer to handle the emissions produced by the IT gadgets themselves or considering IT as an enabler to enable reduction of emissions across all areas of an enterprise. This difference was also highlighted recently in [22], where it is stated that IT can contribute to eco-sustainability in two ways: on the one hand, Green IT (*Green in IT*), when IT itself has an impact on the environment, and on the other hand Green by IT, when IT provides tools for making tasks environment friendly.

This means that when the goal pursued is to reduce the energy consumption and the resources used by IT, we are talking about Green in IT. When the focus is on using IT to achieve more environment-friendly systems in other domains, then this is Green by IT. This same idea is set out in [35], where it is stated that IT can contribute to sustainability from two perspectives. On the one hand, IT can support sustainability by optimising the resources and energy consumption of ICT itself, as induced during the whole life cycle. This concept is called Green IT (Green in IT). On the other hand, IT can support sustainability by providing ICT solutions that reduce the environmental impact in other fields [23]. This is the concept known as Green by IT.

As we know, IT is composed mainly of software and hardware; this means that the same considerations can be applied at these levels; thus, we can have green in software, green in hardware, green by software and green by hardware (Fig. 1.5).

Finally, we can combine the BY and the IN aspects in software and in hardware. We have called this green software and green hardware, respectively, which together make up Green IT. In Fig. 1.5, these relationships are shown in the form of a diagram.

We will use these concepts when presenting the different definitions found because, as shown previously, there are discrepancies between the concepts and the meanings given by the different authors.
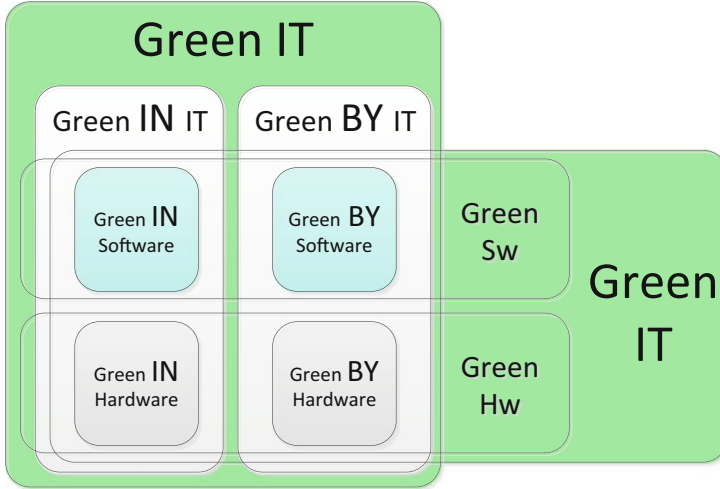
Fig. 1.5 Green software, green hardware and Green IT

We will therefore use the terminology in Fig. 1.5 to unify the different definitions. Readers will find in italic and parentheses the alternative concept to that used by the authors.

In [6], the authors talk about Green IT 1.0 and Green IT 2.0, defining both as follows: Green IT projects can be divided into two categories:

1. Green for IT (*Green in IT*), projects aiming to reduce the environmental impact of IT, also known as Green IT 1.0. For example, 10 GbE (10 gigabit per second Ethernet), clean energy to power data centres, hardware virtualisation, cloud computing services (i.e. software as a service (SaaS), Web services, infrastructure as a service, developing platform as a service), data centre outsourcing and co-location services, IT asset disposal and recycling services, IT energy measurement, localised cooling, managed printing services, PC power management software, storage capacity optimisation, thin clients (i.e. low-cost terminals limited to user interface (UI) processing, data processing being run on the server)
2. IT for Green (*Green by IT*), projects aiming to reduce the environmental impact of operations using IT, also known as Green IT 2.0. For example, process automation, remote collaboration, TelePresence, and resource usage management (energy, water, paper, CO_2), for example, Project 2 degrees

The same names of Green 1.0 and 2.0 are used by [44] but in another sense. These authors argue that we are now marching towards the second wave of Green IT. The first wave, Green IT 1.0, was internally focused on re-engineering IT products and processes to improve their energy efficiency and meet compliance requirements. Green IT 2.0 is externally focused on business transformation, sustainability-based IT innovations and enterprise-wide sustainability [44].

The definition given in [25] highlights the fact that there are two concepts used, depending on the nature of IT: Green IT (*Green in IT*), defined as the IT sector's own activity and its impact on environmental efficiency, and green applications of

IT (*Green by IT*) or IT for Green, defined as the impact of IT on the environmental productivity of other sectors, particularly in terms of energy efficiency and their carbon footprint.

A slightly different definition of the concepts is the one given by [42], where waste (materials or substances which harm the environment or demand surplus energy or resources) is used as a definition criterion: Green IT (*Green in IT*) is to produce as little waste as possible during the whole IT life cycle (development, operation and disposal), and Green by IT aims at producing as little waste as possible by means of IT.

In [51], the author talks about sustainability for software engineering (how to make SE itself more sustainable) and sustainability in software engineering (how to improve the sustainability of the systems we develop). Although the author talks about sustainability, we think she refers to what we call green software engineering and, more concretely, to green in software engineering (from the point of view of the process and of the product).

As can be observed, most of the authors use Green IT and Green by IT instead of Green in IT and Green by IT. We maintain that this is confusing because conceptually Green in and by IT are part of Green IT. That is why we have decided to use Green in IT instead of only Green IT, giving Green IT an upper level that contains Green in IT and Green by IT.

1.3.3   Green Software

As remarked in the Intel technical article *Impact of Software on Energy Consumption*, much of the computer energy used (and saved) is based on the effectiveness of hardware energy efficiency and the hardware power states of the computer. But software has an impact as well, in two ways: while running a 'workload' and while 'idle'.[4]

Until recently, the greater part of the work done within the Green IT industry was related to the area of hardware, focusing mainly on improving the energy efficiency of hardware.

Hardware is of course fundamental, but hardware and software together form a whole; one has no meaning without the other. It thus seems self-evident that research work needs to be broadened to include software. As [22] points out, researchers have to pay attention to the effect of software within Green IT.

The trend has been changing in the last few years, and new pieces of work related to the area of green software are emerging. However, there is no common definition of green software [1], a fact that leads us to outline some of the definitions that can be found for the term *green software*.

---

[4] https://noggin.intel.com/content/impact-of-software-on-energy-consumption

Murugesan and Gangadharan [43] define green software as environment-friendly software that helps improve the environment. The authors classify green software into four categories:

- Software that is greener (consumes less energy to run)
- Embedded software that assists other things in going green (smart operations)
- Sustainability-reporting software (or carbon management software)
- Software for understanding climate change, assessing its implications and forming suitable policy responses

Green software is defined in [57] as software that must fulfil three high-level requirements:

1. The required software engineering processes of software development, maintenance and disposal must save resources and reduce waste.
2. Software execution must save resources and reduce waste.
3. Software must support sustainable development.

According to [22], green software is 'an application that produces as little waste as possible during its development and operation'.

1.3.3.1 Green by Software Versus Green in Software

As happened with Green IT, green software can be divided into green by software and green in software. Again, the main difference is whether the goal pursued is to have more environment-friendly software or if it is rather to produce software that helps the environment. Figure 1.6 shows this in diagram form.
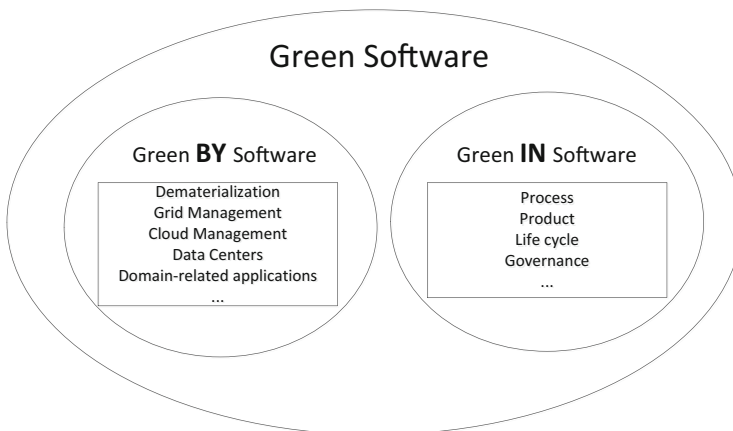
Fig. 1.6 'Green by' and 'Green in' software

Green by software appeared some time ago. In general, green by software covers software developed for domains that work in the preservation of the environment, as well as software that helps to manage energy-intensive applications.

On the other hand, green in software is related to how to make software in a more sustainable way resulting in a more sustainable product (this is called green software engineering). The next section will discuss this.

Of course, green in software also includes other aspects aside from software development, such as governance.

1.4   Green in Software Engineering

Green in software engineering is part of green in software and therefore of green software; green in software engineering is the focus of this book. Its main goal is to include green practices as part of the software development process, as well as the rest of activities that are part of software engineering (see Fig. 1.7).

ISO/IEC/IEEE Systems and Software Engineering Vocabulary (SEVOCAB) defines software engineering as 'the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software' [32].

Based on this definition, we can define green in software engineering as those practices which apply engineering principles to software by taking into consideration environmental aspects. The development, the operation and the maintenance of software are therefore carried out in a green manner and produce a green software product (Fig. 1.8).
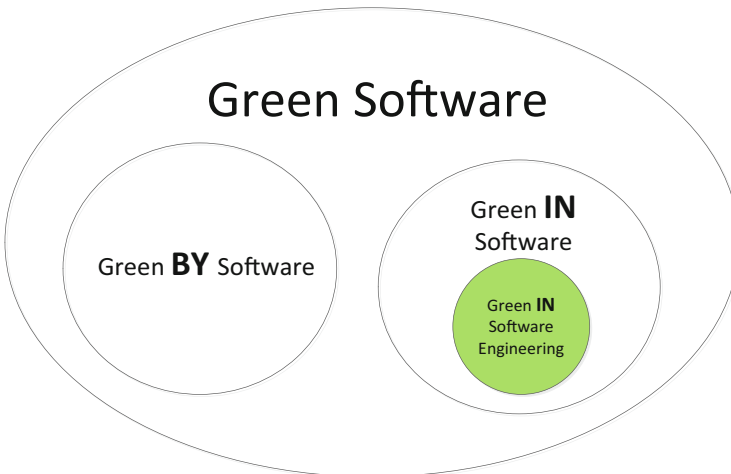
Fig. 1.7   Green in software engineering
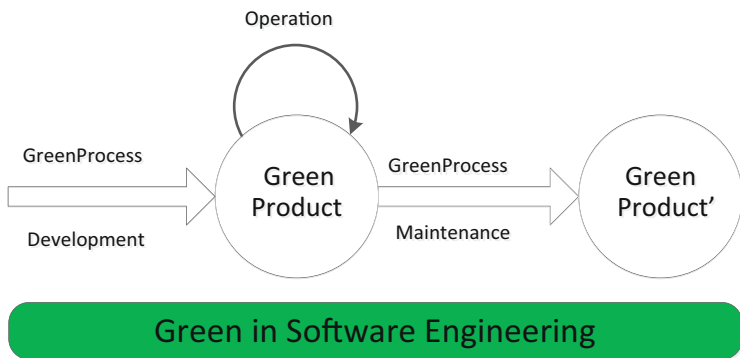
Fig. 1.8 Green in software engineering

In [22], the author explains that software engineering can be green in three ways [60]: (a) by producing green software, (b) by producing software to support environmental consciousness (green by software) and (c) by producing less waste during the development process.

As can be observed, the author mixes green by software with green in software in defining green software engineering. Taking into account the definition of software engineering, it seems that ways (a) and (c) fit, but (b) is does not.

This book gives information on the efforts that are being made nowadays in the arena of green in software engineering. The following chapters will provide information about the different areas of SWEBOK [65], attempting to give a complete snapshot of the present state of the art [32].

1.5 Other Green Concepts in This Book

In this section, we provide a list of the definitions related to green software given in the different chapters of this book but not yet dealt with by us. To that end, Table 1.2 shows the chapter in which the term appears along with the term itself and its definition. It should be underlined that the definition is not necessarily proposed by the authors themselves; it may have been taken from the literature but used in the authors' work. This means that it is advisable to look up the chapter that contains the definition if the reader wishes to find out the exact source of a particular term. It should also be noted that the chapters in this book include definitions of concepts already treated in this chapter that we have not taken into consideration to avoid unnecessary repetitions.

Table 1.2 Green definitions

Chapter number	Term	Definition
2	Software engineering environment (SEE)	It describes the network of people, software, hardware and infrastructure involved in the construction of software
	Design for sustainable behaviour	It denotes how designers can influence users to act in a more environment-friendly manner with respect to their use of products, services and environments
3	Green and sustainable software product	It is a product that should have little impact on the sustainable development and, if it is its specific benefit, promote the pursuit of sustainable objectives
4	Green software services (GSS)	When green software is delivered as online services
5	Green strategy	It is a plan of action intended to accomplish a specific environmental goal
	Green goal	It is an objective that an organisation sets itself to achieve and which is quantified where practical
6	Software engineering for sustainable development	Addresses issues and questions of where and how software and software engineering can help sustainable development
	Green quality	It is only indicated by some quality indicators, either indirectly or directly
	Sustainability of software systems	Systems which generate much waste can be considered more harmful to the environment than those systems that are better at recycling
	Green quality factors	They are factors that define how software supports sustainable development
	Resource efficiency (in software engineering)	It is related to software life cycle, including software design, management, maintenance and disposal
	Software execution resource efficiency	It is related to software execution and software platform usage
	Software client process resource efficiency	It is related to how software stakeholders benefit from software and its software system
	Triftness	It is a factor that evaluates how software reduces waste
	Social sustainability	It is a factor that evaluates how software supports social equality
7	Sustainable	Capable of being upheld; maintainable; and to sustain as 'to keep a person, community, etc., from failing or giving way'; to keep in being; to maintain at the proper level; to support life in; nature, etc., with needs
	Requirements engineering for sustainability	It denotes the concept of using requirements engineering and sustainable development techniques to improve the environmental, social and economic

Table 1.2 (continued)

Chapter number	Term	Definition
		sustainability of software systems and their direct and indirect effects on the surrounding business and operational context
	Green requirements engineering	It denotes the same concept as requirements engineering for sustainability with a specific focus on the direct and indirect environmental impacts of systems
	Environmental requirements	Requirements with regard to resource flow, including waste management, can be elicited and analysed by life cycle analysis (LCA)
9	Green software maintenance	It is performed during the entire software working cycle and ends with the retirement of the software product, undertaking at this point all the required activities to reduce the environmental impact of the retired software. It includes modification of code and documentation in order to solve possible deviation of the greenability requirements (or the implementation of new ones), without modifying the original functionality of the source code
	Ecological debt	The cost (in terms of resource usage) of delivering a software system with a greenability degree under the level of the non-functional requirements established by stakeholders, plus the incurring cost required to refactor the system in the future
10	Sustainable software development	It refers to a mode of software development in which resource use aims to meet (product) software needs while ensuring the sustainability of natural systems and the environment
	Greenability	Degree to which a product lasts over time, optimising the parameters, the amounts of energy and the resources used
	Energy efficiency	Degree of efficiency with which a software product consumes energy when performing its functions
	Resource optimisation	Degree to which the resources expended by a software product, when performing its functions, are used in an optimal manner
	Capacity optimisation	Degree to which the maximum limits of a product or system parameter meet requirements in an optimal manner, allocating only those which are necessary
	Perdurability	Degree to which a software product can be used over a long period, being, therefore, easy to modify, adapt and reuse
	Greenability (in use)	Degree to which a software product can be used by optimising its efficiency, by minimising environmental effects and by improving the environmental user perception

Table 1.2 (continued)

Chapter number	Term	Definition
	Efficiency optimisation	Optimisation of resources expended in relation to the accuracy and completeness with which users achieve goals. Relevant resources can include time consumption, software resources, etc.
	User's environmental perception	Degree to which users are satisfied with their perception of the consequences that the use of software will have on the environment
	Minimisation of environmental effects	Degree to which a product or system reduces the effects on the environment in the intended contexts of use
	Quality in use	It is the degree to which a product or system can be used by specific users to meet their needs in order to achieve specific goals with efficiency, freedom from risk, greenability and satisfaction in specific contexts of use

1.6 Challenges and Future Work

Several efforts trying to highlight the importance of including green aspects within software engineering have been undertaken in recent years.

Our task is to raise awareness among software developers (software industries, development departments, etc.) as well as users, who hold in their hands the responsibility of choosing and demanding a software that is more respectful of the environment.

If we achieve this, the whole software development ecosystem will be forced to adopt greener software processes and produce greener software products if they want to remain competitive.

As the issue of green software develops and strengthens, the terminology used will also become clearer. In this chapter, we have attempted to gather the main terms used today. We are certain that these are subject to modification, evolving as the area itself grows in maturity and thereby solving some of the currently present inconsistencies and lack of precision.

Green in software engineering is a nascent research area, so there are plenty of challenges. It is our firm belief that in the next few years we will see research findings and practical applications that we could never even imagine at the present time.

References

1. Abenius S (2009) Green IT & Green software – time and energy savings using existing tools. In: Environmental informatics and industrial environmental protection: concepts, methods and tools. Shaker Verlag, Aachen, pp 57–66
2. Adams WM (2006) The future of sustainability. Re-thinking environment and development in the twenty-first century: technical report, IUCN
3. Amsel N, Ibrahim Z, Malik A, Tomlinson B (2011) Toward sustainable software engineering: NIER track. In: 2011 33rd international conference on software engineering (ICSE), pp 976–979
4. Donnellan B, Sheridan C, Curry E (2011) A capability maturity framework for sustainable information and communication technology. IT Prof 13(1):33–40
5. Bachour N (2012) Green IT project management, Chapter 7. In: Hu W, Kaabouch N (eds) Sustainable ICTs and management systems for green computing. IGI, Hershey, PA. ISBN 978-1-4666-1839-8
6. Bachour N, Chasteen L (2010) Optimizing the value of Green IT projects within organizations. In: Green technologies conference
7. Bose R, Luo X (2012) Green IT adoption: a process management approach. Int J Account Inform Manag 20(1):63–67
8. Brodkin J (2008) Economy driving Green IT initiatives. Netw World 25(48):16
9. Brown B, Hanson M, Liverman D, Merideth R (1987) Global sustainability: toward definition. Environ Manag 11(6):713–719
10. Calero C, Bertoa MF, Moraga MA (2013) A systematic literature review for software sustainability measures. In: GREENS 2013: Second international workshop on green and sustainable software, pp 46–53
11. Capra E, Francalanci C, Slaughter SA (2012) Is software "green"? Application development environments and energy efficiency in open source applications. Inform Software Tech 54 (1):60–71
12. Castelo B, Lima M (2006) Corporate social responsibility and resource-based perspectives. J Bus Ethics 69:111–132. doi:10.1007/s10551-006-9071-z, Springer 2006
13. Cazier JA, Hopkins BE (2011) Doing the right thing for the environment just got easier with a little help from information systems. In: Proceedings > Proceedings of SIGGreen workshop. Sprouts: working papers on information systems, vol 11, issue 10. http://sprouts.aisnet.org/11-10
14. Chen AJ, Watson RT, Boudreau MC, Karahanna E (2009) Organizational adoption of green IS & IT: An institutional perspective. In: ICIS 2009 proceedings, 142. Retrieved from http://aisel.aisnet.org/icis2009/142
15. Chia-Tien Dan L, Kai Q (2010) Green computing methodology for next generation computing scientists. In: Computer Software and Applications Conference (COMPSAC), 2010 IEEE 34th Annual, pp 250–251
16. Collins (2013) Collins dictionary. http://www.collinsdictionary.com/
17. Dahlsrud A (2008) How corporate social responsibility is defined: an analysis of 37 definitions. Corp Soc Responsibility Environ Manag 15:1–13. doi:10.1002/csr.132, Wiley InterScience
18. Dick M, Drangmeister J, Kern E, Naumann S (2013) Green software engineering with agile methods in green and sustainable software (GREENS). In: 2013 2nd International Workshop, pp 78–85
19. Dick M, Naumann S (2010) Enhancing software engineering processes towards sustainable software product design. In: Greve K, Cremers AB (eds) EnviroInfo 2010: Integration of environmental information in Europe. Proceedings of the 24th International Conference EnviroInfo, Cologne/Bonn, Germany. Shaker, Aachen, pp 706–715
20. Du W, Pan SL, Zuo M (2013) How to balance sustainability and profitability in technology organizations: an ambidextrous perspective. IEEE Trans Eng Manag 60(2):366–385. doi:10.1109/TEM.2012.2206113

21. Easterbrook S (2010) Climate change: a grand software challenge. In: FoSER 2010, November 7–8, Santa Fe, New Mexico, USA, ACM 978-1-4503-0427-6/10/11, pp 99–103
22. Erdelyi K (2013) Special factors of development of green software supporting eco sustainability. In: IEEE 11th international symposium on intelligent systems and informatics (SISY), pp 337–340
23. Erdmann L, Hilty J, Goodman P (2004) Arnfalk the future impact of ICTs on environmental sustainability in technical report EUR 21384 EN. Technical report series EUR 21384 EN. European Commission; Joint Research Centre; IPTS – Institute for Prospective Technological Studies, Seville
24. Ericsson (2013) Ericsson energy, carbon report. On the impact of the networked society. EAB-13:036469 Uen. Ericsson AB. http://www.ericsson.com/res/docs/2013/ericsson-energy-and-carbon-report.pdf. Accessed on April 2014
25. Faucheux S, Nicolaï I (2011) IT for Green and Green IT: a proposed typology of eco-innovation. Ecol Econ 70(11):2020–2027
26. Green Book (2000) European Commission, March 2000
27. Hasan H, Molla A, Cooper V (2012) Towards a green IS taxonomy. In: Proceedings of SIGGreen workshop. Sprouts: Working papers on information systems, vol 12, issue 25, http://sprouts.aisnet.org/12-25
28. Hedwig M, Malkowski S, Neumann D (2009) Taming energy costs of large enterprise systems through adaptive provisioning. In: International conference on information systems (ICIS 2009), paper 140. Retrieved from http://aisel.aisnet.org/icis2009/140
29. Helen H (2010) Taking the green IS message to the world. In: SIGGreen proceedings, pp 139–142. http://siggreen-icis2010-wokshop.wikispaces.com/file/view/SIGGreenICIS2010-WorkshopEBook.pdf/189649215/SIGGreenICIS2010WorkshopEBook.pdf
30. Hilty LM, Arnfalk P, Erdmann L, Goodman J, Lehmann M, Wäger PA (2006) The relevance of information and communication technologies for environmental sustainability – a prospective simulation study. Environ Model Software 21(11):1618–1629
31. IDC (2009) Aid to recovery: the economic impact of IT, software, and the Microsoft ecosystem on the global economy
32. ISO/IEC/IEEE 24765 (2010) Systems and software engineering – Vocabulary
33. ISO26000:2010 (2010) Guidance on social responsibility. https://www.iso.org/obp/ui/#iso:std:iso:26000:ed-1:v1:en
34. Johann T, Dick M, Kern E, Naumann S (2011) Sustainable development, sustainable software, and sustainable software engineering: an integrated approach. In: 2011 International symposium on humanities, science & engineering research (SHUSER), pp 34–39
35. Kern E, Dick M, Naumann S Guldner A, Johann T (2013) Green software and green software engineering – definitions, measurements, and quality aspects. In: First international conference on information and communication technologies for sustainability, pp 87–94.
36. Koomey J (2011) Growth in data center electricity use 2005 to 2010. Analytics, Oakland, CA, August 1. http://www.analyticspress.com/datacenters.html
37. Manteuffel C, Loakeimidis S (2012) A systematic mapping study on sustainable software engineering: a research preview. In: 9th Student colloquium, pp 35–40
38. Masanet E, Shehabi A, Ramakrishnan L, Liang J, Ma X, Walker B, Hendrix V, Mantha P (2013) The energy efficiency potential of cloud-based software: a U.S. case study. Lawrence Berkeley National Laboratory, Berkeley, CA. http://crd.lbl.gov/assets/pubs_presos/ACS/cloud_efficiency_study.pdf. Accessed in April 2014
39. Merriam-Webster (2013) http://www.merriam-webster.com/
40. Mocigemba D (2006) Sustainable computing. Poiesis & Praxis Int J Technol Assess Ethics Sci 4(3):163–184
41. Mohan K, Ramesh B, Cao L, Sarkar S (2012) Managing disruptive and sustaining innovations in Green IT. IT Prof 14(6):22–29
42. Molla A, Cooper VA, Pittayachawan S, IT and Eco-sustainability (2009) Developing and validating a Green IT readiness model. In: International conference on information systems (ICIS 2009). Paper 141. Retrieved from http://aisel.aisnet.org/icis2009/141

43. Murugesan S, Gangadharan GR (eds) (2012) Harnessing Green IT: principles and practices. Wiley, UK. ISBN: 978-1-119-97005-7
44. Murugesan S, Laplante PA (2011) IT for a greener planet. IT Pro January/February, 16–20
45. Murugesan S (2010) Making it Green. IT Prof 12:4–5
46. Naumann S, Dick M, Kern E, Johann T (2011) The greensoft model: a reference model for green and sustainable software and its engineering. Sustain Comput Informat Syst 1(4):294–304
47. Penzenstadler B, Fleischmann A (2011) Teach sustainability in software engineering? In: 2011 24th IEEE-CS conference on software engineering education and training (CSEE&T), pp 454–458
48. Penzenstadler B, Femmer H (2013) A generic model for sustainability with process and product-specific instances. In: Proceedings of the 2013 workshop on green in/by software engineering 2013, ACM, Fukuoka, Japan, pp 3–8
49. Penzenstadler B, Raturi A, Richardson D, Calero C, Femmer H, Franch X (2014) Systematic mapping study on software engineering for sustainability (SE4S). In: 18th International conference on evaluation and assessment in software engineering
50. Penzenstadler B et al (2012) Sustainability in software engineering: a systematic literature review for building up a knowledge base. In: 16th international conference on evaluation and assessment in software engineering (EASE)
51. Penzesdtadler B (2013) Towards a definition of sustainability in and for software engineering. In: SAC'13. ACM 978-1-4503-1656-9/13/03, pp 1183–1185
52. Goodland R (2002) Sustainability: human, social, economic and environmental, Encyclopedia of global environmental change. Wiley, UK
53. Seidel S, vom Brocke J (2010) Call for action: investigating the role of business process management in green IS. In: SIGGreen proceedings. http://siggreen-icis2010-wokshop.wikispaces.com/file/view/SIGGreenICIS2010WorkshopEBook.pdf/189649215/SIGGreen-ICIS2010WorkshopEBook.pdf, pp 132–133
54. Sobotta A, Sobotta I, Gotze J (eds) (2010) Greening IT. How greener IT can form a solid foundation for a low-carbon society. The Greening IT Initiative Italy. ISBN 978-87-91936-02-9
55. Sroufe R, Sarkis J (eds) (2007) Strategic sustainability: the state of the art in corporate environmental management systems. Greenleaf, Sheffield, UK
56. Steigerwald B, Agrawal A (2011) Developing green software. https://software.intel.com/enus/node/183291?wapkw=developing+green+software
57. Taina J (2011) Good, bad, and beautiful software. In search of green software quality factors. CEPIS Upgrade XII 4:22–27
58. Tate K (2006) Sustainable software development: an agile perspective. Addison-Wesley, Upper Saddle River, NJ
59. The Climate Group (2008) SMART 2020: enabling the low carbon economy in the information age. The Global eSustainability Initiative, Brussels
60. Tomlinson B, Silberman SS, White J (2011) Can more efficient IT be worse for the environment? Computer 44(1):87–89. doi:10.1109/MC.2011.10, ISSN:0018-9162
61. Unhelkar B (2011) Green IT strategies and applications. Using environmental intelligence. CRC, Boca Raton, FL
62. United Nations World Commission on Environment and Development (1987) Report of the World Commission on Environment and Development: our common future. In: United Nations conference on environment and development
63. vom Brocke J, Seidel S, Recker J (eds) (2012) Green business process management: towards the sustainable enterprise. Springer, Berlin, p XII, 263 p
64. Watson RT, Boudreau M-C, Chen AJW (2010) Information systems and environmentally sustainable development: energy informatics and new directions for the IS community. MIS Q 34(1):23–38
65. IEEE (2014). SWEBOK V3.0. Guide to the Software Engineering Body of Knowledge. Bourque, P. and Fairley, R.E. (eds.), IEEE Computer Society

Part II
Environments, Processes and Construction

Chapter 2
Green Software Engineering Environments

Ankita Raturi, Bill Tomlinson, and Debra Richardson

2.1 Introduction

The industrial world is made up of highly technological energy-hungry societies that struggle with numerous issues regarding the environmental impacts of our current use of technology. While researchers are often concerned with the carbon footprint of the transport sector, waste management issues in homes and buildings and even consider the energy consumption of the overall IT sector, rarely do they consider the role of software engineering in the environmental impact of our computing technologies.

The goal of this chapter is to describe what kinds of methods, metrics and tools exist and what opportunities there are in these areas to support environmental (particularly energy) issues that exist within software engineering environments. In Sect. 2.2, we describe some motivating questions and issues that exist in software, sustainability and the cross section of the two. Section 2.3 contextualises the need for greening our software engineering environments (SEEs) due to the environmental impacts of IT. We will then break down what the structure and components of a Green SEE are in Sect. 2.4. We instantiate some of these ideas through the example of Joulery, an energy awareness tool for the Green SEE in Sect. 2.5. Finally, we try to tie all the different ideas discussed in this chapter back together with some conclusions in Sect. 2.6.

A. Raturi (✉) • B. Tomlinson • D. Richardson
University of California, Irvine, Irvine, CA, USA
e-mail: araturi@uci.edu; wmt@uci.edu; djrg@uci.edu

2.2   Motivation

In his 1966 paper, *The Economics of the Coming of Spaceship Earth*, Kenneth Boulding states, 'Systems may be open or closed in respect to a number of classes of inputs and outputs. Three important classes are matter, energy, and information' [12]. He uses this classification to analyse systems such as the world economy, human societies, the human body and other aspects of the system of the globe. We can also consider software production as a global system, with matter, energy and information inputs and outputs of its own, where software engineering is concerned with ways to understand and improve these classes.

The basic definition of software engineering as defined by the IEEE standard is: 'application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software' [34]. This description is useful as it gives us a basis for understanding what the different concerns and components of software engineering are and what exactly comprises the software engineering environment.

Software itself is an open system where the majority of the inputs and outputs are oriented towards the information aspect; software engineering tends towards scrutinising and optimising the information involved. Software engineering environments consist of a variety of devices used in the actual production of software.

Boulding's matter class can be instantiated to also include computer hardware, buildings, equipment and other physical components of the software engineering environment. Reductions in the matter involved in software engineering lead to direct reductions in financial costs and space requirements. Due to the immense growth of the software industry, the amount of energy that is consumed during software construction is rapidly increasing. Software engineers are increasingly interested in investigating the energy involved in software construction as it would not only reduce financial costs but also address the global energy crisis.

Boulding's three 'classes' can be considered in concert to work towards discerning variables, with respect to which software engineering can be tweaked to reduce the amount of matter and energy consumed while still ensuring that the quality and purpose of information produced are enhanced. However, these classes are quite broad, and attempting to address all three at once would be a project of a rather global scope. The classes could also be treated independently; however, any changes made to one class may affect another due to their interconnected nature. For example, in order to reduce the amount of energy consumed by certain hardware, yet another piece of equipment may need to be introduced into the system. This means that any analysis of the matter, energy or information involved in a software engineering environment should be mindful of the potential impact on other classes.

In Fred Brooks' 2003 paper *Three Great Challenges for Half-Century-Old Computer Science*, he states:

1. 'Due to the highly complex nature of software and its components, quantifying this complexity is difficult.

2. Due to the variables involved in building software (functional, reliability and performance specifications), we have issues estimating different metrics for software including development time.
3. Due to the wide demographic we build software for, building usable, meaningful and intuitive user interfaces is more like an art than an engineering task making it difficult to justify good design' [13].

These three challenges seem especially relevant when we are just beginning to try to establish software engineering metrics and methods to make both the process and the product more environmentally aware. Issues that researchers face in relation to these challenges could potentially include:

1. How do we decompose the software engineering process and product into quantifiable components for accurate analysis?
2. How do we ensure that the methods and metrics we develop are accurate and correctly estimate the environmental impact of software engineering?
3. How do we build tools that provide software engineers with useful and meaningful data that can be visualised for a greater understanding of potential environmental considerations of software engineering?

From an examination of the existent literature in sustainable software engineering, efforts by researchers such as Naumann et al. [51] involving the life cycle of software and the work of Capra et al. [15] on the energy complexity of software are actively trying to answer the first two questions. There remains a gap when it comes to constructing tools to provide software engineers with environmental impact data. These tools could include energy monitors and resource trackers and should provide meaningful visualisations of such data.

Current efforts involve energy monitoring tools that either focus entirely on data centres as a whole or only on individual activity agnostic devices. In addition, other software monitoring tools such as bug trackers give developers information regarding other metrics such as the health of their source code. There is a need to link environmental impact data across the various components (including the data centre) in a software engineering environment with the goal of providing a comprehensive environmental impact map of the system. How this map is presented or put together, and what information is conveyed, is a design problem that spans across the software engineering landscape.

In discussing means of integrating new tools into a software engineering environment, Thomas and Nejmeh state that we should ensure that any represented information is consistent with the rest of the engineering environment, no matter what data are being transformed [61]. In this sense, it would be important to make sure that any work done visualising such data is consistent with the other types of software environment information already available to engineers. It is important to design solutions that are as flexible and customisable as possible while being aware of the complexity of the systems with which they interact. Due to the varying nature of these engineering environments, modular solutions would best allow for such considerations.

2.3   Contextualising the Need for Green Software Engineering Environments

Information technology has played a key role in attempting to address environmental issues while also contributing to a rise in certain other kinds of environmental issues. However, these two concepts can be considered independently.

We can first consider IT that is being used to actively mitigate environmental issues. To clarify terms, we make a distinction between 'Green IT' and 'greening through IT'. 'Green IT' is technology that is in and of itself attempting to reduce its own environment footprints. Murugesan describes Green IT as: 'the study and practice of designing, manufacturing, using, and disposing of computers, servers, and associated subsystems... ...efficiently and effectively with minimal or no impact on the environment' [50]. 'Greening through IT' [62], on the other hand, is a broader focus on how information technology may make a wide range of other sectors of civilisation more sustainable through the application of IT in those domains.

A broad range of work has been done to ensure that the IT we design and build reduces its own resource consumption and consumption by the systems that it is embedded in. However, due to the sheer vastness and the embedded nature of IT in all sectors, this has proven to be an enormous task that requires cross-sector collaboration. Computers and the Internet are considered general-purpose technologies [36] which can have major impacts on the global economy and society at large, which makes environmental and energy concerns of software global concerns too. The Smart 2020 report describes four mechanisms by which IT can be used to enable other sectors: smart grids, smart buildings, smart logistics and smart motor systems [65]. These categories provide a sampling of the various types of technologies that are being built with the purpose of enabling greening through IT. The IT sector itself is said to be responsible for 2 % of global CO_2 emissions [47], and the overall environmental impacts of this sector also include large amounts of energy consumption [14] and consumption of a variety of other materials [59], as well as the production of e-waste and hazardous substance waste.

One of the major environmental concerns that we are dealing with right now is the world energy crisis. The World Energy Outlook 2010, an IEA report, states that 'the age of cheap oil is over' [10], describing rising energy prices and stressing on the need for a move towards renewables and low- and non-carbon-dependent energy sources. What this means is that the energy consumption of all sectors needs to be minimised.

Due to the growing needs of the IT sector, energy consumption of IT is also becoming a point of investigation [22]. With larger infrastructure, massive power-hungry data centres and embedded technologies in many daily use objects, the potential impacts of the IT sector's footprint could ripple to everything it touches. The energy demands of IT can seem modest in comparison with other sectors. In their 2002 book, *Electricity Requirements for a Digital Society*, Baer, Hassell and Vollard projected that even a dramatic growth in the IT sector could result in only a

moderate increase in electricity use over the next 20 years, remaining under 5.5 % of the national electricity total [6]. They do mention that the projections are based on scenarios involving 'incomplete data and a large number of assumptions about how the future will unfold' [6]. These assumptions include the growing role of IT in power management, continuing technological improvements, interest in telework opportunities (which would skew the energy use burden to residential areas) and so on.

While numbers such as 2 % of global CO_2 emissions and 5.5 % of a national electricity total are only part of a global total, it is important to remember that these numbers are only of the IT itself and do not take into account the domino effect of these systems to other sectors. Elliot and Binney state that these numbers are still 'too large to be maintained' [22]. So, in order to create more complete datasets on which to base better projections and to actually minimise current consumption, it would be useful to have a research agenda in the IT sector that deals specifically with environmentally sustainable IT.

Murugesan [50] lists five reasons for the push to Green IT: lower costs, lower power consumption, lower carbon emissions and environmental impacts, space savings and improved performance and use of systems. Mingay's report [47] describes most succinctly the idea that reducing the impact of IT, including its energy and carbon footprints, is vital due to the entangled nature of environmental issues. Briefly stated, the idea is that any reduction here can lead to another reduction there.

2.4 The Green Software Engineering Environment

Let us begin with the International ISO/IEC/IEEE 2476 Standard for Systems and Software Engineering Vocabulary definition of the software engineering environment:

Clause 3.2761 software engineering environment (SEE)

1. environment that provides automated services for the engineering of software systems and related domains, such as project management and process management. *ISO/IEC15940:2006, Information Technology Software Engineering Environment Services.2.2.1*
2. hardware, software and firmware used to perform a software engineering effort

NOTE: It includes the platform, system software, utilities and CASE tools installed.

The SEE can be broken down into four layers: infrastructure, hardware, software and people, where each is defined as follows:

- *Infrastructure*: Includes physical structures (e.g. buildings, data centres) and utilities (e.g. network cabling, water supply systems) that are used to support and house different software engineering activities.

- *Hardware*: Includes computers, networking equipment, servers, mobile devices, etc. Hardware refers to physical elements that perform computation related to different software engineering activities.
- *Software*: Includes collections of computer programs used during software engineering activities, that is, the platform, system software, firmware, utilities and CASE tools.
- *People*: Includes designers, developers, architects, project managers, etc., people who are directly involved in different software engineering activities.

In this section, we discuss what types of solutions are available with respect to the infrastructure, hardware, software and people involved in the SEE.

2.4.1 Green Infrastructure for the SEE

Most technology today contains some form of software, whether it be an application running on a computer or an embedded program in a microcontroller. Software is a multibillion dollar industry requiring an immense supporting infrastructure with far-reaching impacts. Building software itself is a highly complicated matter. A majority of the work that has been done in enabling software engineering to be a more environment-friendly process has focused on optimising data centres and virtualisation and energy-efficient protocols, creating data collection tools for specific systems and investigating energy-aware cloud computing options.

One of the most comprehensive analyses on the performance of data centres comes from the book *The Datacenter as a Computer* [7] by Barroso and Holzle.

They look at hardware performance and design decisions, monitoring of large-scale systems and, most relevantly, the energy and power efficiency of the data centre as a whole. They also discuss ways to calculate PUE (power usage effectiveness), an energy conversion term devised by The Green Grid [55] that takes into account the energy efficiency of all the individual load-bearing components inside a data centre. Overall, they provide a rather detailed method using well-accepted metrics for analysing data centre performance. They focus mostly on the major culprits: cooling and server power supplies.

According to similar work, 60–70 % [9] of the energy used in data centres goes entirely to cooling. Companies like Google have invested a lot of time, effort and money in improving their servers, network infrastructure and component efficiency, as well as reducing the overall environmental impacts of their systems. They report servers that lose around 15 % of the electrical input, which is half the waste of normal servers [29]. Google has also published details of innovative cooling methods that involve simply using water evaporation to all heat dissipation within their data centres, reportedly reducing their energy-weighted average overhead from the 96 % of standard servers to the 19 % of a Google server [24].

Another major issue faced by data centres concerns always-on policies, where servers are left powered up, regardless of demand, which has led to work on

dynamic power management [8, 48]. Some of the most common processors currently in use can put out around 100 watts of heat alone [28], which could contribute to the overall cooling issues that data centres face. The Advanced Configuration and Power Interface Specification (ACPI) provides four different types of power states that a computer with a processor that is ACPI enabled can utilise to manage local wake states according to computational requirements [31].

This has also given rise to interest in virtualisation as an energy-saving mechanism. This essentially involves spawning virtual devices on demand, thus reducing the number of physical components one has and allowing the existing devices to be used in a more optimal manner [60]. Liu et al. [42] suggest scheduling tasks throughout a data centre in a distributed manner, while Fan et al. [24] propose hardware-based power provisioning strategies.

There is no dearth of variety in mechanisms to reduce the energy impact of data centres. Issues arise in combining these methods and finding tools that are platform agnostic. In addition, most of the concepts described here only address the data centre and do not look at infrastructure in the SEE. While the data centre is assumed to be one of the most energy-intensive parts of an engineering environment, the lack of integration of the data from this end with the energy data of the developer's end could pose an issue.

2.4.2   Green Hardware for the SEE

There exist myriad tools in both software and hardware dedicated to local device energy use monitoring. The primary push seems to come from consumer electronics, where commercial engineers are designing tools for users to monitor their own energy consumption. The goals of these tools are both energy awareness and cost savings as well as being a means to enable consumers to make greener decisions. The interesting part about these tools is that they can also be considered developer tools as they can be used by individual software engineers to monitor local device performance.

One of the first initiatives geared towards informing users of the energy consumption of their electronics was the Energy Star program. Energy Star is a labelling program by the EPA and the DOE and is geared towards giving manufacturers energy efficiency standards which they can use in the design of consumer products promoting them under this green label [2]. In efforts to meet these standards, manufacturers of computers and their peripherals, home appliances, hardware for utility infrastructure and even some organisations in the service industry have improved their production processes and in-use energy consumption of their products. According to studies conducted on the savings made by companies working actively to meet these standards, 5.2 billion dollars in energy costs or 760 petajoules of energy was saved cumulatively between 1993 (the introduction of Energy Star) and 2000 [58]. These kinds of savings are only projected to increase with more and more areas of the IT sector and other industries affected moving
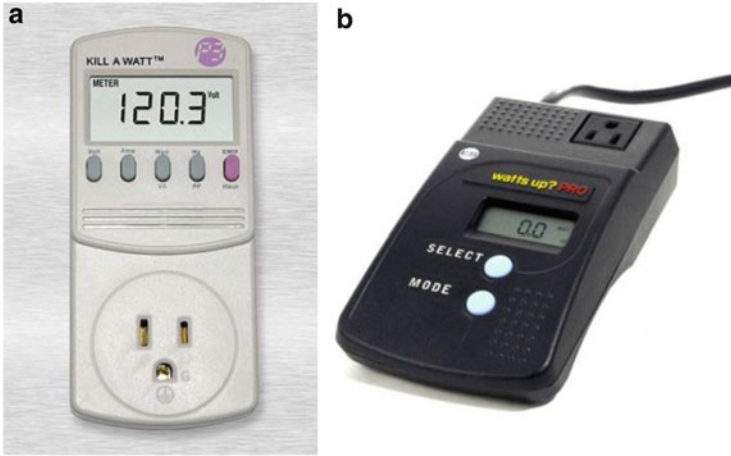
Fig. 2.1 Two popular tools for device energy monitoring: (**a**) Kill A Watt and (**b**) Watts Up Pro

towards producing and using greener technologies. Making sure that the devices used in the SEE are highly rated with respect to Energy Star would mean that the energy impact of the SEE hardware layer is reduced.

The hardware tools that exist are mostly in the form of energy metres. Examples of these devices can be seen in Fig. 2.1. They simply plug into the power outlet, and then the device to be monitored is plugged into them. The range of data provided includes volts, amps, watts, etc. One of the Watts Up products is Web enabled allowing remote monitoring of a device. There is potential to use these to do widespread power monitoring in engineering environments. However, in order to support device-level consumption data, one would need a single-purpose piece of hardware (the monitoring device) for every device to be monitored. A major issue that could arise with the proliferation of individual energy monitoring devices is an increase in e-waste.

2.4.3   Green Software for the SEE

Rosseto points out that 'contrarily from all the other layers, energy efficiency of the software layer of a data centre remains largely unexplored' [57], as is exemplified in Fig. 2.2. You can see that while power input and losses between the source, networking layers and even down to the hardware are known, there is limited energy data during computation, related code and eventually the operation the software does.

There also exists a family of software applications dedicated to power monitoring on computers. For Linux, there exists PowerTOP, which is geared mostly towards developers [18]. It provides detailed power use information per CPU
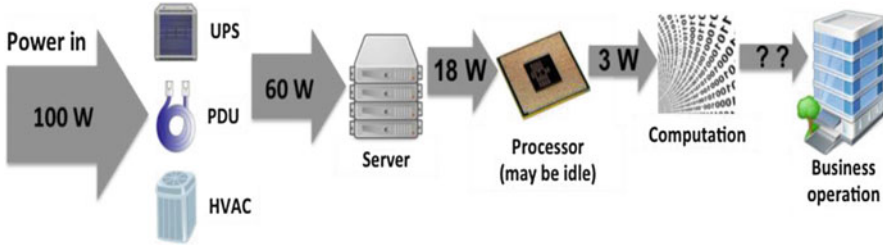
Fig. 2.2 Power consumption breakdown in a data centre (adapted from [14])

cycle, process, application and so on. It is actually a command line-based overlay for Top, a C-based tool that dynamically displays a text output of the top CPU-consuming processes on a local machine [40].

JouleMeter [56] is a Windows power monitoring tool that provides basic power reports and is targeted mostly towards casual users. It requires calibration either via a hardware device like the Watts Up when on desktop machines or via a battery run on laptops (using the battery delta to estimate power use). Current work on JouleMeter has begun to look at using learning mechanisms to make estimates of power consumption from baseline device data [37].

Researchers in green computing and energy consumption have also built a variety of tools. In their work on estimating power consumption during the use of specific software application, Amsel et al. [4, 5] developed GreenTracker, which collects and then displays the graph data for a sample set of applications. This provided the community with both a methodology to compare software power consumption and some insight into the discrepancy between the consumption of different Web browsers, media players and word processors.

PowerScope [27] is another research tool which consists of a system monitor that tracks system calls on the device being profiled and an energy monitor on a separate analysis device that collects current samples from a digital multimeter attached to the device being profiled.

The next group of tools that are designed to enable sustainable software engineering are at the algorithmic level. These are very context-specific tools that can be used to perform tasks akin to code inspections. JConsole is a graphical Java monitoring tool which provides details on CPU usage, memory usage, thread activity, etc., but only of Java applications and subprocesses [19]. JavaSysMon is an API written in C and Java that allows sampling of CPU usage and memory usage data [33]. The intended use requires an application to be written over it to perform the dynamic sampling and visualisation.

There has not been a lot of work that focuses on visualising energy data in a meaningful manner. Most of these are for residential and personal monitoring purposes. Fig. 2.3a shows a project which aimed to visualise smart metre data from consumer homes on their mobile phones [38]. Visually parallel to this work is the application Usage Timelines [41], also for mobile devices, in Fig. 2.3b. This
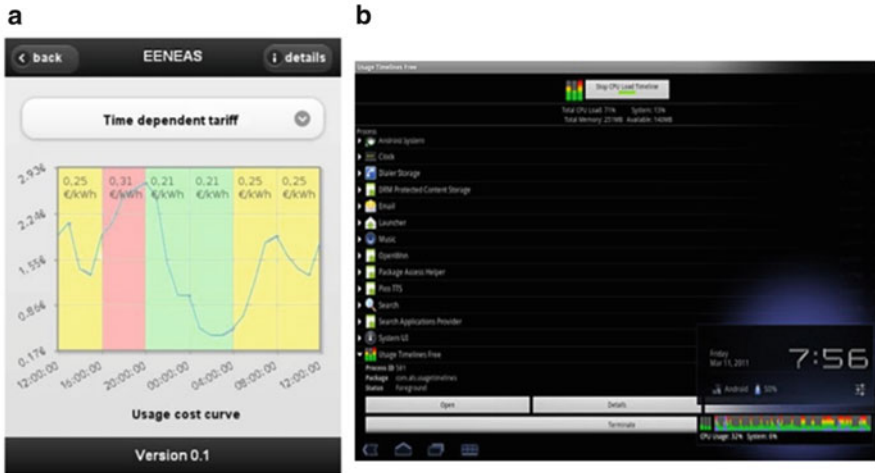
Fig. 2.3 Tools for visualising energy consumption: (**a**) Tariff tool [38] and (**b**) usage timelines [41]

application displays CPU usage per application and per process and has a live graph visualisation for total CPU usage and contains an interface to view details and potentially terminate each application on the local device. We consider these tools as motivating examples for the types of energy data visualisations that are lacking for the SEE. We discuss an analogous tool for the SEE in Sect. 2.5.

Other visualisations that are available for computer-based energy data are also single device based and textual. PowerTOP and JouleMeter, discussed earlier, only display text output as can be seen in Fig. 2.4a and b, respectively. Watts Up, also mentioned in previous sections, has a tool for sale, Logger Pro, that can log and visualise data that comes from the Watts Up device.

2.4.4   Green Behaviour in the SEE

Lockton et al. describe the field of 'design for sustainable behaviour' [43], in which designers consider how they can influence users to act in a more environment-friendly manner with respect to their use of products, services and environments. Within the SEE, we can consider software engineers as users, where they are the people performing activities, using tools and interacting with the SEE, resulting in an environmental and energetic impact. Taking a look at software engineers operating within the SEE would enable us to design better methods, metrics and tools to encourage green behaviour.

The first thing to consider is the Jevons paradox. Alcott's analysis of the Jevons paradox describes it as the issue where increases in technological efficiency in consuming a resource increase the consumption rate of the associated resource; for example, 'a more fuel-efficient car enables one to drive more' [3]. In 2010, David
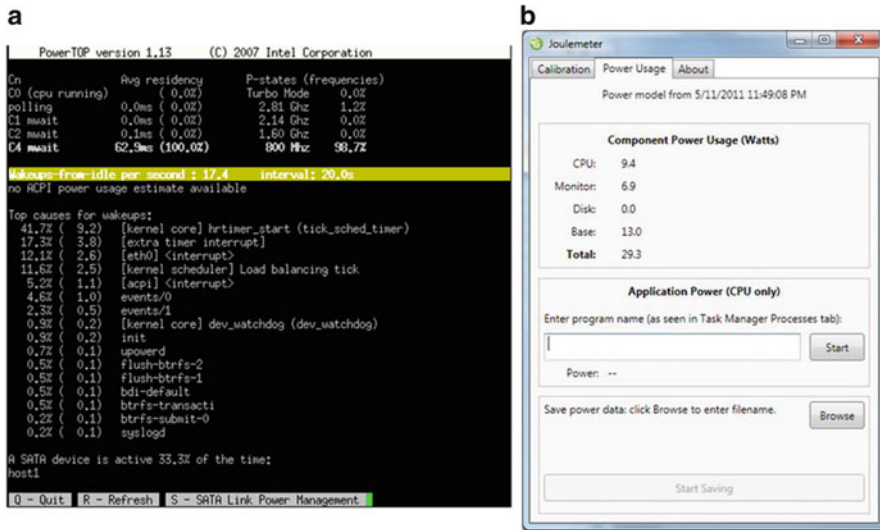
Fig. 2.4 Tools for visualising energy consumption: (**a**) PowerTop [18] and (**b**) JouleMeter [56]

Owens wrote an article in *The New Yorker* titled *The Efficiency Dilemma* [53] where he also described how energy efficiency could result in greater energy use. In a rebuttal to this article, Lovins, a scientist at the Rocky Mountain Institute, writes 'Rebound effects are small in energy-using devices for three reasons: no matter how efficient your house or washing machine becomes, you won't heat your house to sauna temperatures, or re-wash clean clothes; you can't find an efficient appliances savings in your unitemised electric bill; and most devices have modest energy costs, so even big savings look unimportant' [44]. There are certain aspects of the SEE that would also have similarly small rebound effects. Time and effort required to perform software engineering activities play a larger role in whether they are repeated or optimised. A developer will not unnecessarily recompile code that is bug-free just because the energy cost is low.

Lovins' idea can be drawn upon when considering how to reduce the energy impact of the SEE and, consequently, the IT sector. SEE infrastructure, hardware and software all exist to meet certain computational needs. If the SEE grows, it is not out of energy avarice but because there is a need for more computation. The demand for computation and the demand for information result in a demand for energy as the SEE will grow to support it. However, the issue is not the cause of the growth or the value of energy efficiency but of our awareness of energy issues and of our consumption habits themselves [17]. If we engage in adequate self-reflection on the way in which our activities affect our energy footprint, we can begin to make decisions that allow for reduced energy consumption. For example, in 2010, the United States consumed almost 100 quads of total energy [1]. According to Laitner, under the current path the United States would be consuming 122 quads by the year 2050 based on projected growth. He then describes two scenarios where

energy-saving changes are implemented, including better space heating and cooling which would lead to a drop in consumption to between 50 and 70 quads under the same projections [39]. To avoid unbounded growth, we would need to make sure that we are actively implementing efficiency techniques at the infrastructure, hardware and software layers of the SEE. We can see that if left unchecked, the demand for computation could result in large increases in things like the size of data centres and the amount of hardware in the SEE. Therefore, to ensure that we are actively moving towards a Green SEE, we too would need to make sure that appropriate energy-saving changes are made to the SEE, as well as the way in which software engineering activities are conducted.

Due to the variety in energy-consuming nodes in the SEE, there are several tensions that determine whether or not an energy-efficient SEE would suffer from rebound effects or not. As an example, during the testing phase of development, we can generate large numbers of test cases. Regression testing involves running previously passed tests and tasks again in order to check if a new change has introduced any bugs. As the time and effort cost to rerun a large test suite are reasonable, a tester may not think twice about doing this over and over again. So, what was initially a low-energy activity may now result in more energy being consumed. There is a trade-off between the various types of testing that can be done and the energy consumption of the activities that should be kept in mind. Just because the system is efficient does not mean certain behaviours cannot make it wasteful.

Doyle [20] discusses the role of technology in a more sustainable world and concludes that one of the primary goals of Green IT would be to 'maximise energy efficiency during a product's lifetime'. This brings us to a challenge that has been offered to the environmental value of improving efficiency within the SEE. Does reducing the energy footprint of software engineering encourage green behaviour?

Tomlinson makes a suggestion that in order to combat the Jevons paradox, we should consider 'intelligently applied efficiency' [63]. Under this principle, yes, energy-efficient systems may be beneficial in some domains, but they should also be supported by green policies that actively encourage less consumption in general to deter the urge to expand. So, to ensure that software engineers are supported in their quest for low-energy impact behaviours in the activities within the SEE, there need to be methods, metrics and tools, such as those discussed in this chapter, that allow for introspection.

We also need to consider what aspects of the SEE support software engineers in different kinds of decision-making, that is, how are designers encouraged to create more usable interfaces, how are programmers supported when they make more secure or maintainable code and how are project managers assisted in selecting time- and money-saving options? In the paper *Safety, Security, Now Sustainability: The Non-Functional Requirement for the 21st Century* [54], Penzenstadler et al. discuss how requirements engineers were first encouraged to consider safety and security requirements in software systems and what the move towards considering sustainability requirements looks like. The paper describes needs that would

alter the behaviour of requirements engineers to encourage them to include environmental issues in the specification of a software system.

We can conclude that we need to incentivise software engineers to behave in a greener fashion by equipping them with the things they need: metrics, methods and tools. By striving towards a cohesive Green SEE, the barrier to making greener decisions is lowered. We describe some of these decisions in detail in Sect. 2.5.4.

2.5   Example Tool: The Joulery Energy Dashboard

The Joulery Energy Dashboard instantiates some of the ideas discussed in this chapter. There is a need for creating tools that address a combination of the material, information and energy aspects of system involving infrastructure, hardware, software and people layers in the SEE. It is difficult to create a cohesive set of method metrics and tools that address all of the above. Instead, it is both useful and feasible to address a subset. In this section, we will present the Joulery Energy Dashboard, describe its components and design as well as describe how it fits into the overall landscape of a Green SEE. Joulery aims to visualise the energy footprint across the infrastructure, hardware and software layers to enable green behaviours and considers the information and energy aspects of the SEE.

Current energy monitoring tools focus either entirely on large infrastructures (like data centres) or on very localised monitoring of individual devices. Networks of devices are at the core of many aspects of industrial society, being crucial parts of both homes and workplaces, as well as being the support structure of many large industries. In a 2004 article, Christensen et al. describe the need for power management of networked devices, stating 'Energy consumption by digitally networked devices is expected to increase at a faster rate than other types of energy use in buildings' [16]. They discuss research needs such as investigating what and where the energy problems in networks lie. We argue that a mid-scale analysis is an important complement to large-scale and small-scale energy investigations, because it can help the kinds of human organisations, such as corporations, make decisions more effectively about system-scale phenomena that are under their own control, rather than focusing only on optimising individual components.

Now, at the core of the SEE lie physical networks. These networks contain heterogeneous devices that range from high-performance servers, a variety of computers and laptops, storage, input and output devices as well as miscellaneous mobile devices. Figure 2.5 shows some of the connections in the SEE. These devices are used for a range of software engineering activities by a variety of people. There are developers writing code on Linux-based laptops, software architects designing components using tablets, requirements engineers using teleconferencing equipment to communicate with clients, database engineers building repositories on servers and so on. In order to provide insight into how each of these people can modify their behaviours to perform less energy-intensive activities, we created the Joulery Energy Dashboard.
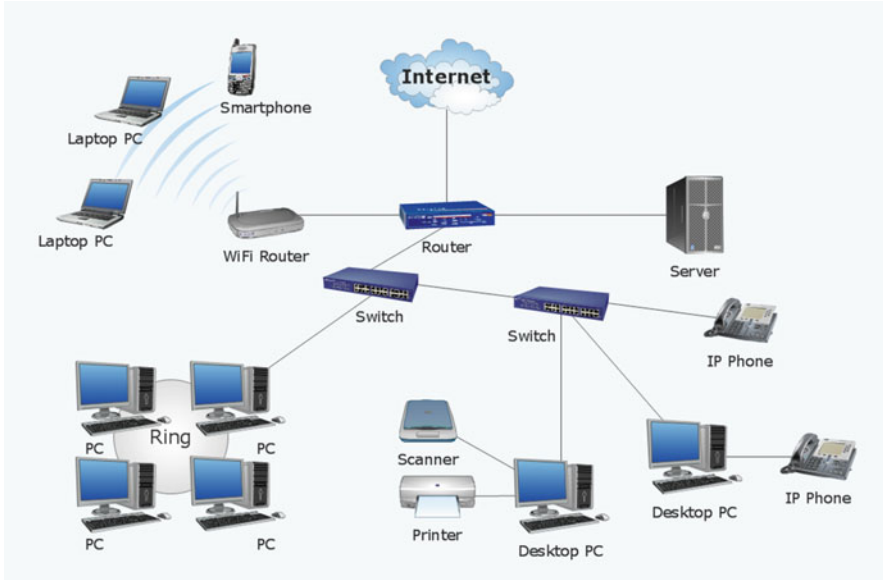
Fig. 2.5 Hardware involved in an example software engineering environment [52]

The Joulery tool consists of a method for consolidating energy data from networked devices and provides auto-visualisation of the energy performance of the network and its components at a granular level. The goal is to enable the identification of energy sinks in the network which can then be tied back to the different activities being conducted.

We begin the design of Joulery by treating each node in a physical network in the SEE in a device agnostic manner and viewing them as resource-consuming entities, specifically their energy-consuming entities. Figure 2.6 describes the manner in which the data for each level (process, application, device and network) would compound to form the overall energy footprint of the software engineering environment.

Software tools (including Joulery) designed to analyse the environmental impact attempt to provide a subset of the following three functionalities:

1. Some level of granularity of energy data that is the base for a potential energy metric
2. A means for energy data collection via hardware tools or estimations
3. Some presentation of dynamic energy consumption data that enables better decision-making
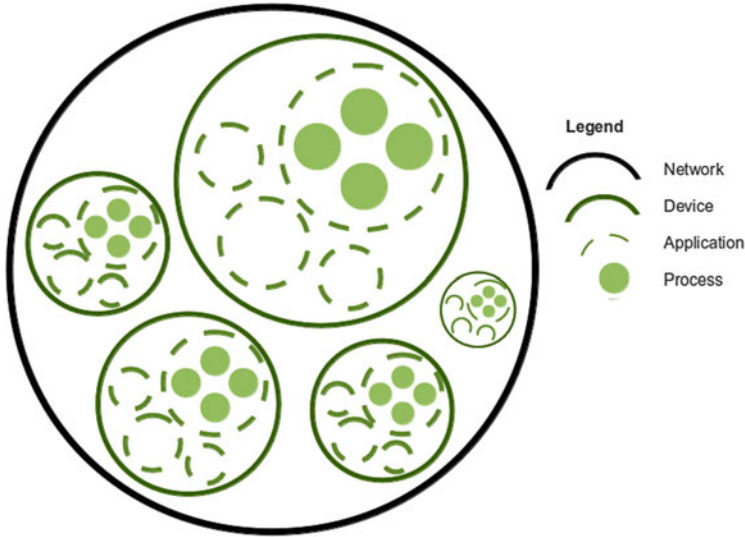
Fig. 2.6 Device nodes as part of a network in a software engineering environment

2.5.1   Consolidating Energy Data

The hardware involved in the SEE (Fig. 2.5) is structurally similar to a data centre when we are thinking about the way in which to decompose the system into resource-consuming nodes. In a white paper by an editor from The Green Grid, Mark Blackburn states, 'A low cost, low disruption method bases power usage calculations on a server's CPU utilisation' [11]. This is based on work by Fan et al. that shows a linear relationship between power usage and CPU utilisation [24]. Mittal et al. [49] conduct similar investigations into energy consumption of mobile devices, and while the impact of CPU use is significant, other components such as the display and networking components are too.

There are a variety of ways in which energy data can be collected, but for scoping purposes, Joulery only focuses on Linux computers using software-based loggers, as the Linux kernel allows for direct access to system data at process level. While the goal would be to avoid using a hardware logger, this prototype requires the use of one to provide a benchmark value for the conversion of CPU utilisation to energy consumption discussed later.

The goals of consolidating the energy data of a network are to obtain:

1. An overall metric describing the total energy consumed by the network
2. Device-specific data that is accessible from a central location for analysis, manipulation and visualisation
3. Application- and process-specific energy data per device to allow for increased data granularity

One means for consolidating energy data is to create data loggers to sit at the device nodes and have a data aggregator located on a central dedicated server. The reason for this role split is to reduce processing overhead induced on the nodes during aggregation by offloading this responsibility to the server.

2.5.1.1 Benchmarking

Heeding advice from centre energy efficiency research, and as suggested by Blackburn [11], Fan et al. [24] and Barroso and Holzle [7], to get energy consumption of a system, one needs, at the very least, benchmark data regarding power use levels of the devices. Barroso and Holzle describe a set of benchmark data that is available for standard servers.

A Watts Up Pro device was used to measure a computer's power consumption over an hour to get the average idle power. This was then repeated but with a CPU stress test run during the hour and the average maximum power recorded. Prime95 [46] is an application that calculates Mersenne prime numbers (prime numbers that are one less than a power of two) and is popularly used as a device stress-testing tool. MPrime, a Linux command line port of this application, was used to conduct the stress tests and obtain benchmarks for a computer running Ubuntu 12.04. The stress tests produced a range of values, which were then hard-coded into the logger for use as described in the following sections.

2.5.1.2 Data Logger

The process involved in getting system data through a Linux client logger is quite straightforward. The '/proc' folder, known as a 'process information pseudo-file system', contains all the system runtime information of the local machine. All the files contained here are essentially system information at the kernel level. Each process running on a system also has its own subfolder here that then contains process-specific information.

The /proc folder is scanned for relevant data which are logged and then pushed to the server. In the Joulery prototype, only the energy consumed due to CPU usage is considered, as it consumes the most energy compared to other standard computer components [45]. This is important to note, as there are potentially many other energy-consuming components in a device such as the graphics card, networking card, attached peripherals, etc.

In addition to per process CPU consumption, the logger also reports overall CPU data. The method here is akin to that used in the tools: ps, Top and PowerTOP [18, 40]. Importantly, note that two data samples must be taken with a small fixed time interval and metadata needs to be attached to the log file demarcating the log as either a 'before' or 'after' sample. These paired samples are used on the server side to parse and calculate the energy used by the device.

It should be noted again that this is just one method to access energy data on a device, and different devices have different kinds of support for access to such data. For example, ACPI-compatible Linux devices have direct access to local energy data [31]. So, for each type of client logger that is created, a corresponding parser on the server side must be written to handle relevant calculation and storage.

2.5.1.3 Data Aggregator

As the client-side logger is responsible for pushing to the server, the primary responsibility of the server involves knowing which device is sending what and where and how to store the relevant data logs. Therefore, the server requires relevant methods to parse and store incoming logs. The corresponding parser for the data logger described in this section runs the data logs through a series of manipulations before we can consider the incoming data to represent client energy consumption data.

First, a log file is received from a client data logger (see Fig. 2.7 for a partial log). This log contains four categories of information: a unique client ID, the benchmark power values, a timestamp and system data for each running process. The client ID can be the MAC address and IP address, or if the names of the devices in the network are unique, then that too is usable (as in the examples here). The timestamp is in ISO 8601 format to allow for compatibility and portability between device times and the other modules involved in this framework.

Line 5, the result of 'cat /proc/stat', contains overall CPU data. The consecutive numbers after the 'CPU' label in this line correspond to the amount of time the CPU has spent in various modes (user, system, etc.). The first three numbers are the time spent scheduled in user mode, system mode and nice mode, which total to the amount of time the CPU has been in use. Lines 6 onwards are the results of 'cat/ proc/stat/PID' where PID is the ID of each process contained in the /proc folder, that is, every single process running on that client machine. The relevant pieces for these process lines in the log file for energy data process consumption are utime (total time scheduled in user mode) and stime (total time scheduled in kernel mode), both of which are measured in jiffies.

This log is parsed at the server side to obtain these values of interest as well as the CPU data from Line 5, and the data are stored in a local database awaiting

```
Line 1: {"type":"Alpha",
Line 2: "benchmark": X, Y
Line 3: "time":"2012-07-23T22:19:29.305-07:00",
Line 4: "data":[
Line 5:        "cpu  14692 604 3837 9582 12448 2 117 0 0 0",
Line 6:        "1 (init) S 0 1 1 0 -1 4202752 8246 230296 22 373 61 82 604 404 20 0 1 0 2 3600384 ...
Line 7:        "10 (ksoftirqd/1) S 2 0 0 0 -1 2216722496 0 0 0 0 3 0 0 20 0 1 0 6 0 0 4294967295 0 ...
Line 8:        "11 (kworker/0:1) S 2 0 0 0 -1 2216722528 0 0 0 0 0 15 0 0 20 0 1 0 6 0 0 4294967295 ...
Line 9:        "1188 (upowerd) S 1 405 405 0 -1 4202752 1800 3303 11 0 28 7 0 0 20 0 3 0 4439 29...
```

Fig. 2.7 The first nine lines of a log belonging to a test client device with the client ID 'Alpha'

manipulation. Using process parent ID values, individual processes can be grouped into their parents. Earlier, it was mentioned that each client sends a pair of data samples tagged as either a 'before' or 'after' file. This allows us to take the delta of the time spent scheduled in each of the modes in order to make an estimate of how much CPU time was consumed by the process and also by the device as a whole.

Now, the first value to calculate is overall CPU utilisation (CPU usage). The equation below calculates overall CPU utilisation where subscript 'a' and 'b' denote values from the 'after' and 'before' samples, respectively:

$$cpuUsage = 100 \times \frac{utime_a + stime_a + nicetime_a}{totaltime_a} - \frac{utime_b + stime_b + nicetime_b}{totaltime_b} \qquad (2.1)$$

Next, the $proc_{cpu}$ value or the percentage of the CPU utilised by each process is calculated. The utime and stime values are the numbers from each process' corresponding line. The total time values are the CPU's total time values. The formula below would be used to calculate the CPU utilisation per process:

$$proc_{cpu} = 100 \times \frac{utime_a + stime_a}{totaltime_a} - \frac{utime_b + stime_b}{totaltime_b} \qquad (2.2)$$

In order to then get the overall power consumption of the device, we take into account the power benchmark values (line 2, Fig. 2.7). The two values X and Y in that line are P_{max} and P_{idle}, respectively, where P_{max} was the power used during the CPU stress test and P_{idle} was the power used during idle time. Blackburn describes a power estimation formula that can be used to convert the CPU usage values into power usage values based on these power benchmarks as shown in the formula below [11]:

$$totalPower = (P_{max} - P_{idle})\frac{cpuUsage}{100} + 100 \qquad (2.3)$$

To get the estimated power consumed by each process, we distribute the totalPower according to the portion of CPU utilised by each process shown below:

$$proc_{power} = proc_{cpu} \times totalPower \qquad (2.4)$$

The $proc_{power}$ and $totalPower$ values calculated are in watts (W). In order to get the energy consumed, it would be converted to kilowatt-hours (kWh) using the formula below, where Time would be the amount of time that elapsed between two sample points:

$$Energy = Power \times Time \qquad (2.5)$$

The final product is the energy footprint and the power consumed at specific times of every process on every logged device in the network. As Fig. 2.6 implied, these values can then be summed, and the footprint can cascade back up. This consolidated data is now stored server side by the aggregator, enabling access to process, application, device and network level data that can be leveraged by any visualisation mechanism created.

2.5.2 Visualising Energy Data

Visualisation tools specifically designed for network energy monitoring are currently lacking. The tools available are either generic visualisation packages that still require adaptation to the energy domain or highly specific energy monitoring tools that can only be applied to, for example, an entire home, with no potential for increased granularity. We therefore investigate what the most appropriate visualisation options would be for making energy data meaningful to the user.

The goals of visualising energy data collected from a device network would be:

1. To provide a live view of the energy consumption at the network, device, application and process level
2. To provide access to the history of the network energy consumption at each level
3. To inform the user of patterns in energy consumption in the network
4. To enable the identification of energy sinks through relevant visual cues

One of the biggest challenges faced in any kind of system is finding ways to turn raw input data into useful information. Dashboards are typically single-screen visual displays that aggregate a variety of information about a particular system. They have been used for displaying financial information, sensor details, medical information and so on. There is a growing interest in adopting the use of dashboards for displaying energy data of systems. In Few's book, *Information Dashboard Design*, he details how a dashboard can be appropriately used to display important information by taking into account historical context and visual design principles. He states, 'An effective dashboard is the product of ... informed design: more science than art, more simplicity than dazzle. It is, above all else, about communication' [25].

The network is continuously monitored, and energy data is logged temporally, requiring the visualisation of sets of discrete time series data, where the energy data observations are recorded at specific time intervals. Time series data highlights four data components: trends, cycles (describing regular fluctuations that are non-seasonal), seasons (variation that is directly affected by time of year) and irregularities (abnormalities that do not fit into any other component) [21]. These would be useful when dealing with energy data that is geared towards identifying energy

sinks as they can allow the user to then consider why the observed energy patterns are occurring. To this effect, any visual display located on the energy dashboard must enable the display of these kinds of information. This constraint trims the number of appropriate visualisations to the following:

1. *Basic graphs*: Dense scatter plots and line graphs are typically used to display time series data and are particularly effective when one has a small set of series on the chart. They are widely used in many tools, as has been shown in the previous sections. They also support a variety of annotations and so, when used in combination with each other, can result in a cluttered display.
2. *Sparklines*: Edward Tufte describes sparklines as 'small, high resolution graphics in a context full of words, numbers, images' [64]. They resemble miniature linegraphs or scatter plots that allow the display of graphs for a large number of series, where the focus is not on the individual values but on highlighting trends. An example of a successful use of sparklines for local system monitoring is the Ubuntu multiload indicator applet [32].
3. *Horizon graphs*: These can be described as a colour-coded collapsed line chart [26]. Time series data in a line graph can be converted into a horizon graph. The main goals of these graphs are to enable identification of anomalies and patterns, allow both independence of each series and comparison among series and ensure that the data presented preserve both history and accuracy. Work by Heer et al. [30] on comparing line graphs to horizon graphs found that users could estimate values on horizon graphs quicker and more accurately. They have been successfully used in the financial domain, particularly for visualising stock data. There is also growing interest in the software industry in both building visualisation tools that support horizon graphs and using them to monitor data centres.

2.5.3 Energy Dashboard Features

In order to enable a user of the dashboard to gain insight into the energy usage of the network at various levels, there must be components, interactions or features in place to support it. Following is a list of recommended features:

1. *Device management system*: By taking an object-oriented approach, devices in the system can be thought of as objects. Each device has a list of applications running on it, and each application has a corresponding process list. In order to support this view of the system, there should be a device management system in place. This would not only be useful for being able to interact with the data stored after consolidation from a variety of devices on the network but also to allow for administration of the system. It would also be important to ensure that raw data are accessible through the dashboard. This means that in addition to the visual displays there should be the ability to simply look at a list of processes, applications and devices. Overall the device management system would be reminiscent of any standard content management system, with the focus being

Fig. 2.8 Screenshot of the Nagios dashboard [23]

on ensuring that the energy data associated with each level of the network are clearly and easily accessible.

2. *Network map*: A map of the system is a common way of visualising a network to inform the user about how computers and servers are connected and routed through the system. Nagios is a widely used network monitoring tool used in SEEs (shown in Fig. 2.8). It supports network maps through a device discovery tool [35]. Thomas and Nejmeh state [61] that well-integrated SEE tools should blend with the existing SEE and provide similar information. Joulery should therefore be consistent with tools like Nagios. As the server would have a list of active clients in the network being monitored, a live energy map of the network could be produced. This map would allow for a high-level view of the system, to give the user a big picture of the size of the network as well as components within it.

3. *Visual cues*: The visual displays described in the previous section are also useful for enabling drill down capabilities in an energy dashboard. For example, having a speedometer-style display for each application on each device may be overkill; however, having a series of horizon charts or sparklines for each process may be more appropriate. In addition, the use of colour and size should be leveraged to aid in easy identification of high-energy use areas of the network.

Figures 2.9, 2.10 and 2.11 are design mock-ups that demonstrate how these features are to be put together to form a cohesive visualisation of the energy footprint of the network.
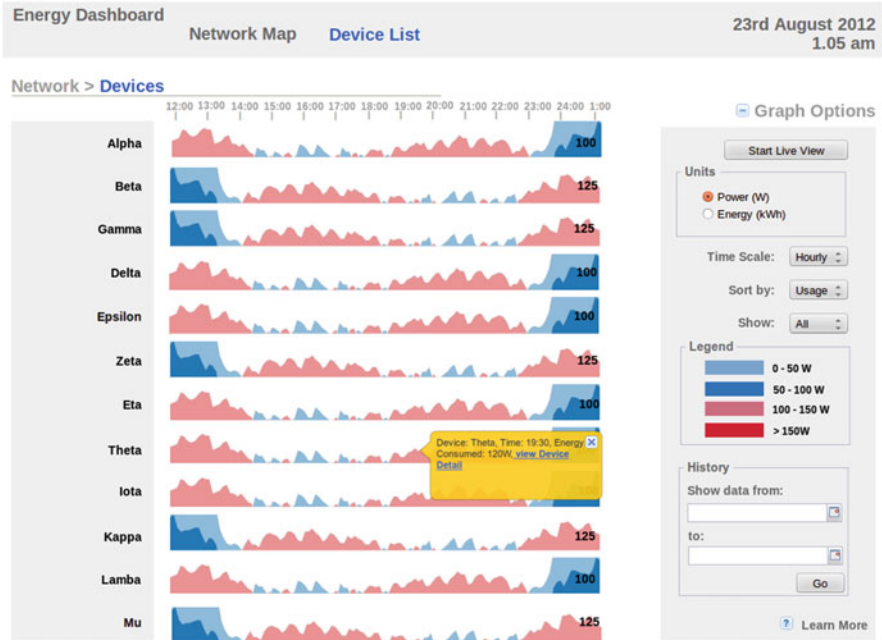
Fig. 2.9 Joulery dashboard: overview of all devices

2.5.4 Joulery in the Green SEE Landscape

This section has described Joulery, a tool for analysing the energy consumption of networks, via consolidating energy data, and a proposed visualisation for energy data. The data produced by the prototype consolidation tool provides granular information, from a network-wide footprint down to consumption per application on each device.

The energy footprint described here is similar to the IT equipment power component of the PUE calculation. To this effect, if we can calculate the overall power associated with the environment containing the network, then the same ratio can be obtained for a network. As the PUE is a widely recognised and adopted metric [55], this could be a means to convert the energy consumption value into an energy metric for networks in SEEs. We demonstrate how a network energy awareness tool can be developed to provide granular energy data that supports an existing energy metric (PUE), be designed to support data collection from components of a network and present the information in a potentially meaningful manner to support making better energy decisions.

We briefly describe a set of motivating scenarios that demonstrate how a tool like Joulery can support the energy and information aspects at the infrastructure, hardware and software layers and encourage green behaviour.

Fig. 2.10 Joulery dashboard: process level detail for device 'Alpha'

Fig. 2.11 Joulery dashboard: network map view

2.5.4.1 Nettie, the Network Administrator, and the Rogue Server

Nettie, the network administrator for a small software start-up, uses Joulery to monitor servers in their data centre. By looking at a network map like that in Fig. 2.11, she is able to see that the device Theta is consuming more energy than average. Theta is supposed to be a backup server that only kicks into high-performance mode once a day when i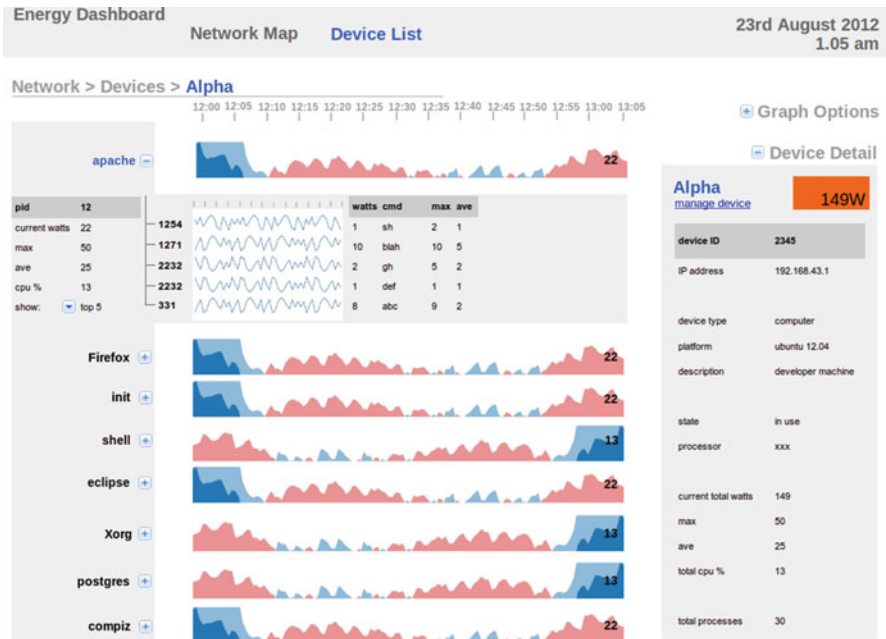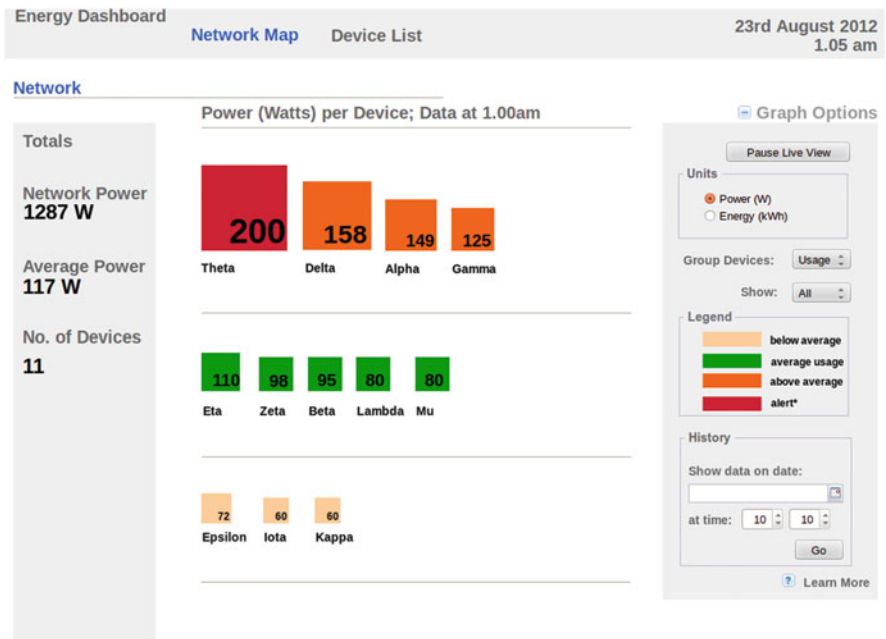t is backing up data. Nettie can then use the drill down mechanisms in Joulery to view detailed process level energy consumption data (like that in Fig. 2.10) to identify a rogue process that is the cause of the anomaly. Joulery enabled the Nettie, the network administrator, to fix an energy-related issue in the SEE, thereby reducing unnecessary consumption.

2.5.4.2 Dev, the Developer, and the Continuous Integration Conundrum

A midsize software engineering firm focuses on building control systems in a very systematic fashion. There is a team of developers who are responsible for writing their allocated pieces of functionality and pushing their code to the repository. The current system is based around the practice of continuous integration (all developer code is merged at the repository several times a day). Dev, a curious developer, has been wondering how energy intensive each build-test-deploy cycle is and if it really is productive and energy efficient to be doing this multiple times a day. In order to figure this out, Dev convinces the developers to install the Joulery logger on their development devices and also installs a logger on the main servers responsible for continuous integration. Dev needs data over a few weeks to be able to assess how their integration habits are affecting the overall energy consumption of their SEE. Dev uses the device list view (like that in Fig. 2.9) to view the energy trends in their SEE and notices that because of the way their system is set up, each integration causes an above-average (for their SEE) level of energy consumption.

He then begins a more detailed investigation into what the right trade-off is between making sure that their mainline repository is clean and that their energy consumption is not too high. Joulery enabled Dev, the developer, to assess the impact of an implementation practice that is a norm for certain software engineering companies. He can use his findings to influence their continuous integration protocols to be less energy intensive.

2.5.4.3 Archie, the Architect, and the Design Problem

Archie is a software architect for a large software engineering company. The requirements engineer has described the client's needs for a data sharing system that will be used internally in the client's organisation. The client is a mining company that operates in remote areas, where power and utility outages are

frequent. With this in mind, Archie realises that the software will therefore need to consume as few resources as possible. As Archie designs the software, he needs a method for estimating the overall energy footprint of the system and how it responds to power interruptions. To do this, Archie fires up a virtual network of machines and creates a virtual energy footprint of different design options. He uses the Joulery tool on these virtual machines (viewing Fig. 2.9) and runs a set of experiments to determine which of his software designs are estimated to be most robust, energy-efficient and conducive to the client's needs. Once he settles on the optimal design, Archie sends this off for approval before implementation can begin. Joulery enabled Archie, the software architect, to select from a variety of designs to determine which was least energy intensive.

2.6   Conclusions

To restate the challenge for green software engineers based on Brooks' challenges for computer science, how do we build tools that provide software engineers with useful and meaningful data that can be visualised for a greater understanding of potential environmental considerations of software engineering?

In this chapter, we have described how the software engineering environment can be enhanced with respect to the matter involved, information available and energy consumed within it. We surveyed existing solutions at the infrastructure, hardware and software layers and described how this can influence green behaviour and instantiated these ideas through the example of Joulery, an energy awareness tool.

Networks are a core component of software engineering environment. These kinds of networks are common in industrial civilisations; they make up the informational backbones of corporations, universities, local governments and many other mid-level institutions. Performing analyses of energy usage at this level allows for greater leverage on system-wide consumption than device-level analysis while being more tractable and actionable than broader-scale assessments. The Joulery tool aims not only to optimise the energetic footprint of the hardware and infrastructure in the SEE but is an addition to the suite of SEE software that is geared towards enabling software engineers to exhibit environment-friendly behaviours. We hope that by providing a tool to support this level of analysis, we may contribute to the broader goal of supporting the transition to sustainability across many different sectors within the industrialised world. While the relationship between energy efficiency and sustainability is not always clear [63] and while sustainability at very least entails a broader set of responses than just efforts to increase energy efficiency, we believe greater knowledge about energy usage is nevertheless a critical piece of the sustainability puzzle.

The engineering of software involves a variety of levels of abstraction that express the system to be built from a set of specifications to implemented code. The various phases allow for different ways of influencing the end system goals. Due to this influence that software engineers have, we can consider how to imbue

each phase with motivations regarding environmental sustainability. With the increased pace of innovation in our society, software engineering faces shorter and faster cycles. This speed of development means that the goals are mostly short term. Energy issues are considered because they are often tied to direct financial issues as well. As a long-term issue, therefore, other environmental aspects tend to get left out of the picture. There is still much that can be done to support environmental (including energy) issues that exist within software engineering environments. We would therefore like to conclude with a set of open questions that still need to be addressed with respect to green software engineering environments:

1. How can we track and assess the environmental impact of resource consumption during different activities in the SEE? Is it possible to do this at each layer of the SEE and can we support cross-sector resource flow tracking?
2. How can we calculate the carbon footprint of different kinds of software engineering activities? Is this a useful metric for analysing the SEE?
3. As software engineering evolves and we move between treating software as a product and software as a service, are there different environmental considerations that can and should be made within the SEE?
4. Is using cloud-based solutions simply offloading the environmental responsibility onto other organisations?
5. How can we assess the effectiveness of this green introspection that is occurring within the SEE? Are current methods, metrics and tools appropriate?
6. What does it mean to have a truly Green SEE? What kinds of standards or requirements do we need to set?

References

1. AEO2012 early release overview, Technical report. U.S. Energy Information Administration. http://www.eia.gov/forecasts/aeo/er/pdf/0383er%282012%29.pdf
2. Agency USEP (2003) Energy star the power to protect the environment through energy efficiency. Tech. rep.
3. Alcott B (2005) Jevons' paradox. Ecol Econ 54(1):9–21
4. Amsel N, Ibrahim Z, Malik A, Tomlinson B (2011) Toward sustainable software engineering: NIER track. In: 2011 33rd international conference on software engineering (ICSE). IEEE, pp 976–979
5. Amsel N, Tomlinson B (2010) Green tracker: a tool for estimating the energy consumption of software. In: CHI'10 extended abstracts on human factors in computing systems. ACM, pp 3337–3342
6. Baer WS, Hassell S, Vollaard BA (2002) Electricity requirements for a digital society. RAND, Santa Monica, CA
7. Barroso LA, Hölzle U (2009) The datacenter as a computer: an introduction to the design of warehouse-scale machines. Synth Lect Comput Architect 4(1):1–108

8. Benini L, Bogliolo A, De Micheli G (2000) A survey of design techniques for system-level dynamic power management. IEEE Trans Very Large Scale Integrat (VLSI) Syst 8(3):299–316

9. Berl A, Gelenbe E, Di Girolamo M, Giuliani G, De Meer H, Dang MQ, Pentikousis K (2010) Energy-efficient cloud computing. Comput J 53(7):1045–1051

10. Birol F (2010) World energy outlook. International Energy Agency

11. Blackburn M, Grid G (2008) Five ways to reduce data center server power consumption. Green grid

12. Boulding KE (1996) The economics of the coming spaceship earth. Environ Qual Growing Econ 2:3–14

13. Brooks FP Jr (2003) Three great challenges for half-century-old computer science. J ACM 50 (1):25–26. doi:10.1145/602382.602397, URL http://doi.acm.org/10.1145/602382.602397

14. Capra E, Formenti G, Francalanci C, Gallazzi S (2010) The impact of MIS software on it energy consumption. In: European conference of information science 2010

15. Capra E, Francalanci C, Slaughter SA (2012) Is software "green"? Application development environments and energy efficiency in open source applications. Inform Software Tech 54 (1):60–71. doi:10.1016/j.infsof.2011.07.005, URL http://dx.doi.org/10.1016/j.infsof.2011.07.005

16. Christensen K, Nordman B, Brown R (2004) Power management in networked devices. Computer 37(8):91–93

17. Commission CE (2000) Energy accounting: a key tool in managing energy costs many costs. Tech. rep., California Energy Commission

18. Corporation I. Powertop. URL https://01.org/powertop

19. Corporation O. The Java monitoring and management console (jconsole). URL http://openjdk.java.net/tools/svc/jconsole/

20. Doyle MW (2005) Three pillars of the liberal peace. Am Polit Sci Rev 99(3):463–466

21. Easton VJ, McColl JM. Statistics glossary. URL http://www.stats.gla.ac.uk/steps/glossary/timeseries.html

22. Elliot S, Binney D (2008) Environmentally sustainable ICT: developing corporate capabilities and an industry-relevant is research agenda. In: Pacific Asia conference on information systems, Suzhou, China, 4–7 July 2008

23. Enterprises N. Nagios xi screenshots. URL http://www.nagios.com/products/nagiosxi/screenshots

24. Fan X, Weber WD, Barroso LA (2007) Power provisioning for a warehouse-sized computer. ACM SIGARCH Comput Architect News 35(2):13–23

25. Few S (2006) Information dashboard design. O'Reilly, Sebastopol, CA

26. Few S (2008) Time on the horizon. Vis Bus Intell Newslett 1–7

27. Flinn J, Satyanarayanan M (2009) Powerscope: a tool for profiling the energy usage of mobile applications. In: Proceedings. WMCSA'99. Second IEEE workshop on mobile computing systems and applications, 1999. IEEE, pp 2–10

28. Garrett M (2007) Powering down. Queue 5(7):16–21

29. Google: going green at Google – clean energy initiatives. URL http://www.google.com/about/datacenters/index.html

30. Heer J, Kong N, Agrawala M (2009) Sizing the horizon: the effects of chart size and layering on the graphical perception of time series visualizations. In: Proceedings of the SIGCHI conference on human factors in computing systems. ACM, pp 1303–1312

31. Hewlett-Packard, Intel, Microsoft, Phoenix, Toshiba. Advanced configuration and power interface specification. URL http://www.acpi.info

32. Hofmann M. System load indicator. URL https://launchpad.net/indicator-multiload

33. Humble J. javasysmon. URL https://github.com/jezhumble/javasysmon

34. IEEE Std 610.12 (1990) IEEE standard glossary of software engineering terminology

35. Josephsen D (2007) Building a monitoring infrastructure with Nagios. Prentice Hall PTR, Upper Saddle River, NJ

36. Jovanovic B, Rousseau PL (2005) General purpose technologies. Handbook Econ Growth 1:1181–1224
37. Kansal A, Zhao F, Liu J, Kothari N, Bhattacharya AA (2010) Virtual machine power metering and provisioning. In: Proceedings of the 1st ACM symposium on cloud computing. ACM, pp 39–50
38. Kohlbrecher J, Hakobyan S, Pickert J, Grossmann U (2011) Visualizing energy information on mobile devices. In: 2011 IEEE 6th international conference on intelligent data acquisition and advanced computing systems (ID-AACS), vol 2. IEEE, pp 817–822
39. Laitner JAS, Nadel S, Elliott RN, Sachs H, Khan AS (2012) The long-term energy efficiency potential: what the evidence suggests. Tech. rep., American Council for Energy Efficiency
40. LeFebvre W. Unixtop. URL http://sourceforge.net/projects/unixtop/
41. Leubner   A.   Usage   timelines.   URL   http://www.refined-apps.com/trials/ UsageTimelinesProrelease.apk
42. Liu C, Qin X, Kulkarni S, Wang C, Li S, Manzanares A, Baskiyar S (2008) Distributed energy-efficient scheduling for data-intensive applications with deadline constraints on data grids. In: IEEE international performance, computing and communications conference, IPCCC 2008. IEEE, pp 26–33
43. Lockton D, Harrison D, Stanton NA (2012) Models of the user: designers' perspectives on influencing sustainable behaviour. J Des Res 10(1):7–27
44. Lovins AB (2011) Re: The efficiency dilemma. The Mail, The New Yorker
45. Mahesri A, Vardhan V (2005) Power consumption breakdown on a modern laptop. In: Power-aware computer systems. Springer, pp 165–180
46. Mersenne Research I. The great Internet Mersenne prime search. URL http://www.mersenne. org/freesoft/
47. Mingay S (2007) Green IT: the new industry shock wave. Gartner RAS Research Note G 153703
48. Mitchell-Jackson JD (2001) Energy needs in an internet economy: a closer look at data centers. Ph.D. thesis, University of California
49. Mittal R, Kansal A, Chandra R (2012) Empowering developers to estimate app energy consumption. In: Proceedings of the 18th annual international conference on mobile computing and networking. ACM, pp 317–328
50. Murugesan S (2008) Harnessing Green IT: principles and practices. IT Prof 10(1):24–33
51. Naumann S, Dick M, Kern E, Johann T (2011) The GREENSOFT model: a reference model for green and sustainable software and its engineering. Sustain Comput Informat Syst 1 (4):294–304, http://dx.doi.org/10.1016/j.suscom.2011.06.004. URL http://www. sciencedirect.com/science/article/pii/S2210537911000473
52. Odessa: sample computer network diagrams. URL http://www.conceptdraw.com/samples/ network-diagram
53. Owen D (2010) The efficiency dilemma. Annals of environmentalism, The New Yorker
54. Penzenstadler B, Raturi A, Richardson D, Tomlinson B (2014) Safety, security, now sustainability: the non-functional requirement for the 21st century. In: Software (IEEE) Vol. 31 (3)
55. Rawson A, Pfleuger J, Cader T (2008) Green grid data center power efficiency metrics: PUE and DCIE. The green grid white paper 6
56. Research M. Joulemeter: Computational energy measurement and optimization. URL http:// research.microsoft.com/en-us/projects/joulemeter/default.aspx
57. Rosseto EP (2011) Study of the correlation between software developer profile and code efficiency. Ph.D. thesis, Politecnico di Milano, Milan
58. Sanchez MC, Brown RE, Webber C, Homan GK (2008) Savings estimates for the United States Environmental Protection Agency's energy star voluntary product labeling program. Energy Policy 36(6):2098–2108
59. Shah A, Christian T, Patel CD, Bash C, Sharma RK (2009) Assessing ICT's environmental impact. IEEE Comput 42(7):91–93

60. Smith JW (2010) Green cloud a literature review of energy-aware computing. Ph.D. thesis, University of St. Andrews, Fife
61. Thomas I, Nejmeh BA (1992) Definitions of tool integration for environments. IEEE Software 9(2):29–35, http://doi.ieeecomputersociety.org/10.1109/52.120599
62. Tomlinson B (2010) Greening through IT. MIT Press, Cambridge, MA
63. Tomlinson B, Silberman MS, White J (2011) Can more efficient it be worse for the environment? Computer 44(1):87–89
64. Tufte ER (2006) Beautiful evidence, vol 23. Graphics Press, Cheshire, CT
65. Webb M et al (2008) Smart 2020. Enabling the low carbon economy in the information age. The Climate Group. London 1(1), 1–1

Chapter 3
Processes for Green and Sustainable Software Engineering

Eva Kern, Stefan Naumann, and Markus Dick

3.1 Introduction

Within research in the context of 'Green in IT' (ways to make ICT itself greener) and 'Green by IT' (possibilities to encourage environmental-friendly movements by ICT), it turns out that, next to the hardware aspects, the software side also gains importance. Software causes hardware activity and is responsible for energy consumption in that way. It has become an important aspect of daily life, and most people cannot imagine future development without software.

Since one big objective of Green IT activities is to find solutions to solve the problem of increasing energy consumption, energy-efficient software is required. In order to develop such software, developers need to know how this can be achieved and to have suitable tools to reach this goal. Above all, the software engineering process, regarding questions of sustainability, needs to be clarified.

We will present a description of a whole life cycle of software products from a life cycle thinking point of view and a process model for green software engineering, based on a definition for green software engineering in the following section. We will then go into the different phases of software development processes and integrate the aspects in existing process models.

E. Kern (✉)
Institute for Software Systems, Trier University of Applied Sciences, Leuphana University of Lüneburg, Germany
e-mail: mail@nachhaltige-medien.de

S. Naumann • M. Dick
Institute for Software Systems, Trier University of Applied Sciences, Trier, Germany
e-mail: s.naumann@umwelt-campues.de; sustainablesoftwareblog@gmail.com

3.2   Related Work

Many contributions on software engineering exist (e.g. [36, 41]); therefore, we focus on those articles which cover green and sustainable software engineering.

A good overview can be found in [35]: here, the author found 96 relevant publications on sustainable software engineering. Penzenstadler [32] discusses the question what sustainability means in (and for) software engineering. Penzenstadler et al. [33] ask who the stakeholders are in a sustainable engineering process. Amsel et al. [4] state that sustainable software engineering should develop a software that meets the needs of users while reducing environmental impacts.

A methodology to measure and incrementally improve the sustainability of software projects is presented by Albertao [2]. This methodology advocates the implementation of sustainable aspects continuously, divided into the following phases: assessment phase, reflection phase, and goal improvement phase. In order to make the different sustainability issues manageable, properties of a quality model that is further developed in a later work are referred to, considering the overall software process [3]. In view of the fact that the common assumption states that software is in general 'environment friendly', metrics that can be assessed in a real software project are presented. This approach shows that it is feasible to continuously improve software projects regarding sustainability issues by measuring a set of metrics repeatedly over several iterations.

Calero et al. [8] take a deeper look at the quality standards of software and how they can reinforce sustainability.

Agarwal et al. [1] take a look at the definition of green and sustainable software engineering (see Sect. 3.3 and [31]) and discuss the possibilities and benefits of green software. They claim that more efficient algorithms will take less time to execute, which, as a result, will support sustainability. Additionally, they present methods to develop software in a sustainable way, compared to conventional methods, and list some more environment-friendly best practices in the development and implementation of software systems.

Based on the life cycle of software, Taina proposes metrics [43] and a method to calculate the carbon footprint of software [42]. To do so, he analysed the impacts of each software development phase for a generic project. The resulting carbon footprint is mainly influenced by the development phase but also by the way it is delivered and how it will be used by the customers. The main problem regarding the calculation is that detailed data are required, which is often not available. Lami et al. [28] define software sustainability from a process-centric perspective and define processes so that they can be measured in terms of process capability according to the ISO/IEC 15504 standard. They distinguish between the sustainability factors power, paper and fuel consumption, especially.

Inspired by the GREENSOFT model, Mahmoud and Ahmad [30] present a complete software development life cycle (SDLC) for ecologically sound software engineering, called the *green and sustainable software life cycle*. This is a two-level

model that consists in the first level of a software engineering process and in the second level of a categorisation of tools that can help to monitor and to identify how software causes energy and resource consumption.

The proposed software engineering process incorporates ideas from sequential, iterative and agile processes. It is designed to reduce its own negative impacts on the environment as well as the expected impacts of the software product. To make each phase of the SDLC environmentally sustainable, there are green processes, green guidelines and metrics that indicate the greenness of a phase. The process phases are requirements, design, unit testing, implementation, system testing, green analysis, usage, maintenance and disposal. According to Mahmoud and Ahmad, an increment milestone can be placed between implementation and system testing. The release milestone is located between green analysis and usage. It is not possible to go back to the previous phase during process execution. Instead, from unit testing, system testing, green analysis and maintenance, one has to return to requirements. An important phase regarding the environmental optimisation of the software product is the green analysis phase [30]. Its purpose is to promote energy efficiency and to determine the greenness of the software artefacts of each increment. Appropriate tools and metrics help to identify problems in requirements, design or implementation, which make it necessary to step back to requirements phase. The disadvantage of this valuable approach is obviously that it is complete, which means that developers have to drop their well-understood and well-mastered software process to learn a new one that comes with very limited documentation and professional support.

Another approach of a green software development is presented by Shenoy et al. [39]. They take the whole life cycle of software into account and also give suggestions for the typical development phases of software. Käfer [22] also presents conceptual and architectural issues concerning software energy consumption and ideas for incorporating energy efficiency issues into software projects. Mahaux [29], Penzenstadler [34] and Kocak [26] especially consider requirements engineering. Sahin [38] and Shojaee [40] look at the design of energy-efficient software while focusing on software engineering in the Web. Dick et al. [14] and Bordage et al. [6] describe some procedures and rules to reduce the energy consumption of websites.

The different aspects representing the dimensions of green software engineering are covered in the GREENSOFT model [31], which is described in Sect. 3.4. The GREENSOFT model is a conceptual reference model that supports IT professionals and software users in the sustainable development and usage of software. It consists of four parts: a life cycle of software products, criteria and metrics, procedure models and tools. The procedure models examine the development, purchasing, administration and usage of software.

3.3   Definitions

In times of climate change, the topic of sustainability takes on an important role even in the area of ICT. Besides the efforts of improving the hardware, the software aspect should get attention as well [24]. Derived from the basic requirements on green and sustainable software, the following two definitions can be given.

> **Definition 1 'Green and Sustainable Software'** [31]
>
> [Green and Sustainable Software] is software, whose direct and indirect negative impacts on economy, society, human beings, and environment that result from development, deployment, and usage of the software are minimal and/or which has a positive effect on sustainable development.

A green and sustainable software product can only be achieved if the person in charge of the requirements and the organisation which are involved in the software development is aware of the effects of sustainable development that emanate from the product. In order to enable different stakeholders to include those effects within their decision-making process, it is necessary to institutionalise the evaluation of this software. To make sustainability aspects manageable, measures in specified requirements analysis and the software development process must be integrated. In that way, requester, software architects and software developers will be able to optimise the software product. The precondition for that is a sustainable development process. This means the consideration of environmental impacts during the software life cycle and the pursuit of the goals of sustainable development. From this and definition 1 follows the definition for green and sustainable software engineering:

> **Definition 2 'Green and Sustainable Software Engineering'** [31]
>
> Green and Sustainable Software Engineering is the art of developing green and sustainable software with a green and sustainable software engineering process. Therefore, it is the art of defining and developing software products in a way, so that the negative and positive impacts on sustainable development that result and/or are expected to result from the software product over its whole life cycle are continuously assessed, documented, and used for a further optimization of the software product.

The definitions for green and sustainable software and for green and sustainable software engineering are based on product life cycles in terms of life cycle assessment (LCA) or a cradle-to-grave approach. These are the findings on the effect of levels in the ICT to sustainable development and the impact of ICT on the life cycles of other products and services.

Overall, a green software product should be green and sustainable itself. This means that the negative environmental, social and economic effects arising from the software product over its entire life cycle should be as small as possible. Most obvious in this regard are the first-order effects, the so-called effects of supply, such as performance requirements, network bandwidth, hardware requirements or the product packaging, which lead to a higher or lower demand for natural resources and energy.

The second-order effects (usage effects) result from the use of ICT in the life cycles of other services and products. Information-technological components whose essential function is to implement services can be found in almost every product of daily life. In this respect, software plays a crucial role in the life cycles of products and services: product design, production process, waste disposal and the usage of other products can be optimised with the software. However, it must be noted that the second-order effects are more difficult to estimate than the first-order effects.

It is even more difficult to assess the third-order effects, the so-called systemic effects, due to the many systemic interdependencies and demands of experience. An example of this is the so-called rebound effect that can occur when resources are freed through optimisation, which will be eventually more than overcompensated by increased business activity of free resources, and the original scale may be altered. The activities of Green IT are primarily the first-order effects, that is, the presented effects of supply. In contrast, the second- and third-order effects, which are usage effects and systemic effects, are associated with *Green by IT*.

From these considerations, it appears that the conservation of resources and energy by optimising the ICT can only be one partial aspect. Another important aspect is the conservation of resources and energy, which can be achieved by the usage of ICT in other segments of products and services. Seen from a broader point of view, the question arises how negative effects on ecology, society and economy can be reduced and how positive effects can be enhanced.

The problem here is that there is software that directly supports the objectives of sustainable development, because it is their specific purpose; examples include software which enables smart heating, smart logistics or paperless offices (e.g. see [18]). In these cases, it is relatively easy to assess the usage effects of an information technology system. For standard software that can be used in diverse industries, the assessment of both the usage effects and the systemic effects is much more costly or even impossible (e.g. office suites). Since these kinds of software are used in many different contexts, various usage scenarios exist, and one needs to make huge efforts to assess these. Therefore, a green and sustainable software product itself should have little impact on sustainable development and, if it is its specific benefit, promote the pursuit of sustainable objectives.

3.4   The GREENSOFT Model

The GREENSOFT model (Fig. 3.1) includes a holistic life cycle model for software products, sustainability criteria and metrics for software products, process models for various stakeholders as well as recommendations and tools supporting stakeholders while developing, procuring, maintaining and using software products in a way compatible with the objectives of sustainable development.
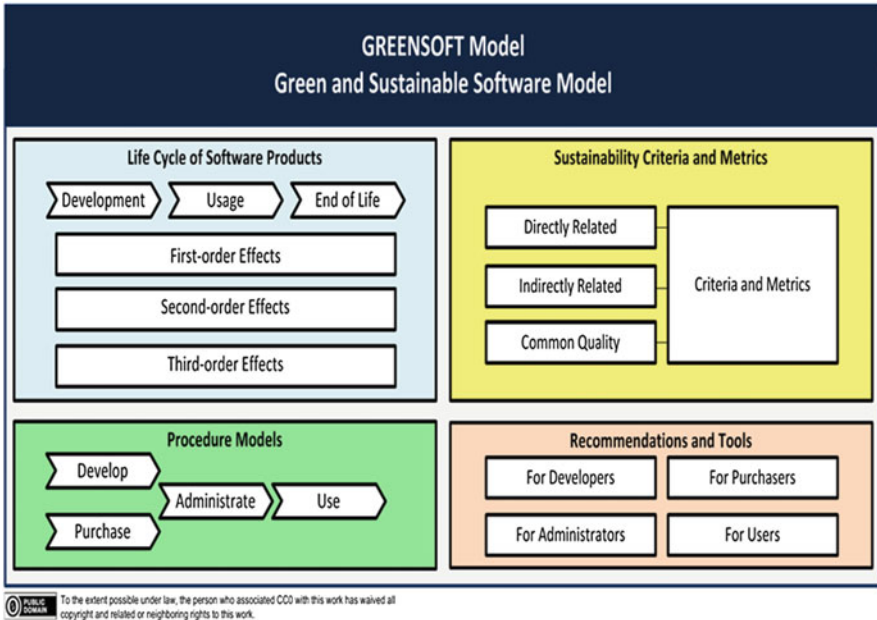
Fig. 3.1 The GREENSOFT reference model [37] (cf. [31])

The life cycle for software products is based on life cycle thinking (LCT), which means in fact that it is not a software development life cycle (SDLC). LCT observes the principle 'from cradle to grave' or 'from cradle to cradle' [7, 44]. Its aim is to assess the economic, human, social and ecological compatibility of a product over its entire life cycle. A typical life cycle for material products begins with raw material procurement and ends with recycling or disposal, for example, raw material extraction, material processing, manufacturing, packaging, distribution/transport, usage and recycling/disposal. Thus, such a life cycle does not describe engineering processes or workflows that are necessary to develop or produce a product. Regarding, for example, environmental effects, the life cycle models all extractions from the environment (e.g. ores, fossil resources or fuels, water) and all emissions to the environment (e.g. heavy metals, CO_2, CH_4, radiating particles) that occur in each life cycle phase. This balance and its assessment can then be used to optimise the product itself (e.g. raw material mix, expected lifetime, energy consumption and emissions during usage) and also its production, distribution or recycling/disposal processes, or it can just be used to compare the product with competing products [44]. The life cycle for software products presented herein is an adaption of life cycle models for material products to software.

The model part of sustainability criteria and metrics covers general metrics and criteria for the measurement of software quality [20] and allows the classification of criteria and metrics for assessing the effects of a software product on sustainable

development. Criteria and metrics that meet the requirements include models for the measurement of software quality in terms of product and process quality, as well as methods that are borrowed from life cycle analysis. Key differentiators are the immediate criteria and metrics that are based on the first-order effects or effects of supply and the indirect criteria and metrics that relate to the second- and third-order effects or usage effects and systemic effects. Some basic criteria and metrics are presented that can be used to evaluate the effects and impacts of a software product on sustainable development. Some more criteria and metrics are proposed in [9, 3, 21, 43]. In order to sum up these criteria, Kern et al. [25] introduced a quality model for green and sustainable software.

The model part enables the classification of process models that support the procurement, development, administration and usage of software in the context of sustainable development. As an example, a generic extension for software development processes has been proposed, which aims at a systematic consideration of sustainability aspects in software projects [13]. The extension is described in detail in Sect. 3.6.

The last part of the model shows recommendations for action and tools. These provide support to the stakeholders in the application of techniques to promote the sustainable objectives. The different levels of knowledge and experience of the actors should be taken into consideration. Possible actors in this area are software developers, buyers, administrators as well as professionals and private users but also all those involved in software products in general [14, 17].

3.5 Life Cycle of Software Products

In order to assess the ecological, social, human and economic impacts during software development, one should consider the entire life cycle of software products from the start. LCT considers more than the production process of a product. As seen from the perspective of an entire life cycle, one covers the process beginning with the production of raw materials, over the production itself and the usage, up to the deactivation and the disposal of the specific product. With the help of this knowledge, it is possible to optimise the product in a balanced way [44].

Oriented towards the LCA as an international standardised method [12], we propose the life cycle for software products depicted in Fig. 3.2. This life cycle is aimed at considering and classifying sustainability aspects during the development of software products (cf. [11, 19]). It is based on the LCA model of material products and shows exemplary impacts on sustainable development of usually immaterial software products.

In the following, we will especially describe the direct effects (so-called first-order effects) during the life cycle of a software product. These can be easily generalised to other software products.

	Development		**Usage**	**End of Life**	
	Development	Distribution	Usage	Deactivation	Disposal
First-order Effects	- Business trips - Office HVAC - Energy for ICT - Office lighting - Working Conditions - ...	- Packaging - Data medium - Manuals - Transportation - Download size - ...	- Software induced energy consumption - Software induced resource consumption - Hardware requirements - Accessibility - ...	- Backup size - Long term storage of data (due to legal issues) - Data conversion (for future use) - ...	- Packaging - Data medium - Manuals - ...
Second-order Effects	- Telework - Globally distributed development - Higher motivation of team members - ...		- Dematerialization - Smart logistics - Smart metering - Smart buildings - Smart grids - ...	- Media disruptions - ...	
Third-order Effects	- Changes in software development methods - Changes in corporate organizations - Changes in life style - ...		- Changes of business processes - Rebound effects - ...	- Demand for new software products - ...	

Fig. 3.2 Life cycle for software products, oriented towards life cycle thinking and giving some exemplary effects [37] (cf. [31])

3.5.1 Development Phase

In the development phase of *a software* product, impacts of the development process of the software engineering itself as well as impacts of shared functionalities of different departments of the development enterprise are taken into account. This includes departments like accounting, human resource management, marketing, product packaging, etc. Here, especially the energy consumed by workstations of the software developers and the IT infrastructure, for example, for network and servers as well as energy for lighting, heating and air conditioning of the buildings, needs to be considered.

Above that, there is the energy for the personal transport and needed resources. One big point is business trips like meetings with the development team or the customer and the daily trips to work of the employees. Some of these trips, and thus their impacts, could be reduced by telework or video conferences [10]. This means in general that the consequences caused by material products can be relieved by immaterial products (the second-order effects). In this case, the material product is commuting, whereas the immaterial product is telecommuting. Indeed, this might result in the third-order effects if there are changes in organisations, software development methods or general life design.

Apart from the described ecological and economically centred effects, there are also social and human aspects, respectively: regarding social acceptability, one has to look at the specific working conditions, the payment as well as the social insurance of so-called offshore workers.

Another aspect going into the development phase is the impact of maintenance, which means bug fixing of the software product. This kind of work can be seen as a part of software development.

3.5.2 Distribution Phase and Disposal Phase

The effects caused by the distribution and, further, the disposal of a software product are, for example:

- Printed or digital manuals and their used resources
- Means of transport
- Kind and design of product packaging
- Download size and IT infrastructure

All of these belong to the distribution phase of the life cycle of a software product, whereas everything that can be seen as waste management and recycling of the material parts of the software product goes to the disposal phase.

3.5.3 Usage Phase

The usage of software products especially leads to direct effects: executing software consumes computing time and, thus, energy. Additionally, some kinds of software need network bandwidth and extra computing resources (e.g. ERP systems). Hence, the energy consumption goes up.

Another point is updates: depending on the size and frequency, updates need different sizes of memory space. Updates also influence the capacities that are necessary for data backups. Moreover they need to be transported to the user, which results in additional data traffic.

In general, the effects of the usage of a software product can be positive as well as negative. The positive effects are, for example, the reduction of energy and resource consumption by using ICT in order to optimise processes. These effects that result indirectly from using ICT are classified among the second-order effects and are called dematerialisation effects. Another example is the substitution effect, which means that resources are conserved by replacing material products by their immaterial counterparts. The negative effects are, for example, induction and rebound effects that have been already mentioned before.

In the early stages of software development, one can only make a rough estimate of those effects, especially if the field of application of the software product is widespread, that is, standard software, for example, office suite.

New software products usually need more powerful hardware than similar older products. Therefore, the existing hardware will be replaced by newer hardware if there is a new software product or a new version released. This is the case in

enterprises as well as at home. Although newer hardware is normally more energy efficient, it causes negative ecological effects, since the old hardware needs to be disposed of. This and also the production of new products need energy and natural resources (cf. [19]).

Next to these ecological effects, there are systemic effects because the mining of ores in developing countries for the production of hardware leads to ecological, social and economic imbalances (cf. [5]). Some of the hardware that is decommissioned is also brought to these countries. If the material is recycled and disposed of in uncontrolled ways, this compounds the situation by straining the environment and health of the affected people [19].

In summary, the usage phase considers the effects of supply, systemic effects and usage effects.

3.5.4 Deactivation Phase

The last phase (see Fig. 3.2) considers the impacts caused by the deactivation of a software product. In many cases, deactivation of one product means the implementation of another one. If so, it might be necessary to convert existing files into a new format or to archive them. This might lead to occasional costs and other disadvantages. Moreover, one has to take into account the required memory usage for backups of the existing files.

3.6 A Generic Process Model for Green and Sustainable Software Engineering

Our proposal of a green and sustainable software development process does not implement a full SDLC. It rather describes an add-on that can be mixed with any implementation of an SDLC to raise the awareness of the participating stakeholders for an ecologically sound software design. Hence, we show later on how its procedures can be added with ease to Scrum as an agile SDLC, as well as to more formal Unified Process like OpenUP.

The main process tasks (Fig. 3.3) are sustainability review and preview, process assessment, interim sustainability presentations and sustainability presentation and report release. These tasks are accompanied by an auxiliary task, the sustainability presentation and report preparation, and two artefacts, the sustainability journal and the sustainability report.

Figure 3.3 depicts the SLDC as iteration based, which is only an example. Actually, the iterations may be replaced with phases or increments known from common SDLC models.
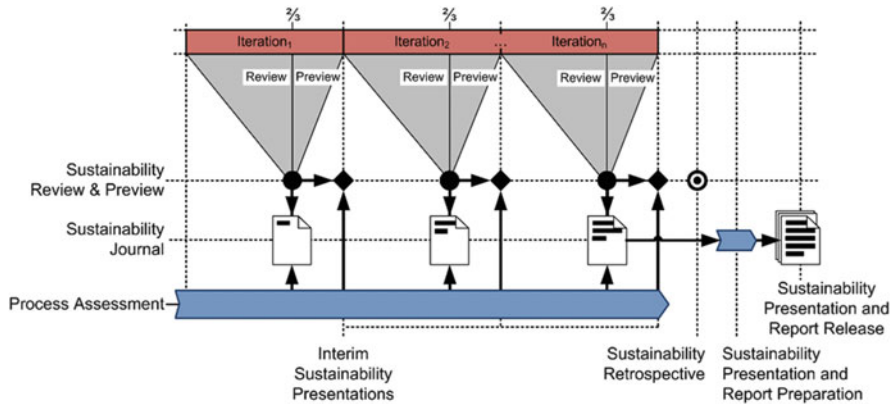
Fig. 3.3 Generic model for a green and sustainable software process [37]

The process assessment activity assesses the sustainability of the SDLC itself, whereas the reviews and previews focus on the sustainability of the software product under development. Both use the same journal to record their results for further use or reference. At the end of an iteration, the results of both process assessment and reviews and previews are reported to stakeholders. At the end of the software development project, a final sustainability report is compiled out of the journal and interim presentations, and a presentation is prepared. Finally, the report is presented to the stakeholders in a formal sustainability presentation and report release event. At the end of the project, the development team executes the sustainability retrospective to discover knowledge that should be preserved for upcoming software projects.

3.6.1 Roles

The model knows three roles: development team, sustainability executive and customer representative.

The development team develops the software product but is also specifically responsible for conducting the reviews and previews and recording the results in the journal. It participates in the interim presentations. After the development project has ended, it performs a retrospective.

The sustainability executive is responsible for organising the sustainability-centric tasks and events of the software development process. She is especially responsible for performing process assessment and recording the results in the journal. She is also responsible for preparing the final sustainability report and for preparing and presenting the final report release event. Additionally, she maintains the journal and supports the team regarding sustainability issues.

The customer representative is responsible for representing the customers' requirements and expectations on the sustainability of the software product when attending the interim presentations and the final report release event.

3.6.2 Sustainability Review and Preview

The sustainability review and preview is mainly performed by the development team. As its name suggests, it consists of two main parts: the review and the preview. The review takes a look at the work done, whereas the preview tries to look ahead at future implementations based on the decisions taken by reviews.

Thus, the reviews and previews form a continuous improvement cycle within an iteration. The reviews and previews should take place after approximately two-thirds of an iteration, so that there is a chance to implement the decisions made within the same iteration. Depending on the length of an iteration, there may be more than one review and preview. In such a case, the succeeding review and preview review also the decisions made by its predecessor.

Another asset of the reviews and previews is a short presentation of the data gathered for process assessment and its ongoing assessment results, which is given by the sustainability executive. Care must be taken not to blame the development team for results that are not under its control, for example, huge amounts of carbon dioxide emissions due to an insufficient thermal insulation of office buildings. On the other hand, there may be results that are under the control of the development team, for example, choosing public transport over the car for business trips or the daily way to work.

Eventually, the outcomes of the reviews and previews are presented in an interim presentation at the end of each iteration.

The outcomes and measures taken should be recorded in the sustainability journal for further reference, for example, to examine later on, if taken measures were successful or not and as a basis to prepare the interim presentations and the final report.

The reviews and previews are scheduled and organised by the sustainability executive in close consultation with the development team. This does not necessarily mean that the executive is also the facilitator of the meeting.

3.6.3 Process Assessment

Process assessment continuously assesses the negative and positive effects on sustainable development caused by the SDLC. It is the sole responsibility of the sustainability executive to perform this task.

More specifically, data should be gathered in such detail that it is possible to perform an ongoing carbon footprint calculation. It should be considered that the

carbon footprint is calculated in such a way that the footprint of the development activities can be reported to the development team in review and preview sessions and to the customer representative in interim presentations. How a carbon footprint of the software development phase can be calculated is shown in [23].

Another possibility is to perform a continuous LCA according to ISO 14040; however, these are of much higher complexity than carbon footprints.

The calculations should consider not only past development activities but also planned activities and approximated emissions from using the product, for example, planned number of person months for the project, emissions from shipping and using the product according to expected sales figures. The approximations can be supported by other participants, for example, energy consumption data measured by the development team.

Relevant data that should be collected, approximated or calculated can be easily spotted in the life cycle of software products (Fig. 3.2). This includes, but is not limited to, energy for heating, ventilation and air conditioning of offices, energy for preparing hot water, electrical energy for lighting, energy for workstations and IT infrastructure, length of business trips and used means of travel as well as proportionate impacts of commonly used corporate departments.

The collected data and their results are recorded in the sustainability journal.

3.6.4   Interim Sustainability Presentations

In interim sustainability presentations, the development team reports the measures taken and their results, which attenuate the negative impacts and strengthen the positive effects on sustainable development to the customer representative. Usually, these are the outcomes of the review and preview sessions and depend on the extent to which the developed measures were successful.

The sustainability executive reports the results and current approximations of process assessment to the customer representative. This may be the current carbon footprint as well as approximations of distribution and usage.

3.6.5   Sustainability Journal

The sustainability journal is the focal point of the process add-on. It is a well-structured document in which all data items and results of process assessment as well as the issues, solutions and outcomes discovered in review and preview sessions are recorded.

The journal is not intended to be an extensive textual report. Actually, it should be result oriented and as short and concise as possible but still extensive enough so that the decisions and measures taken can be traced.

The technical implementation of the journal is open. Although a handwritten collection of papers in a folder is sufficient, we suggest to implement the journal at least with a text processor, preferably with a system for computer-supported collaborative work, for example, wiki, issue tracker or online forum.

The document should be organised in three sections:

1. Environmental impacts and effects recorded and calculated in process assessment
2. Measures taken in reviews and previews according to the pattern:

    (a) Initial situation (e.g. if available supported by data from measurements, software tests, footprint calculation, etc.)
    (b) Decisions made or actions taken
    (c) Verification of the effectiveness of the decisions/actions (e.g. supported by appropriate measurements/metrics)

3. Appendix with data items and their origin that were used to calculate the figures in process assessment (e.g. energy consumption, CO_2 emissions per kWh electrical energy, allocation procedure of proportionate emissions from common corporate departments, assumed emission base data for different types of travel, etc.)

3.6.6 Sustainability Retrospective

The sustainability retrospective takes place at the end of the development project. In a team facilitation approach, the development team and the sustainability executive reflect on the development process to find valuable ways to improve the sustainability of future software projects. Discussion input can be taken from the journal but also from any other source connected with the project. Expected outcomes are, for example, lessons learned, best practices, decisions for future projects and reflections on impacts and effects of software products. The findings should be preserved in a knowledge base.

3.6.7 Sustainability Presentation and Sustainability Report

At the end of the development project, the sustainability executive prepares the final sustainability report and the presentation ceremony (aka sustainability presentation and report release).

The report is compiled from the journal and the interim presentations. If process assessment performs an ongoing carbon footprint calculation, the report may mainly consist of the carbon footprint calculation and its data sources but also of outstanding measures and solutions that were developed to decrease or increase the effects on sustainable development.

The report is accompanied by a business presentation that summarises the report.

The sustainability presentation and report release finalise the development project. It is a small formal ceremony, where the sustainability executive presents the final report (the already prepared business presentation) and symbolically hands the report over to the customer representative. The development team should attend the ceremony.

The final sustainability report is based on the journal and the interim presentations. It should contain the results of process assessment, that is, the carbon footprint calculation, but also outstanding measures and solutions from reviews and previews that made the software product more sustainable.

3.7 Integrating Aspects in Existing Process Models

This section shows how the add-on can be integrated for two examples of SDLCs. The first example uses the agile lightweight method Scrum [15]; the second example uses OpenUP [16], a simplified and open source version of the Unified Process [27].

3.7.1 Scrum

Scrum [15] is an agile iterative software development method that tries to deliver a potentially shippable product increment in each iteration. An iteration called Sprint starts with the Sprint planning meeting and ends with the Sprint review meeting and the Sprint retrospective.

The product owner, who represents the interests of all stakeholders, and the development team, the so-called Scrum team, negotiate in Sprint planning meetings the development goals of the new Sprint. These goals are used in Sprint review meetings to assess whether the Sprint was successful or not. The overall objective is to deliver a potentially shippable product in each Sprint. The Sprint retrospective follows directly after the review meeting. The objective of this team facilitation approach is to solve impediments to the development process or to discover issues that have the potential to advance the development process. During a Sprint, the team meets in a stand-up meeting every morning, called the Daily Scrum, which is led by the Scrum Master. Here, team members shortly report what they implemented yesterday, what they plan to implement today and any impediments for the planned work. The Scrum Master is responsible for the Scrum process to work as expected and that the team works by the rules and practices of Scrum. He or she is intentionally not the project leader.

The green and sustainable software engineering add-on can be implemented as follows (Fig. 3.4): The role of the customer representative is mapped to the product owner and obviously the development team to the Scrum team. The role of the sustainability executive does not fit to any existing role in Scrum. When the process
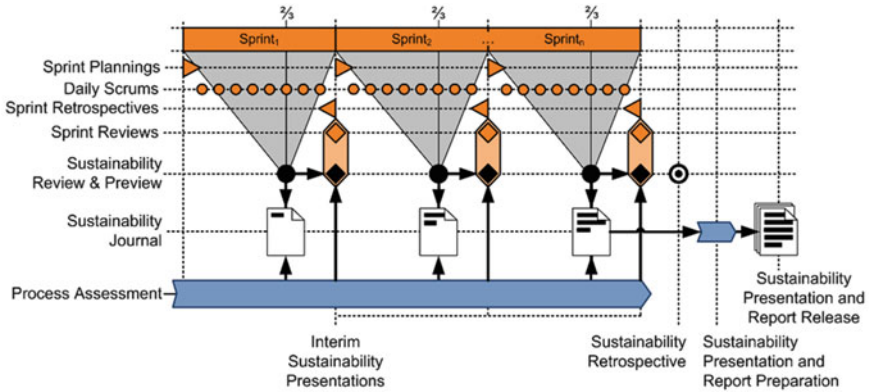
Fig. 3.4 Integrating sustainable software aspects into Scrum [37]

is instantiated, possibly a member of the development team may hold this role as well.

An iteration is represented by a Sprint. After approximately two-thirds of a Sprint, the sustainability review and preview takes place. The interim presentations are added to the Sprint review meetings. Process assessment is performed in parallel to the different Sprints of the project.

When the project is finished, the presentation and report preparation task can be performed just before the last Sprint review meeting. Thus, the final sustainability report can be presented and released during the course of the meeting, without the need for another meeting. The sustainability retrospective should not be confused with the Sprint retrospective, because of the different objectives.

3.7.2 OpenUP

OpenUP [16] is, in principle, a simplified version of the Unified Process [27], without many optional parts. It was released on open source by IBM and is further developed by the Eclipse Foundation. It is an agile, iterative and mostly tool agnostic development method, well suited for small development teams.

The structure of the life cycle has two dimensions. The first dimension represents the life cycle phase: inception, elaboration, construction and transition. In each phase, several sequential iterations may occur. The second dimension represents different disciplines that occur during the life cycle: requirements, architecture, development, test, deployment, project management and development environment (there is no special design discipline; design is part of the development discipline). Activities occur during iterations (things to do) and consist of several tasks, which

are in turn classified into the disciplines (an activity may have steps that belong to different disciplines). Which activities are executed during a phase and the work-load generated by an activity depends on the focus of the current phase and the necessity of the running process. This means that tasks of different disciplines can be executed with different strengths according to the needs of the process phase.

Each iteration closes with the Assess Results task within the plan and manage iteration activity, where the team demonstrates the value and gathers feedback from stakeholders. After the assessment has been performed, the retrospective tries to improve development and process execution of any existing obstacles. When an iteration is the last iteration of a phase, a milestone review meeting takes place. Its objective is to reach an agreement together with the stakeholders on moving to the next phase. If an iteration is the last iteration, the task has an additional step to gain the final acceptance of the software product by the stakeholders.

We propose performing sustainability review and preview meetings after two-thirds of an iteration as an additional task within the Plan and Manage Iteration activity that occurs in all iterations of all phases (Fig. 3.5). Additionally, interim sustainability presentations can be performed alongside the Assess Results task at the end of an iteration. Additionally, the sustainability retrospective is performed at the end of the last iteration of the project. The sustainability presentation and report preparation task also belongs to the Plan and Manage Iteration activity. The process assessment task belongs to the activity of ongoing tasks. However, the activity is not present in the inception phase. Hence, it has to be added to make process assessment available at the very beginning of the project.
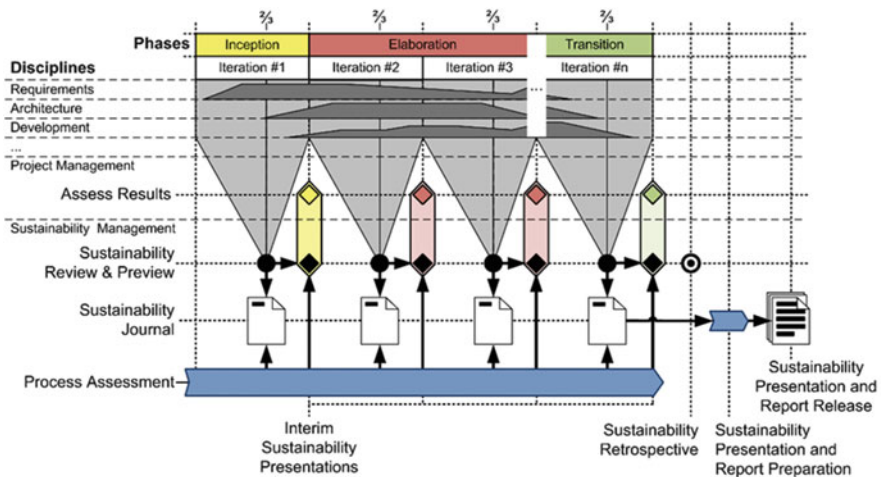
Fig. 3.5 Integrating sustainable software aspects into OpenUP [37]

3.8   Conclusion

In order to enable a green and sustainable software engineering process as a first step, we propose to have a look at the life cycle of software products. This life cycle consists of three parts: (1) development, including the development and the distribution of software products; (2) usage; and (3) end of life, including the deactivation and disposal of the product. This kind of life cycle thinking ('from cradle to grave') helps to consider the sustainability effects of software over the whole product life. Keeping the whole life cycle in mind, one should especially concentrate on the processes to create green and sustainable software.

In this context, we presented a process model to organise the development of green and sustainable software in software development processes. To do so, software engineering processes should be enhanced, taking sustainability aspects into account. These are sustainability reviews and previews, process assessment, sustainability journal, sustainability retrospective, sustainability presentations and the final sustainability report. The first enhancements consider environmental impacts. On the one hand, the needs during the development itself are considered in the so-called sustainability reviews and previews. On the other hand, the effects over the whole software development life cycle are continuously assessed within process assessment. During the whole procedure of developing a software product, sustainability presentations should be implemented to inform everyone participating in the process about sustainability issues (e.g. measurement results of energy consumption). In that way, possibilities can be found to optimise the sustainability of the process and the product itself. The overall decisions about sustainability aspects are documented in the sustainability journal. At the end of the process, there is a sustainability retrospective to exchange lessons learned, a final sustainability presentation and a final sustainability report.

The presented generic model can be adapted to existing models. This was exemplified for Scrum and OpenUP. Overall, the generic model can be seen as an add-on for software development processes to produce green and sustainable software. Our aim is to display a generic extension for any of the existing process models instead of presenting a complete process without specific activities.

The described procedure model for the development process can be complemented by models for using, administrating and purchasing green and sustainable software. Indeed, the development process is the central process of the engineering phase, since the course for the phases following in the life cycle of software is set. Overall, sustainability aspects need to be taken into account as early as possible to be able to optimise them in the best possible way.

References

1. Agarwal S, Nath A, Chowdhury D (2012) Sustainable approaches and good practices in green software engineering. IJRRCS 3(1):1425–1428
2. Albertao F (2004) Sustainable software engineering. http://www.scribd.com/doc/5507536/Sustainable-Software-Engineering#about. Accessed 30 Nov 2010
3. Albertao F, Xiao J, Tian C et al (2010) Measuring the sustainability performance of software projects. In: IEEE Computer Society (ed) 2010 I.E. 7th international conference on e-business engineering (ICEBE 2010), Shanghai, China, pp 369–373
4. Amsel N, Ibrahim Z, Malik A et al (2011) Toward sustainable software engineering: NIER track. In: 2011 33rd international conference on software engineering (ICSE), pp 976–979
5. Behrendt S, Kahlenborn W, Feil M et al (2007) Rare metals. Measures and concepts for the solution of the problem of conflict-aggravating raw material extraction – the example of coltan. Texte, 23/07. Umweltbundesamt
6. Bordage S, Philippot O, Bordage F et al (2012) Ecoconception web-Les 100 bonnes pratiques: Doper son site et réduire son empreinte écologique. Eyrolles
7. Braungart M, McDonough W (2009) Cradle to cradle. Remaking the way we make things. Vintage, London
8. Calero C, Bertoa MF, Moraga MÁ (2013) Sustainability and quality: icing on the cake. In: Penzenstadler B, Mahaux M, Salinesi C (eds) Proceedings of the 2nd international workshop on requirements engineering for sustainable systems, Rio, Brasil, 15 July 2013. http://ceur-ws.org
9. Capra E, Francalanci C, Slaughter SA (2012) Measuring application software energy efficiency. IT Prof 14(2):54–61
10. Coroama VC, Hilty LM, Birtel M (2012) Effects of Internet-based multiple-site conferences on greenhouse gas emissions. Telematics Informatics 29(4):362–374
11. Deutsches Institut für Normung e.V. (2003) Umweltmanagement-Integration von Umweltaspekten in Produktdesign und -entwicklung. Deutsche und englische Fassung ISO/TR 14062:2002, 1st edn. DIN-Fachbericht. Beuth, Berlin
12. Deutsches Institut für Normung e.V. (2006) Environmental management – life cycle assessment – requirements and guideline (ISO 14044:2006); German and English version EN ISO 14044:2006 13.020.10(DIN EN ISO 14044:2006-10)
13. Dick M, Naumann S (2010) Enhancing software engineering processes towards sustainable software product design. In: Greve K, Cremers AB (eds) EnviroInfo 2010: integration of environmental information in Europe. Proceedings of the 24th international conference on informatics for environmental protection, 6–8 Oct 2010, Cologne/Bonn, Germany. Shaker, Aachen, pp 706–715
14. Dick M, Naumann S, Held A (2010) Green web engineering. A set of principles to support the development and operation of "green" websites and their utilization during a website's life cycle. In: Filipe J, Cordeiro J (eds) WEBIST 2010 – Proceedings of the sixth international conference on web information systems and technologies, vol 1, Valencia, Spain, 7–10 April 2010. INSTICC, Setúbal, pp 48–55
15. Eclipse Foundation (2008) Scrum. http://epf.eclipse.org/wikis/scrum/. Accessed 28 Jun 2010
16. Eclipse Foundation (2009) OpenUP. http://epf.eclipse.org/wikis/openup/. Accessed 28 Jun 2010
17. Fischer J, Naumann S, Dick M (2010) Enhancing sustainability of the software life cycle via a generic knowledge base. In: Greve K, Cremers AB (eds) EnviroInfo 2010: integration of environmental information in Europe. Proceedings of the 24th international conference on informatics for environmental protection, 6–8 Oct 2010, Cologne/Bonn, Germany. Shaker, Aachen, pp 716–725
18. GeSI, Global e-Sustainability Initiative; The Climate Group (2008) SMART 2020: Enabling the low carbon economy in the information age

19. Hilty LM (2008) Information technology and sustainability. Essays on the relationship between ICT and sustainable development. Books on Demand, Norderstedt

20. International Organization for Standardization (2005) Software engineering – software product quality requirements and evaluation (SQuaRE) – guide to SQuaRE 35.080(ISO/IEC 25000:2005 (E))

21. Johann T, Dick M, Kern E et al (2012) How to measure energy-efficiency of software: metrics and measurement results. In: IEEE (ed) Proceedings of the first international workshop on green and sustainable software (GREENS) 2012, held in conjunction with ICSE 2012, The international conference on software engineering, June 2–9, Zurich, Switzerland. IEEE Computer Society, pp 51–54

22. Käfer G (2009) Green SE: ideas for including energy efficiency into your software projects. Technical briefing (TB2). In: 31st international conference on software engineering, Vancouver

23. Kern E, Dick M, Drangmeister J et al (2013) Integrating aspects of carbon footprints and continuous energy efficiency measurements into green and sustainable software engineering. In: Page B, Fleischer A, Göbel J et al (eds) EnviroInfo 2013 – environmental informatics and renewable energies. 27th international conference on informatics for environmental protection. Proceedings of the 27th EnviroInfo 2013 conference, Hamburg, Germany, 2–4 Sept 2013. Shaker, Achen, pp 300–308

24. Kern E, Dick M, Johann T et al (2011) Green software and Green IT: an end user perspective. In: Golinska P, Fertsch M, Marx-Gómez J (eds) Information technologies in environmental engineering. Proceedings of the 5th international ICSC symposium on information technologies in environmental engineering (ITEE 2011), 1st edn. Springer, Berlin, pp 199–211

25. Kern E, Dick M, Naumann S et al (2013) Green software and green software engineering – definitions, measurements, and quality aspects. In: Hilty LM, Aebischer B, Andersson G et al (eds) ICT4S ICT for sustainability. Proceedings of the first international conference on information and communication technologies for sustainability, ETH Zurich, 14–16 February 2013. ETH Zurich, University of Zurich and Empa, Swiss Federal Laboratories for Materials Science and Technology, Zürich, pp 87–94

26. Kocak SA (2013) Green software development and design for environmental sustainability. In: 11th international doctoral symposium an empirical software engineering (IDOESE 2013), 9 Oct 2013, Baltimore, MD

27. Kruchten P (2003) The rational unified process. An introduction, 2nd edn, The Addison-Wesley object technology series. Addison-Wesley, Boston

28. Lami G, Fabbrini F, Fusani M (2012) Software sustainability from a process-centric perspective. In: Winkler D, O'Connor R.V, Messnarz R (eds) EuroSPI 2012, CCIS 301. Springer, pp 97–108

29. Mahaux M, Canon C (2012) Integrating the complexity of sustainability in requirements engineering. In: Svensson RB, Berry D, Daneva M et al (eds) 18th international working conference on requirements engineering: foundation for software quality. Proceedings of the workshops RE4SuSy, REEW, CreaRE, RePriCo, IWSPM and the conference related empirical study, empirical fair and doctoral symposium, pp 28–32

30. Mahmoud SS, Ahmad I (2013) A green model for sustainable software engineering 2013. Int J Software Eng Its Appl 7(4):55–74

31. Naumann S, Dick M, Kern E et al (2011) The GREENSOFT model: a reference model for green and sustainable software and its engineering. SUSCOM 1(4):294–304. doi:10.1016/j.suscom.2011.06.004

32. Penzenstadler B (2013) What does sustainability mean in and for software engineering? In: Hilty LM, Aebischer B, Andersson G et al (eds) ICT4S ICT for sustainability. Proceedings of the first international conference on information and communication technologies for sustainability, ETH Zurich, 14–16 Feb 2013. ETH Zurich, University of Zurich and Empa, Swiss Federal Laboratories for Materials Science and Technology, Zürich

33. Penzenstadler B, Femmer H, Richardson D (2013) Who is the advocate? Stakeholders for sustainability. In: 2013 2nd International workshop on green and sustainable software (GREENS), pp 70–77
34. Penzenstadler B, Khurum M, Petersen K (2013) Towards incorporating sustainability while taking software product management decisions. In: 7th international workshop of software product management, Essen, Germany
35. Penzenstadler B, Bauer V, Calero C, Franch X (2012) Sustainability in software engineering: a systematic literature review. In: Proceedings of the 18th international conference on evaluation and assessment in software engineering
36. Pressman RS (2010) Software engineering. A practitioner's approach, 7th edn. McGraw-Hill, New York
37. Research Project "GREENSOFT" (2014) Website: research project "green software engineering" – downloads. http://www.green-software-engineering.de/en/downloads.html
38. Sahin C, Cayci F, Clause J et al (2012) Towards power reduction through improved software design. In: IEEE Energytech 2012, 29–31 May 2012, Cleveland, OH. IEEE, Piscataway, NJ
39. Shenoy SS, Eeratta R (2011) Green software development model: an approach towards sustainable software development. In: 2011 Annual IEEE India conference (INDICON), pp 1–6
40. Shojaee H (2007) Rules for being a green software engineer. Ship Software OnTime! The blog that helps you build great software. http://shipsoftwareontime.com/2007/12/24/rules-for-being-a-green-software-engineer/. Accessed 26 July 2011
41. Sommerville I (2011) Software engineering. International Edition, 9th edn. Pearson, Boston
42. Taina J (2010) How green is your software? In: Tyrväinen P, Cusumano MA, Jansen S (eds) Software business. First international conference, ICSOB 2010, Jyväskylä, Finland, 21–23 June 2010. Proceedings. Springer, Berlin, pp 151–162
43. Taina J (2011) Good, bad, and beautiful software – in search of green software quality factors. CEPIS UPGRADE XII (4):22–27
44. Tischner U, Dietz B, Maßelter S et al (2000) How to do EcoDesign? A guide for environmentally and economically sound design. Verlag form, Frankfurt am Main

Chapter 4
Constructing Green Software Services: From Service Models to Cloud-Based Architecture

Fei Li, Soheil Qanbari, Michael Vögler, and Schahram Dustdar

4.1   Introduction

In recent years, green software research is gaining momentum from the acute need for sustainable development as well as the far-reaching effect of ICT to our society. "[Green and] Sustainable Software is software, whose direct and indirect negative impacts on economy, society, human beings, and environment that result from development, deployment, and usage of the software are minimal and/or which have a positive effect on sustainable development" [4]. Based on this definition, green software research is growing in two directions. The first direction looks into the runtime energy consumption of software [15] and its engineering pro-aspects of our society and investigates how software can be used to improve the sustainability of a broader range of business, social, and individual activities [5]. This chapter is focused on the research and development in the second direction—to leverage software to solve sustainability problems on a wider scope.

The emergence of cloud computing and Internet of Things (IoT) makes software services further reach out to the physical world at a larger scale. Many existing business operations are being improved with respect to scalability and manageability through automation. Such new developments have strong implications to green software research since sustainability can be improved by applying software services based on these new computing paradigms. This view prompts us to rethink the delivery models and service scope of green software: when green software is delivered as online services, or green software services (GSS), a broader range of business, governments, and individual processes can more easily employ the services to reduce their energy consumption. Furthermore, more flexible business relationships can be established between different stakeholders so that the financial

F. Li • S. Qanbari (✉) • M. Vögler • S. Dustdar
Distributed System Group, Vienna University of Technology, Vienna, Austria
e-mail: Li@dsg.tuwien.ac.at; Qanbari@dsg.tuwien.ac.at; voegler@dsg.tuwien.ac.at; dustdar@dsg.tuwien.ac.at

and social interests of green software can in turn promote its research and development.

To this end, this chapter presents the core GSS constructs based on a cloud-based architecture. The research is aimed at providing a systematic, high-level view on four key elements in the development of GSS: stakeholders, their requirements, various business models, and corresponding software architecture. The stakeholders of GSS are detailed with the services they can provide and consume, thus clarifying their interests to GSS. Based on this analysis, we present the domain-independent core requirements to GSS that are considered by different stakeholders. Six business models are proposed to promote collaborations of stakeholders on the delivery of GSS. Each of the business models is then mapped to a scope of high-level components in the IoT PaaS architecture [11]. This chapter completes our previous work on business model analysis for GSS [6] by providing the software architecture for each model, which can serve as the reference for constructing GSS by service providers.[1] In the end, we will discuss how the business models are related to cloud services and the challenges of realizing a marketplace for GSS.

The remaining part of this chapter is structured as follows: Sect. 4.2 introduces the core stakeholders and their relationships in GSS. Then the requirements for GSS are analyzed in Sect. 4.3. Section 4.4 introduces the business canvas for describing business models and our past work on IoT PaaS as the basis for GSS architecture. Section 4.5 presents the business models and corresponding cloud services. Finally, Sect. 4.6 discusses the implications of these business models to the research and development of GSS, and the chapter is concluded in Sect. 4.7.

4.2 Stakeholders in GSS

In order to investigate the requirements for GSS, we first conduct a detailed analysis on the related stakeholders. Overall, the stakeholders are classified into core stakeholders and supportive stakeholders. Core stakeholders are those business entities that have direct business interests in providing GSS, whereas supportive stakeholders are those organizations or individuals who have financial or social interests in GSS but do not directly profit from delivering GSS. The stakeholders are illustrated in Fig. 4.1.

---

[1] Part of this chapter was published in [6], and this chapter extends the previous publication by adding the software architecture for each model.
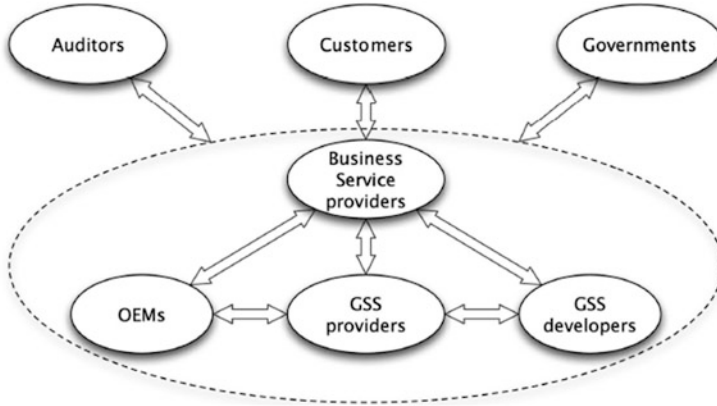
Fig. 4.1 Stakeholders of GSS

4.2.1 Core Stakeholders

- **Business service providers** are operating diverse businesses that might benefit from GSS, for example, building operators, transportation services, and data centers. They share one common objective of maximizing the sustainability of their businesses (by saving energy or reducing wastes). Since they have the direct financial incentives of reducing operational costs by applying GSS to their existing businesses, they are the main driving force for the development of GSS.
- **Original Equipment Manufacturer (OEM)** produce equipment that are the source of energy consumption. Their efforts alone on developing sustainable equipment can result in significant energy conservation (e.g., LED lights and energy-efficient chillers). Integrating, managing, and leveraging the energy-saving capabilities of OEM devices are one of the most important approaches of realizing GSS. More importantly, GSS is able to optimize complex systems that constitute a large number of OEM equipment [3].
- **GSS providers** offer GSS that are used by business service providers. The services are in diverse business domains and of various functions, such as home automation, facility management, offline analysis, and so on. The services provided by them can be realized by other stakeholders or domain experts. GSS providers retain the service interface and establish direct business relationships with customers who need GSS.
- **GSS developers** realize business logics and optimization methods in the target domain. Different implementations of the same business logic could have considerably different effects in terms of sustainability. Thus, domain knowledge is often required for GSS developers. Promoting a GSS developer community will help to leverage the growing amount of data available on the Internet and the increasingly connected devices [1] to create more diverse GSS and apply them to more business domains.

Table 4.1 Roles of stakeholders

Stakeholders	Services or information provided	Services or information consumed
Business service providers	• Domain-specific business service • Domain process optimization knowledge	• Optimization services • Auditing
GSS providers	• Device integration • Optimization services • Data acquisition • Analytics • Data visualization	• Domain-specific knowledge • Device connectivity • Application development
OEMs	• Devices • Device optimization knowledge	• Device integration
Application developers	• Business logic implementation • Optimization method implementations	• GSS platform services • Domain-specific knowledge • Device optimization knowledge
Governments	• Raising public awareness to GSS • Public information • Regulation and legislation • Policy enforcement	• Standardization • Auditing results
Auditors	• Auditing	• Data access • Process monitoring
Service consumers	• Usage feedbacks	• GSS • Business services

4.2.2 Supportive Stakeholders

- **Governments** are strong driving forces and important advocates for promoting sustainable development. Their activities such as policy-making, policy enforcement, legislation, and standard enactment are essential for the adoption and long-term growth of GSS. Governments are also important public information providers.
- **Auditors** systematically assess the performance of GSS. They provide a solid baseline for comparing and further improving GSS by applying standardized evaluation methods. Audit may be applied to any system components or domain-specific business services.
- **Service consumers** are, in most current GSS applications, passive stakeholders who benefit from business services at reduced costs. However, since the behavior of consumers is also decisive to the effects of GSS, the usage information and behavior patterns can be collected for designing better GSS.

Table 4.1 summarizes the main services provided and consumed by each stakeholder.

4.3   Requirements for GSS

Based on the previous analysis about different stakeholders and their involvement in GSS, this section presents the high-level requirements to GSS. These requirements are intended to be domain independent, as each of them addresses the needs of multiple stakeholders:

1. *Identifying core services (concerned stakeholders: all core stakeholders)*
   Given our perspective that green software is a software designed to improve the sustainability of other business, social, or individual activities, the easiness of engaging with various target domains is the key to the wide adoption of GSS. This prompts us to identify a set of core GSS capabilities that are independent from the domain specifics of target systems while making GSS easily adaptable to address domain requirements. Therefore, we regard the following core capabilities critical to the success of GSS. These core capabilities form the basic features for a domain-independent platform that can serve as the basis of green software services:

   - **Collecting and preparing data from target systems**: This means that GSS should have the capability to access and acquire raw data from diverse environments that are to be made 'greener', including physical environments, hardware, software, and information generated by humans. Thus, GSS should not be restricted to certain communication protocols or data exchange formats. Furthermore, for the data to be effectively utilized in GSS, GSS has to prepare the raw data at two levels. The basic level is syntactic preparation to normalize the presentation of the data. This task is covered in many domain-specific standardization efforts like oBIX (Open Building Information Exchange) [14]. The higher level of data preparation is in the ongoing research on semantic technologies, which aims at the semantic interoperability of systems [13].
   - **Customizing for different target system**s: Providing the GSS to a specific target system means that GSS should be tailored on provisioning. Such tailoring can either be physically separating system components, configuring them, and deploying only the necessary components for the target system or virtually excluding the target system from using other irrelevant system capabilities. Thus, extensibility and customizability are critical to GSS. This chapter will illustrate how such tailoring can result in various service models.
   - **Accommodating various scales**: The target systems of GSS may be of largely different scales, ranging from a single home to a large city.[2] They differ with each other in terms of numbers and types of equipment, data volume, and resource requirements for applications. Thus, GSS should be customized not only for specific functions but also customized in terms of the

---

[2] http://www.pacificcontrols.net/projects/ict-project.html

resources needed for each target system. This requires flexible allocation of computing resources and on-demand scaling of GSS.

2. *Supporting a broad range of process optimization and analytics methods (concerned stakeholders: GSS providers, business service providers, GSS developers)*

   Based on the data collected, GSS will support a variety of process optimization methods on the underlying resources they are running to ensure the reliability, sustainability, and cost-effectiveness of the semantics they have promised. GSS is not limited to a certain optimization method but has to decide on the exact method to be used according to the specific task at hand. The known methods for optimizing energy usage include offline data modeling and simulation [3, 17], context-aware controls (e.g., presence-based light control), and agent-based systems [9]. The growing amount of data further requires GSS to be employed for results in big data research. The implication of this requirement is that the optimization capability of GSS lies in the capabilities of handling various types of data formats, including time series data, well-structured data, unstructured data, or even natural language.

3. *Supporting realization and enforcement of sustainability policies (concerned stakeholders: GSS providers, business service providers, governments, auditors)*

   Supporting sustainability policies is one of the basic expectations from GSS. Multiple aspects need to be considered for the realization of sustainability policies. First of all, GSS is required to model and understand the sustainability policies of the target systems, for example, temperature limit for a green building. Second, GSS is required to find efficient ways of meeting the goals defined by the policies, for example, how to efficiently control the HVAC systems. Third, in realizing the policies, GSS themselves should be energy efficient, complying to sustainability policies for IT systems.

4. *Ensuring end-to-end privacy and security coverage (concerned stakeholders: business service providers, service consumers)*

   Given the sensitivity to private, commercial, and public data that might be used in GSS, data privacy, security, and confidentiality have to be incorporated into GSS from the beginning and visible during every stage of development of the system. Furthermore, since GSS usually need to apply certain controls or changes in order to change the energy consumption of target systems, such control capabilities are to be secured so that only the authorized software components and personnel can perform the allowed controls.

5. *Supporting collaborations between stakeholders (concerned stakeholders: all stakeholders)*

   As stated in the stakeholder analysis, successful GSS are not built by any single party but built by the collaborations of multiple stakeholders, each of which provides their knowledge and services. The collaborations are the basis for flexible business models. Thus, GSS need to support collaborations by providing interfaces for different stakeholders, for example, interfaces for third-party developers to develop new functions or interface for auditors to

inspect the sustainability status. More importantly, such collaborations are not only reflected on system interfaces but also the capability of directly presenting and sharing business interests on serving certain needs of GSS stakeholders.

4.4 Background

The impetus behind business model development is how to create, deliver, and capture the values of a system. The fast pace of cloud innovation and increasing diversity of GSS, coupled with unpredictable and ever-changing business requirements, require flexible and adaptive business models built upon reliable framework to maximize GSS utilization. Before proceeding to the introduction of business models and corresponding architectures, this section will introduce the business model description approach and the IoT PaaS architecture.

4.4.1 Business Model Description

This chapter leverages an established business model framework [2] to describe the business models of GSS. The framework is illustrated in Fig. 4.2.

A business model is described in four areas—*finance, infrastructure, customer, and value proposition*. These four sections have strong and mutual interrelationships with each other that have to be taken into account in forming business models. Financial aspects aim at providing profitable and sustainable revenue streams. The cost structure in the financial area is directly related to the stakeholders who are providing resources and conducting service activities, whereas the revenue structure is related to customers who are interested in the specific services. The monetary flow of this cost and revenue streams are effectively in use under the two models of *metered usage of service* and *subscription basis*. In the infrastructure area, the OEM devices are provided with optimization capabilities. The infrastructure aspect offers virtualization layer over infrastructure resources by providing utilization interfaces. The customer area's focus is on providing interfaces that define the consumer segments with their communication, distribution, and sales channels as a touching point for service delivery. Overall, the three areas converge on the value proposition of a business model. It seeks to solve customer problems and satisfy their needs with value propositions. Readers can refer to Bucherer et al. [2] to find more details on the description framework.
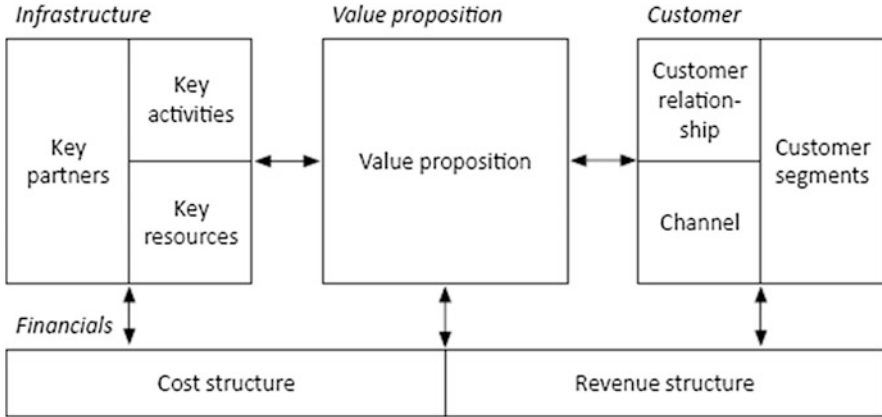
Fig. 4.2 The method of describing business models

4.4.2 IoT PaaS Architecture

The overall IoT PaaS architecture is depicted in Fig. 4.3. IoT PaaS is a domain-independent platform-as-a-service framework. In general, IoT solutions are highly domain specific, so the IoT PaaS framework is built to be generic and extendable enough to be used in different IoT domains. Furthermore, IoT PaaS provides essential platform services on cloud that can be used and extended by IoT solution providers.

To get a better understanding of the overall IoT PaaS architecture, we will start at the lowest tier, the *infrastructure*, which can be seen at the bottom of Fig. 4.3. The infrastructure inherits OEM devices, databases, file systems, and computation units. To communicate with these devices, the *virtualization* layer is used. This virtualization layer is both an integral part of infrastructure and platform. It provides device drivers and several low-level communication protocols to connect heterogeneous devices and furthermore offers a new level of abstraction by efficiently translating device/network interfaces to software interfaces. To deal with different domain-specific data models, which would lead to a new level of heterogeneity, the IoT PaaS uses *domain mediators* to mediate between different virtualized device interfaces. IoT PaaS provides two types of services related to data to handle real-time and persisted data, respectively. *Data processing* is focused on the processing and analyzing of real-time data generated by, for example, sensory devices, whereas data storing/retrieving facilitates storing, retrieving, and manipulating of persisted data by hiding the actual underlying data infrastructure. Since in IoT PaaS, each application runs in a complex and dynamic context, *application context management* is focused on providing and maintaining optimal runtime resources and software configurations for applications. Based on the resources acquired through tenant management, application context management helps applications to select the necessary resources at runtime to fulfil the
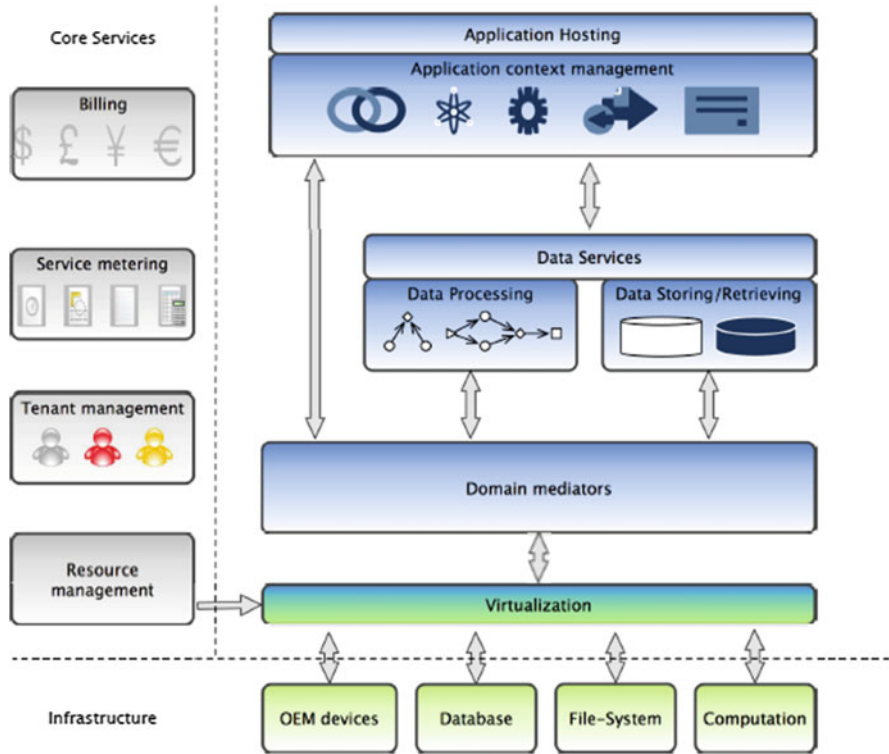
Fig. 4.3 The IoT PaaS architecture

functional requirements and meet service-level agreements and cost targets. The combination of tenant management and application context management provides a virtually isolated operational environment, enacting the concept of virtual verticals, for each application.

In addition to platform-specific components, IoT PaaS also offers a set of core services, which can be seen on the left-hand side of the figure. One of the most important services is *resource management*, which provides a registration point for any form of resource, for example, cloud resources, virtualized devices, control applications, etc. It monitors the resource status and enforces the access policies via the virtualization layer. In IoT PaaS, resources include not only cloud resources such as virtual machines and software instances in traditional cloud offerings but also IoT resources. Since device capabilities and control applications can be used by multiple tenants via virtualization, IoT PaaS uses *tenant management*, which assembles a consolidated view of the resources that are accessible by each tenant. To measure the usage of various services that can be used by an application, the *service metering* component mainly monitors service messages and invocations that are concerned by the platform and stakeholders. The metered information of both IoT and cloud resources gets composed to provide a comprehensive view of service

usage. To conclude the set of core services, *billing* generates bills for stakeholders by analyzing the metered information according to charging schemes, which gets configured by stakeholders.

4.5 Business Models and Reference Architecture

The GSS architectures that fulfill the requirements and satisfy the stakeholders will become the cornerstone of flexible GSS business models.

4.5.1 Infrastructure Services

The capabilities of OEM devices and other computational resources play a significant role in the value generation of infrastructure services. GSS customers can benefit from infrastructure services for accessing these capabilities through virtualization, as illustrated in Fig. 4.4.

Its core value proposition is to provide optimization services on OEM devices, thus directly reducing the cost of using these devices. Resources are virtualized [8] so that GSS can easily access and control them. GSS providers are responsible for device virtualization that opens up the APIs for customers to access the infrastructure capabilities, as illustrated in Fig. 4.5.
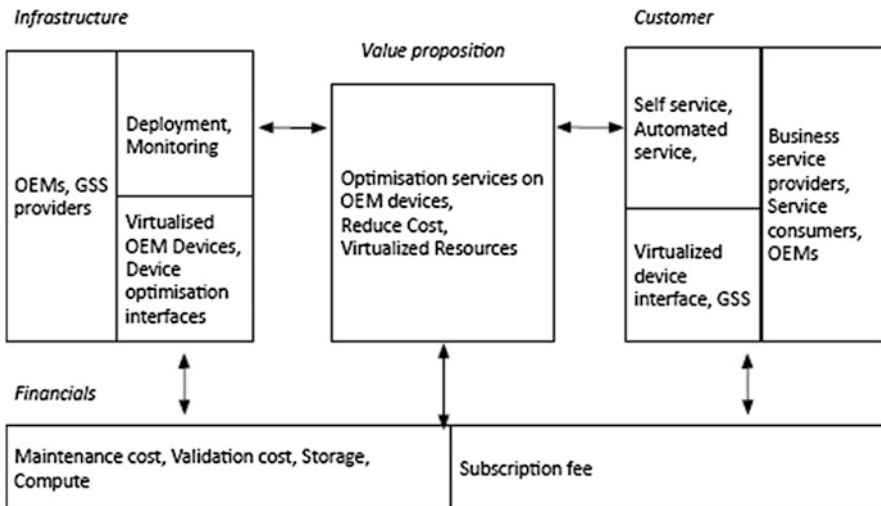
Fig. 4.4 Infrastructure services
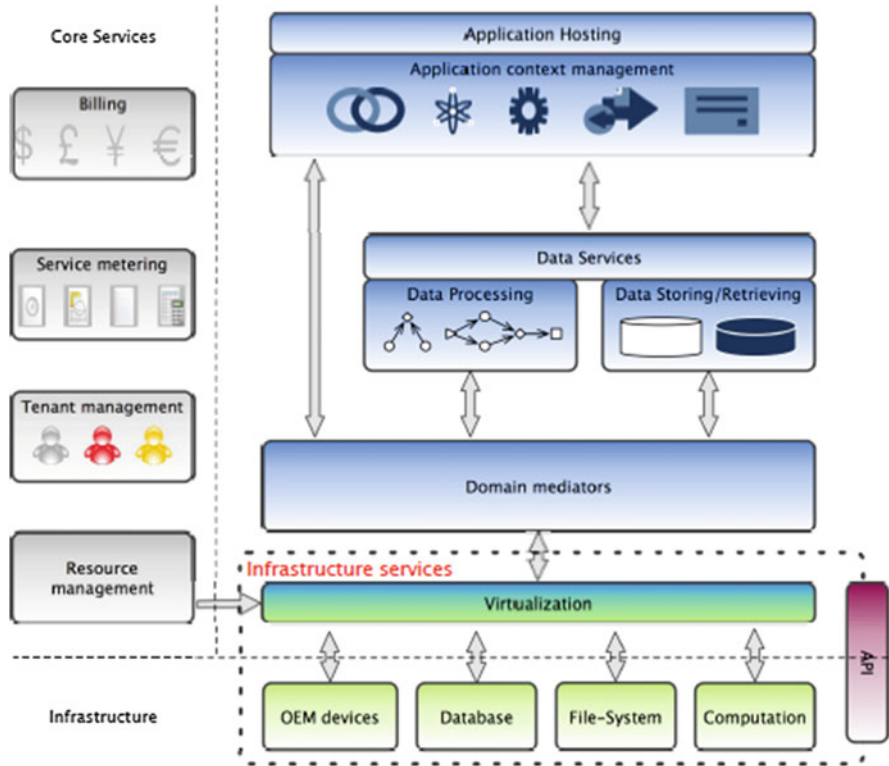
Fig. 4.5 Infrastructure services

The customer value that can be created by this model is mainly to provide business service providers and their consumers the ability to efficiently operate their facilities without constantly requesting the support of OEMs.

OEMs also benefit from this model through improved automation on their customer services since maintenance activities can be automated. The services under this model can be charged by "subscription" or "pay-as-you-go" models.

This business model provides an abstraction layer with programmable interfaces to perform administrative tasks over infrastructure resources. Therefore, GSS consumers do not manage or control the underlying resources but have control over how the infrastructure capabilities are used. Last but not least, another significant payback of these services is to cope with load fluctuations in an automated and consistent manner by cooperating with other virtualized resources, such as data storage and computational power.
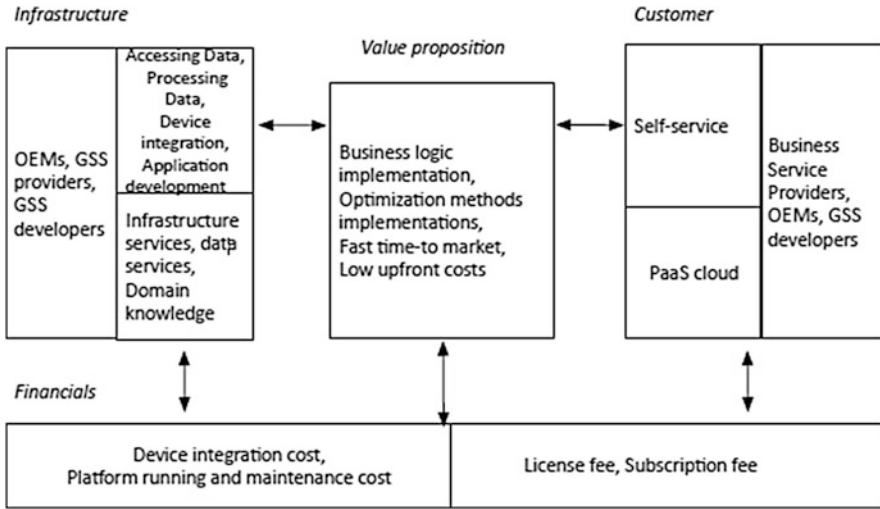
Fig. 4.6 Platform services

4.5.2 Platform Services

The platform services business model is illustrated in Fig. 4.6. It aims at providing GSS providers with a platform including application libraries, device integration APIs, data services, development environment comprising the end-to-end life cycle of GSS application coding, customization, testing, deploying, and hosting the applications as a service, as illustrated in Fig. 4.7.

The service provider of this model is not limited to certain business domains or optimization methods. OEMs can use the platform to realize the first business model—infrastructure services. GSS developers can develop GSS for specific customers on top of the platform, and GSS providers can operate the platform. The platform provides the core capabilities discussed in Sect. 4.3. Stakeholders can use the platform through a self-service portal. Essentially, this model is an adaptation of PaaS cloud for GSS. For stakeholders using the platform, the key value proposition is to realize green business logics and domain-specific optimization methods at lower upfront cost with faster time to market.

4.5.3 Virtual Verticals

Virtual verticals are provided to domain-specific business services such as smart buildings or data centers, as illustrated in Fig. 4.8. It deals with configuring and deploying appropriate GSS for specific domains and operational environment in order to make the vertical application more efficient in its business and
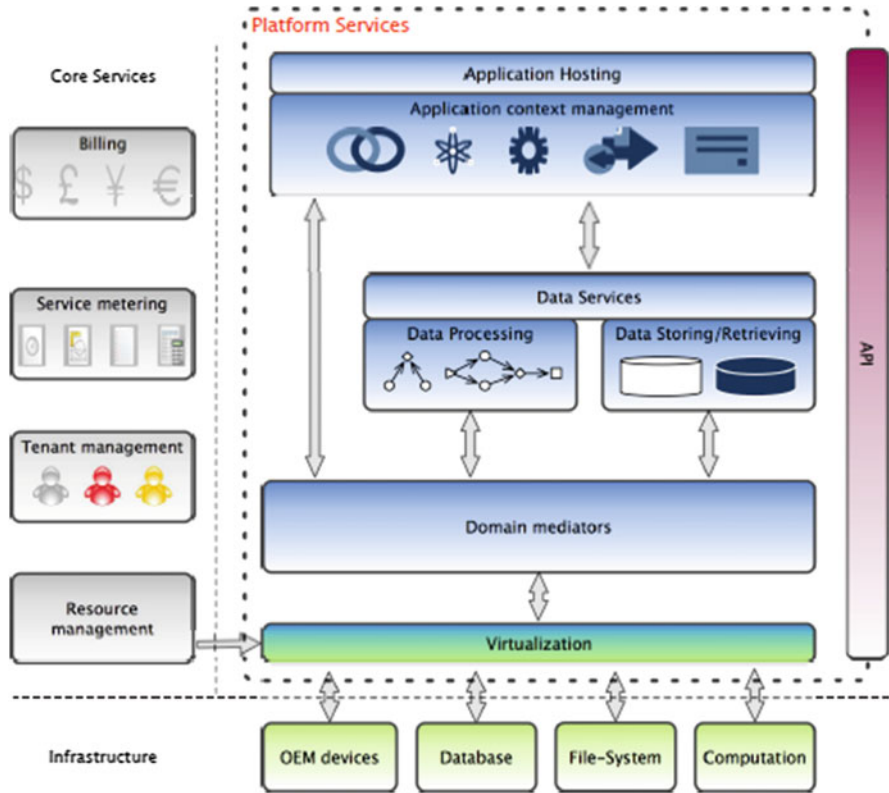
Fig. 4.7 Platform services
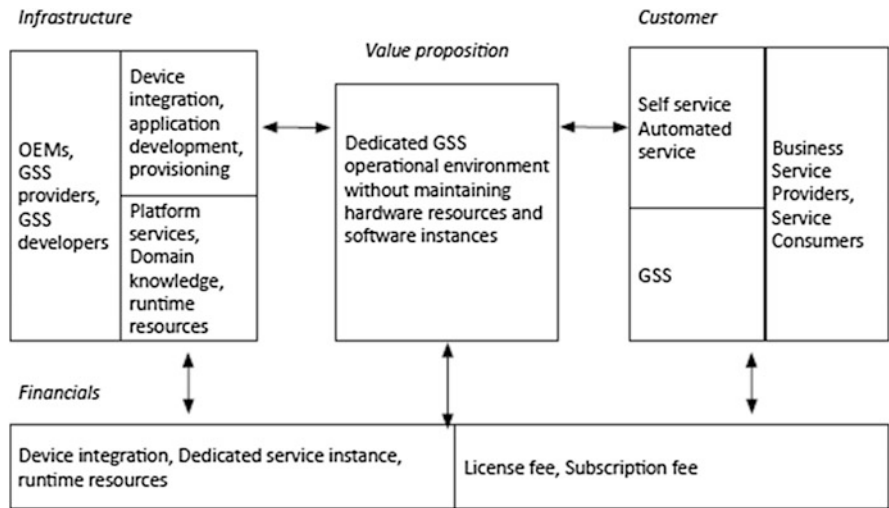
Fig. 4.8 Virtual verticals
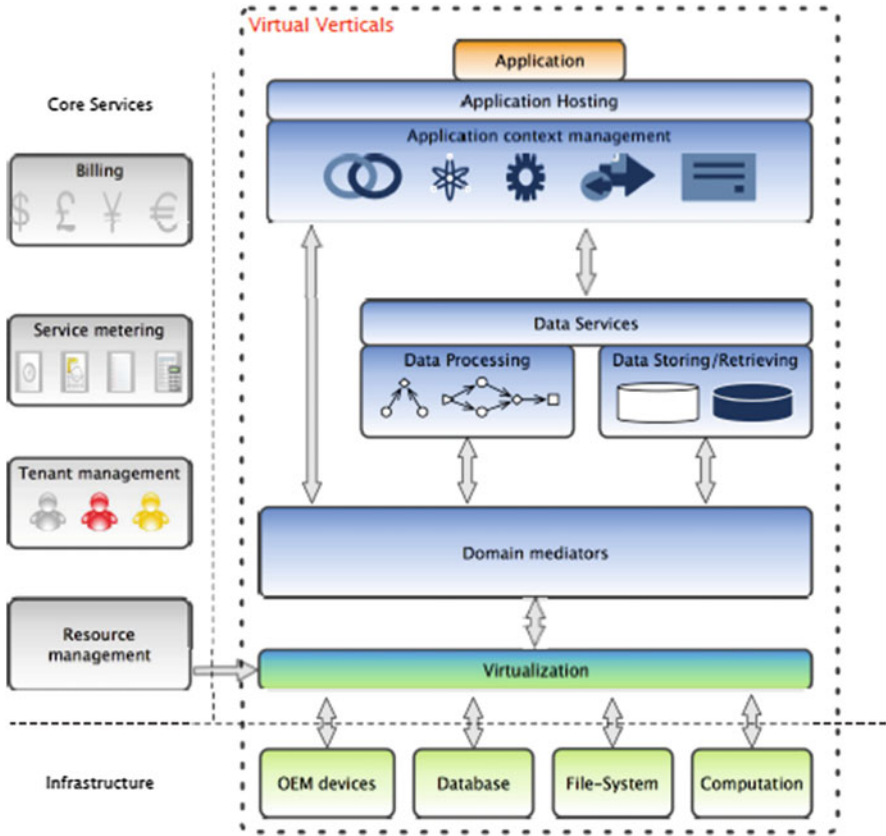
Fig. 4.9 Virtual verticals

technological context. 'Vertical' means that such applications are delivered as an end-to-end service coverage including physical devices, middleware, and applications for a certain physical environment, as illustrated in Fig. 4.9.

Virtual verticals can be realized by the collaborations of GSS providers, OEMs, and GSS developers. They integrate physical devices in the target environment and develop dedicated applications, such as light control or chiller management. Business service providers, who are the direct customers of virtual verticals, enjoy their dedicated GSS operational environments without committing computing resources or maintaining an IT infrastructure for GSS. Instead, they subscribe to virtual vertical services or license the software services for their operational environment. It is worth noting that the key difference between the proposed virtual vertical model and traditional physically isolated vertical model is the capability of sharing computing resources between verticals, which makes it easier for vertical applications to scale up.
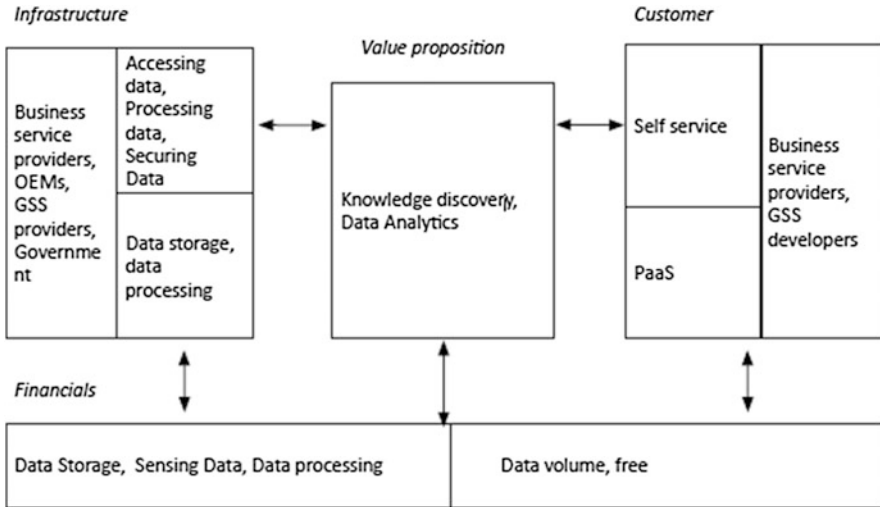
Fig. 4.10 Data services

4.5.4 Data Services

As the amount of data generated from business services is immense and still growing, the data service business model plays an important role in addressing data governance with a focus on data storage and processing aspects. Having the confidentiality of data properly managed and providing data to external experts or the public will further increase the utility of data. The data service business model is illustrated in Fig. 4.10. This model also deals with data concerns like privacy enforcement, up-to-dateness, data availability, and consistency in order to assure and improve data quality. Figure 4.11 illustrates the scope of data services. In principle, the service provider does not have to own the data sources, nor does it provide applications. It only concerns management and provisioning of data. Both real-time data and persistent data can be in the scope of service.

The data are owned by business service providers, OEMs, or governments. GSS providers offer the platform for them to open data access and establish business relationships with the customers who are interested in using the data for knowledge discovery or analytics. Business service providers can benefit from publicly available data to optimize their operations. GSS developers can create novel applications on the data or discover hidden knowledge in the data. The idea of providing data services have been realized through IoT platforms like Xively,[3] and more public data are being made available [7] for the developer community to discover their value.

---

[3] https://xively.com/
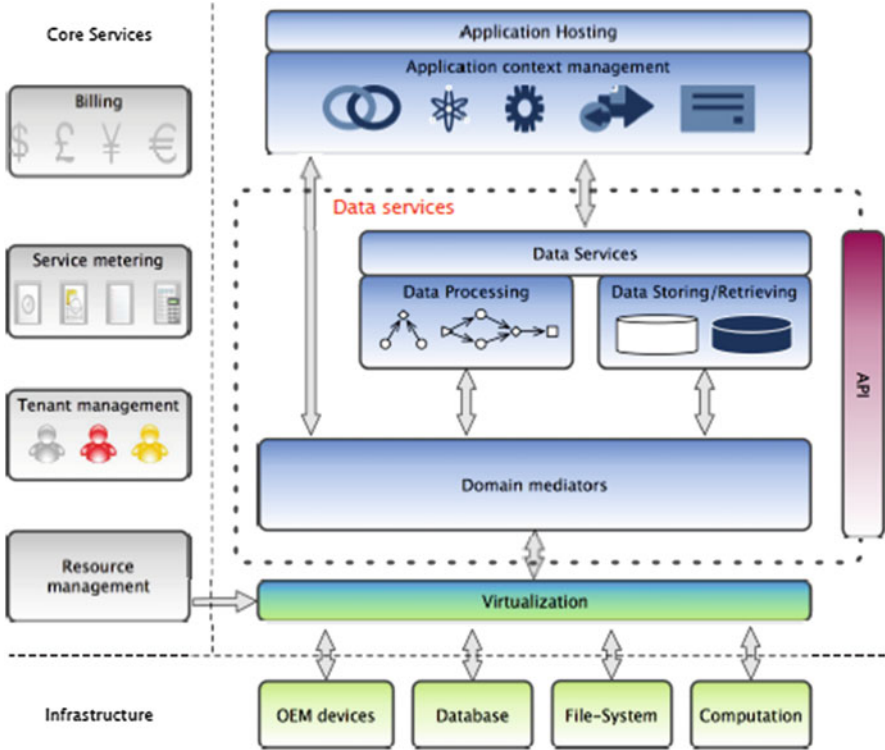
Fig. 4.11 Data services

4.5.5 Third-Party Applications

When the access to data, infrastructure, and core services is open to third-party developers, novel applications can be developed and provided as GSS, as illustrated in Fig. 4.12.

This business model can be built on top of either infrastructure services, platform services, or data services. In any case, this model is characterized by opening application development capabilities to third parties and, typically, providing application hosting service to them, as illustrated in Fig. 4.13. The involvement of third party in GSS can help establish a robust resource capacity planning with provider landscape. On GSS platform, the applications are offered online and used through subscription or licensing. Virtual verticals can be enhanced with third-party engagement to further extend the scope of applications. The range of applications can be broad, including optimized business processes, device optimization methods, or analytics. This model harnesses the creativity of the developer community and users to create various novel services.
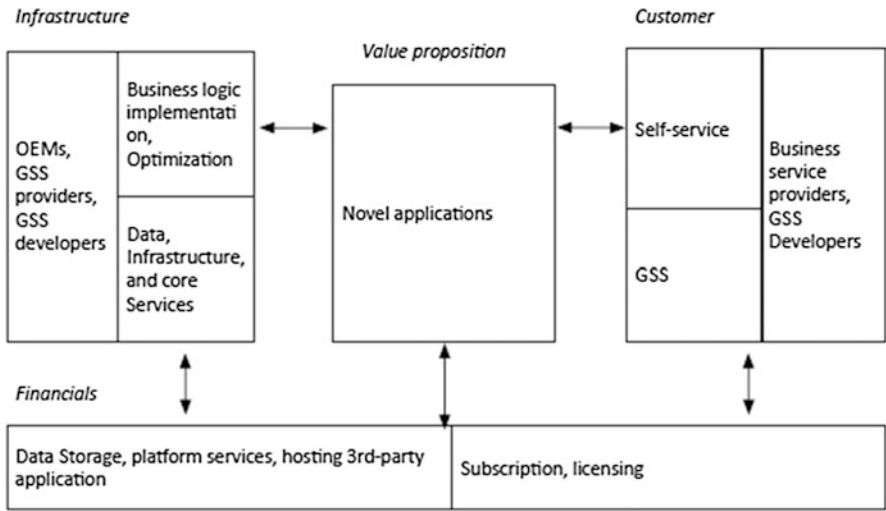
Fig. 4.12 Third-party applications
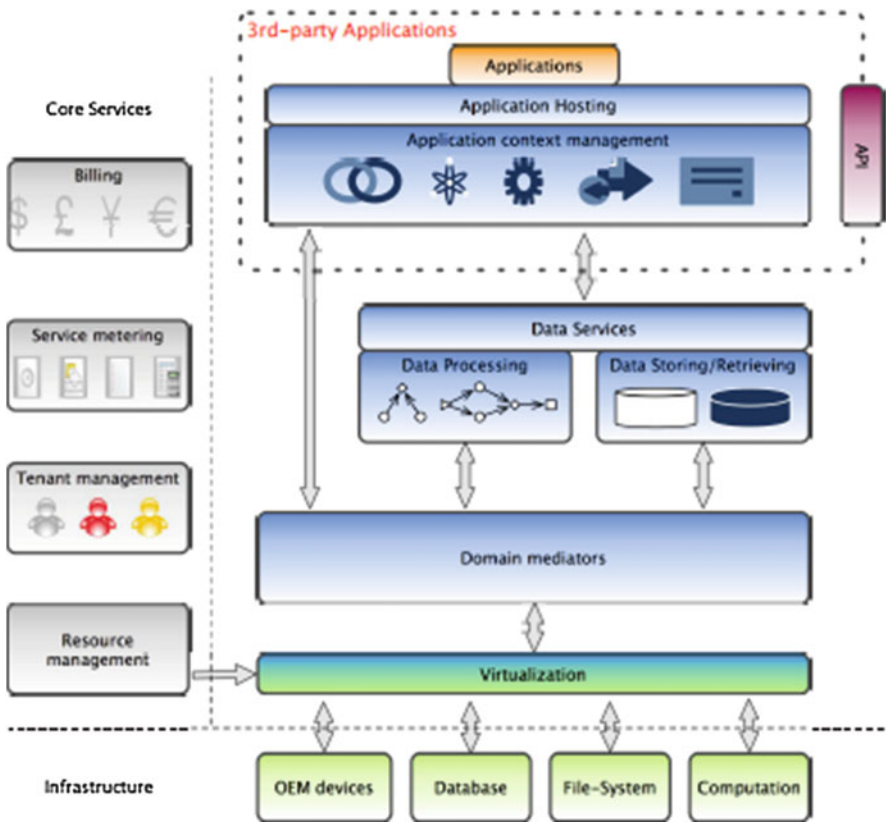
Fig. 4.13 Third-party applications
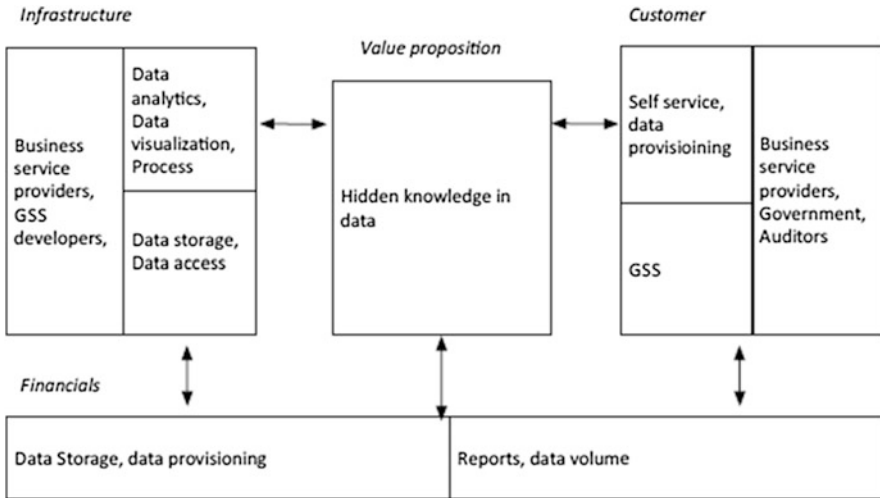
Fig. 4.14 Analytics as a service

4.5.6 Analytics as a Service

Analytics on the sustainability of business services are delivered by third-party analytics services [16], as illustrated in Fig. 4.14. Analytics as a service enables GSS consumers to leverage specialized analytics capabilities to identify previously unknown patterns and trends in their data.

This model is to a certain extent a type of third-party application. However, analytics as a service stresses that analytics are highly specialized tasks. External experts specialized in analytics are often required for conducting offline analysis. This model can also be employed by external auditing in order to assess the performance of GSS. Either data provided by the GSS platform or external data sources can be taken as input for analytics, as illustrated in Fig. 4.15. For the data to be effectively used in analytics, provisioning [10] (e.g., cleansing, normalization) is important but not necessarily a task of data providers since the provisioning can also be handled by data experts. Business service providers, governments, and auditors can all be interested in the results of analytics. They may purchase the reports or pay by the volume of data being analyzed.
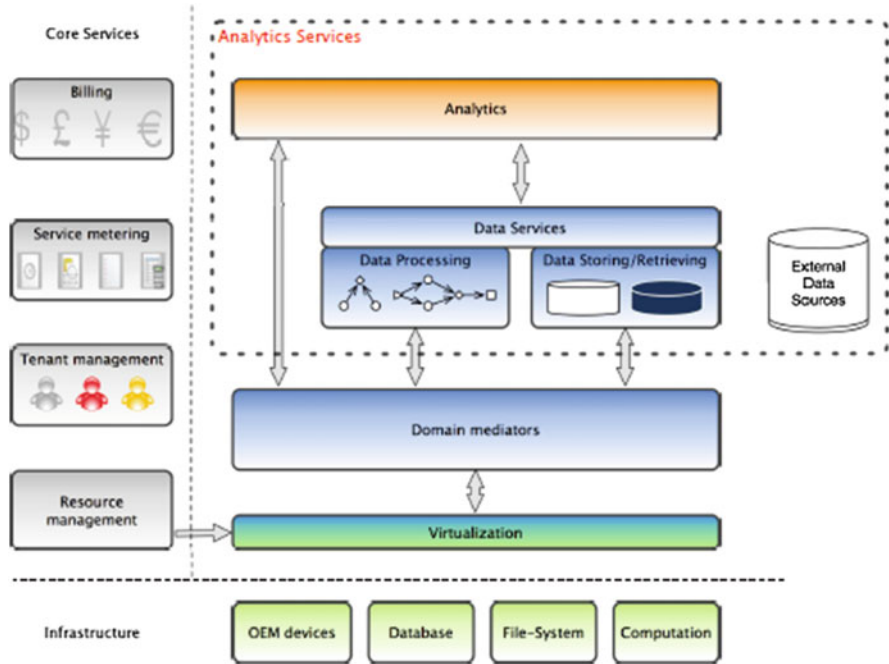
Fig. 4.15 Analytics as a service

4.6   Discussions

4.6.1   A Cloud Perspective to GSS

The rapidly growing popularity of cloud computing has made energy consumption of large data centers a trending topic in cloud research. Both of the two perspectives of green software research—energy efficiency of software and using software for energy efficiency—are applicable to cloud. On the one hand, the energy efficiency of cloud is affected by the operating systems, middleware, and applications. On the other hand, software tools can be built in order to manage the energy consumption of cloud. Mechanisms like resource scheduling and application workload prediction have been widely applied and are still improving.

   This chapter relates cloud and green software in a new perspective: the service models of cloud. IaaS (infrastructure as a service), PaaS (platform as a service), and SaaS (software as a service) [12] are already familiar to researchers and IT professionals. While green software services are provided on the Internet, the models are referred to here in order to classify the services provided by each stakeholder. In fact, the business models proposed in this chapter can easily be mapped to the cloud service models. Infrastructure services that open up interfaces of OEM devices are similar to the IaaS cloud, which provides computing resources as services. The

platform services and data services can be realized on a PaaS cloud by extending its capabilities of integrating with OEM devices. Third-party applications, virtual verticals, and analytics as a service are different forms of SaaS. Such mappings also serve as important reference for existing cloud service providers that plan to offer their services to the green software market.

4.6.2 Towards a Marketplace for GSS

The core vision of this chapter is to promote stakeholders in GSS to establish flexible business relationships through cloud-based service delivery models. As a natural result of the growing participation of stakeholders, a marketplace will emerge. In the marketplace, each type of service could be provided by multiple stakeholders, who offer different implementations for the same service with different QoS and price. Therefore, on top of the common platforms of GSS that stakeholders can collaborate at a technical level, there is a further need of supporting business activities including billing and SLA monitoring.

Although a cloud-based marketplace has been demonstrated by Amazon[4] and the concept of application store[5] is well accepted, a marketplace for GSS will face several new challenges. Evaluating and comparing GSS are hard since the effect of each service for each customer is tightly related to the specifics of target systems and their physical environments. Comprehensive metrics, especially domain-specific metrics, need to be incorporated or developed for objectively and accurately describing the services in the marketplace. These metrics will also help to establish effective monitoring mechanisms for GSS. A marketplace for GSS also means that the services should be delivered online. This challenges the deployment and provisioning mechanisms for GSS since business models such as infrastructure services and virtual verticals need to be coupled with devices in customers' physical environments. Automated or semiautomated methods, such as device integration or customer tools, need to be created to enable efficient and customized service delivery.

4.7 Conclusion

The research on the novel topic of green software is still at its infancy. Early research problems and technical solutions have been proposed, but wide adoption of green software is yet to happen. In this chapter, we tackled green software from the business perspective—trying to identify the requirements and the business

---

[4] https://aws.amazon.com/marketplace

[5] https://play.google.com/store

models that could benefit a wide range of stakeholders. We detailed the stakeholders and their interests in GSS. Based on this analysis, the high-level requirements of GSS were identified. Diverse business models were then proposed in order to motivate stakeholders to collaborate on the delivery of GSS. A cloud-based reference GSS architecture was presented with six variations to implement the corresponding business models. In the end, we discussed how the business models are related to cloud service models and the challenges of realizing a marketplace for GSS.

References

1. Atzori L, Iera A, Morabito G (2010) The internet of things: a survey. Comput Netw 54 (15):2787–2805. doi:10.1016/j.comnet.2010.05.010, URL http://dl.acm.org/citation.cfm?id=1862461.1862541

2. Bucherer E, Uckelmann D (2011) 10 Business models for the internet of things. Business 1–25

3. Cook J, Smith D, Meier A (2012) Coordinating fault detection, alarm management, and energy efficiency in a large corporate campus. In: 2012 ACEEE summer study on energy efficiency in buildings, pp 83–93

4. Dick M, Naumann S, Kuhn N (2010) A Model and Selected Instances of Green and Sustainable Software. In: Berleur J, Hercheui M, Hilty L (eds) What kind of information society? Governance, virtuality, surveillance, sustainability, resilience SE – 24, IFIP advances in information and communication technology, vol 328. Springer, Berlin, pp 248–259. doi:10.1007/978-3-642-15479-9 24

5. Dustdar S, Dorn C, Li F, Baresi L, Cabri G, Pautasso C, Zambonelli F (2010) A roadmap towards sustainable self-aware service systems. In: Proceedings of the 2010 ICSE workshop on software engineering for adaptive and self-managing systems – SEAMS '10. ACM, New York, pp 10–19. doi:10.1145/1808984.1808986. URL http://dl.acm.org/citation.cfm?id=1808984.1808986

6. Dustdar S, Li F, Truong HL, Sehic S, Nastic S, Qanbari S, Vogler M, Claesens M (2013) Green software services: from requirements to business models. In: 2nd international workshop on green and sustainable software (GREENS). IEEE, pp 1–7. doi:10.1109/GREENS.2013.6606415

7. (2012) Greenbiz: Hack City–Verge SF @Greenbuild Resources, GreenBiz Group Inc.

8. Guinard D, Trifa V, Karnouskos S, Spiess P, Savio D (2010) Interacting with the SOA-based internet of things: discovery, query, selection, and on-demand provisioning of web services. IEEE Trans Serv Comput 3(3):223–235. doi:10.1109/TSC.2010.3

9. James G, Cohen D, Dodier R, Platt G, Palmer D (2006) A deployed multi-agent framework for distributed energy applications. In: Proceedings of the fifth international joint conference on autonomous agents and multiagent systems – AAMAS '06. ACM, New York, p 676. doi:10.1145/1160633.1160752. URL http://dl.acm.org/citation.cfm?id=1160633.1160752

---

[6] http://pc3l.infosys.tuwien.ac.at/

10. Li F, Nastic S, Dustdar S (2012) Data quality observation in pervasive environments. In: The 10th IEEE/IFIP international conference on embedded and ubiquitous computing (EUC 2012), Paphos, Cyprus
11. Li F, Vögler M, Claeßens M, Dustdar S (2013) Efficient and scalable IoT service delivery on cloud. In: 6th IEEE international conference on cloud computing, (Cloud 2013), Industrial Track, Santa Clara, CA
12. Liu F, Tong J, Mao J, Bohn R, Messina J, Badger L, Leaf D (2011) NIST cloud computing reference architecture. NIST Special Publication 500, 292
13. Loutas N, Kamateri E, Tarabanis K (2011) A semantic interoperability framework for cloud platform as a service. In: 2011 IEEE third international conference on cloud computing technology and science. IEEE, pp 280–287. doi:0.1109/CloudCom.2011.45
14. OASIS: Open Building Information Exchange (oBIX) (2012). URL https://www.oasisopen. org/committees/tchome.php?wgabbrev=obix
15. Steigerwald B, Agrawal A (2011) Developing green software. Tech. rep., Intel
16. Sun X, Gao B, Fan L, An W (2012) A cost-effective approach to delivering analytics as a service. In: 2012 IEEE 19th international conference on web services. IEEE, pp 512–519. doi:10.1109/ICWS.2012.79
17. Zachhuber D, Doppler J, Ferscha A, Klein C, Mitic J (2008) Simulating the potential savings of implicit energy management on a city scale. In: 2008 12th IEEE/ACM international symposium on distributed simulation and real-time applications. IEEE, pp 207–216. doi:10.1109/DS-RT.2008.26

Part III
Economic and Other Qualities

Chapter 5
Economic Aspects of Green ICT

Héctor Fernández, Giuseppe Procaccianti, and Patricia Lago

5.1 Introduction

Over the last decades, the use of information and communications technology (ICT) and software systems has exploded, making our lives and work much more efficient. However, besides the benefits that ICT brings us, it also contributes significantly to environmental issues [11, 13], not only because of the electricity consumed by computers, data centers, networks, or other ICT utilities but also due to the rapidly growing computation needs of emerging software systems that contribute significantly to ever-increasing energy demands and greenhouse gas emissions [1, 21]. Decreasing ICT operation expenses becomes more and more crucial. How to make ICT greener (i.e., environmentally sustainable) and how to develop greener software have been gaining significant attention [16, 17, 24].

ICT can contribute to addressing environmental concerns in two ways: (1) by optimizing the implementation of ICT or migrating to a sustainable software and thus minimizing its own environmental impact and (2) by optimizing the business processes via more environmentally sustainable software and thus minimizing the use of ICT resources [5, 22]. Currently, many green ICT practices already exist to improve the energy efficiency of both IT and its supported processes. Examples include reducing the energy consumption of PCs by enabling power management features [20], enforcing double-sided printing to save both paper and energy [18], applying cloud computing technology to significantly reduce hardware and software resources needed for individuals [15], and using a fleet management system and dynamic routing of vehicles to avoid traffic congestion and thus minimize energy consumption and transportation costs [2].

From the examples above, we can see that greening ICT may save energy consumption (hence reducing cost), but it often requires additional investments,

H. Fernández (✉) • G. Procaccianti • P. Lago
VU University Amsterdam, Amsterdam, The Netherlands
e-mail: Hector.fernandez@vu.nl; g.procaccianti@vu.nl; p.lago@vu.nl

business process changes, and extra efforts from both companies and individuals. According to the analysis by Corbett [4], the most commonly cited driver for reusing Green IT practices is saving costs. Especially in times of economic crisis, cost reduction becomes the most important economic objective [23] of many companies. If green practices do not lead to an explicit (and significant) reduction of costs, environmental goals are often regarded as a nice optional bonus rather than a must-have target.

There is no one-size-fits-all green solution due to the diversity of requirements and characteristics of these enterprises. Executives need to assess the effectiveness of green ICT practices not only from a technical perspective but more importantly from an economic point of view and not only look into short-term return on investments (ROIs) but also have a vision on long-term ones. In addition, when green ICT practices involve software, calculating costs and ROIs is more difficult. The impact of software cannot be estimated in isolation, as it depends on many indirect factors including operation costs, hardware usage, human involvement, and system configuration. Often there is an intuition of some advantage gained when investing in such practices. This intuition is sufficient only if the company and decision makers are already fully committed to regreening their software and ICT portfolio. In most cases, evidence and quantification are the only way to handle the complexity of the practices mentioned above and hence to create such commitment.

To ensure economic benefits while making a sustainable business, there must be an alignment between green goals and organizational/business goals. Only when such an alignment is in place, the decision makers of a company can be motivated to take green actions [14]. In the business domain, the strategy modeling language (SML) [19] has been used to align business models with business goals, business plans, and optimization objectives to ensure business strategies can be optimally realized. However, a systematic approach is needed to describe solutions, actions, or strategies that can produce environmental benefits and enforce the alignment between green strategies and business goals [8].

Moreover, there is a need for quantifying in advance the economic impact of green practices. To the best of our knowledge, there is a software tool, named Going Green Impact Tool,[1] that compares the economic value among multiple green practices specifically for data centers. This tool provides a very comprehensive analysis on the key environmental and economic consequences of the application of certain green practices, which aids executives to determine the most effective practice. The major limitation of this tool lies in the fact that it works only for predefined practices, including server optimization, power management, virtualization, free cooling, and the reuse of waste heat. End users are not able to add other solutions for analysis and comparison.

Consequently, to our knowledge there is no single tool that is able to aid decision makers to run a holistic assessment and make informed decisions. In this chapter, we present a green model and software tool that allow us to model, estimate,

---

[1] http://ercim-news.ercim.eu/en79/special/the-going-green-impact-tool

quantify, and compare the economic consequences of the application of green practices. To do that, we used the e^3value model, which is an economic tool to model business networks and has been successfully applied in several real-life business case studies [7]. We designed and implemented a Web-based software tool that enables to analyze and compare the economic impact of applying a green practice versus an existing ICT solution. To this end, we carried out an experiment of modeling a green ICT practice called "desktop virtualization." The results show that by applying this practice a company would reduce overall expenses by 47 % and reduce electricity consumption by 20 %. This research combines formalized descriptions of green ICT practices with economic models estimating the business values of ICT solutions. By modeling the application of a green practice, we can customize the value exchanges to real scenarios and estimate the expected ROIs using our software tool.

The estimations above showed that while intuition was promising, the actual figures were delivering amazingly higher ROIs. We argue that such quantifications would convince organizations more easily to adopt green ICT practices and motivate them to reuse green ICT solutions even if requiring significant investments. The remainder of the chapter is structured as follows: Sect. 5.2 introduces the e^3value model; Sect. 5.3 presents our green strategy model; Sect. 5.4 shows an example on how to model a green ICT practice called *desktop virtualization*; Sect. 5.5 presents our Web-based software tool and an example using the *desktop virtualization* practice; and Sect. 5.6 concludes the chapter.

5.2 Background: The e^3value Methodology

e^3value models enterprises and end users exchanging things of economic value, such as goods, services, and money, in return for other things of economic value. In the following, we introduce the main concepts or constructs supported by the e^3value modeling tool and their associated notations [6, 12]:

- **Actor**: An economically, and often legally, independent entity. Examples of an actor include a customer, an organization, and a company. In the notation, an actor is represented by a plain rectangle.
- **Value object**: Something that actors exchange which is of economic value for at least one actor. A value object is a service, a good, money, or an experience. Examples of value objects are products, delivery service, and tuition fee. In the notation, a value object is represented as a label on a value exchange.
- **Market segment**: A set of actors that share a set of properties. Actors in a market segment assign economic value to a value object equally. In the notation, a market segment is represented by a set of stacked rectangles.
- **Value interface**: Something that groups value ports together and shows economic reciprocity. Economic reciprocity means that actors/market segment will only offer value objects if they will receive value objects in return. In the

notation, the value interfaces are drawn at the sides of actor/market segments as a thin rectangle with rounded corners, with value interfaces within.

- **Value port**: Something that is used by an actor/market segment to provide or request a value object. In the notation, a value port is shown as a small arrow inside a value interface.
- **Value exchange**: Something that connects two value interfaces and represents a potential trade of value objects. In the notation, value exchanges are drawn as lines connecting the port of actors/market segment to each other.
- **Dependency path**: The path where value exchanges, which is used to count the number of exchanges. In the notation, dependency path starts with a start stimulus and ends with a stop stimulus.

As illustrated in Fig. 5.3, many e^3value constructs can be associated with numbers or parameters, such as money transfers as well as the number of consumer needs (hence the need for concurrent computing). If done correctly, the e^3value modeling tool generates net value flow sheets, which show for each actor in the model the amount flowing into and out from an actor.

5.3 Modeling the Value Exchange of Green ICT Practices

The design of our green strategy model was inspired by the definitions collected by the Global Development Research Center (GDRC[2]), which is an independent nonprofit think tank carrying initiatives in education, research, and practice. Their goal is to contribute to broad-based global development by facilitating the creation and use of knowledge. The GDRC glossary of environmental terms elicited definitions from international organizations (like ISO and the Environmental Protection Agency of the United States). We found two definitions especially relevant to our purposes:

- **Definition 1**: A **green strategy** (aka environmental strategy) is a plan of action intended to accomplish a specific environmental goal.
- **Definition 2**: A **green goal** (aka environmental goal) is an objective that an organization sets itself to achieve and which is quantified where practical.

The first definition breaks down a strategy into two components: a plan of action and a specific environmental goal the action should help achieving. Moreover, the second definition implies that whenever applicable we should be able to quantify the extent to which an environmental goal is achieved. This calls, in our opinion, for the association of metrics (either qualitative or quantitative) that measure the contribution of each action to the achievement of the goal.

---

[2] www.gdrc.org

Fig. 5.1 Example: a green strategy for electronic bookstores
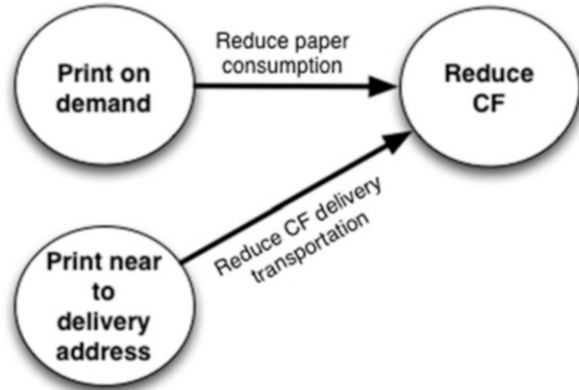
Putting the above elements together, we designed the first version of a green strategy as graphically illustrated by the example in Fig. 5.1. The example has been extracted from the electronic bookstore domain. It represents a strategy aiming at reducing the carbon footprint (CF) of printing books.

The strategy includes two actions: the first action *print on demand* refers to printing books only after customers order them. This action makes the business more sustainable by reducing the costs of storing books in large quantities before customers order them. While this effect is not "green per se," it does have an indirect positive impact on the total CF by reducing paper consumption to the minimum (i.e., exactly the amount of books that are actually ordered by customers). The second action *print near to delivery address is* to physically print the ordered books in a store as near as possible to the address of the customer. This allows to shorten the delivery distance, hence reducing the CF of transportation. We developed a number of examples (from both theory and practice) to challenge our first model of green strategy. In doing that, we have identified the following weaknesses:

1. Each action can have one or more effects that help in achieving the environmental goal. In order to select the best actions to put in place in a certain organization, we must make each effect explicit. In doing that, we can understand further what we need to measure to monitor the progress towards achieving the green goal. For instance, in the example of Fig. 5.1, the action effects (added on the associated arrows) identify that by monitoring paper consumption and transportation distances, respectively, we can draw the trend towards reducing the total CF.

2. While action effects are typically technical or environmental in nature, they do not explain the economic impact that they have. We had various discussions with companies actively involved in green ICT and/or in decreasing their CF, and all explained that the major incentive for them to go green is to reduce costs. Hence, if green strategies do not lead to an explicit (and significant) reduction of costs (hence increase in revenues), they are (again) nice but not part of the business
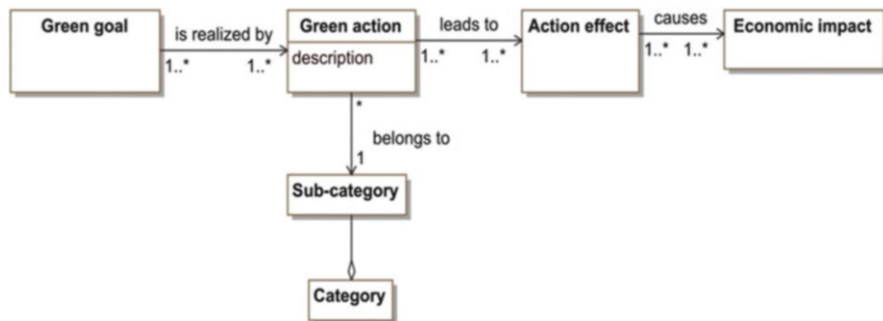
Fig. 5.2 Green strategy model

strategy of the organization. In periods of economic crisis, they are the first to be forgotten or neglected.

To challenge our first model in aligning green strategies and business strategies, we associated each action effect with its (potential) economic impact. This resulted in the revised green strategy model illustrated in Fig. 5.2. A green goal is realized by a number of green actions, and a green action can achieve a number of green goals. Each green action has a description to explain what the green action means. A green action leads to at least one action effect, which causes at least one economic impact. A green action belongs to one subcategory, which is a subset of a category. While the green goal represents ecologic impact of the strategy as a whole, the action effects detail the ecologic impact of each action individually.

5.4 Example of Modeling a Green ICT Practice

Aiming at assessing the feasibility of quantifying economic values of green ICT practices, we carried out an experiment by modeling the application of a practice called *desktop virtualization*, which has been selected from the list of green solutions provided by MJA (Meerjarenafspraken meaning long-term agreements).[3] This practice is described as:

   A desktop virtualization software facilitates the use of thin clients (i.e. workstations with minimal hardware configurations). These thin clients are far more energy efficient than regular fat client computers. There is however an increase in server side computing due to the extra load of providing the desktops, which leads to an increase in energy consumption of servers.

---

[3] The MJA is a voluntary agreement between the Dutch government and the largest energy consumers in the Netherlands, these being both large industries (e.g., banks and telecom providers) and higher education institutes (e.g., universities).

From this description, we elicited the following expected effects and associated economic impact:

- *Decrease in energy consumption of client workstations, which decreases energy consumption costs of client workstations*
- *Increase in energy consumption of servers, which increases energy consumption costs of servers*
- *Acquisition of thin clients, which may raise IT equipment acquisition costs*
- *Need to implement or purchase virtualization software, which requires short-term investment*

Using the e^3value modeling tool, we modeled an AS-IS situation (i.e., usage of fat client without virtualization) and TO-BE situation (i.e., usage of thin clients with virtualization) with the period of *3 years*. Figure 5.3 shows the AS-IS situation, where company X purchases a number of fat clients and servers from hardware suppliers in order to meet its computation needs, pays money to electricity suppliers for the electricity consumed by these fat clients and servers, and hosts an IT department (within the company or outsourced) to maintain the hardware devices ensuring they perform as expected. Figure 5.4 illustrates the TO-BE situation, where company X purchases thin clients rather than fat clients and the IT
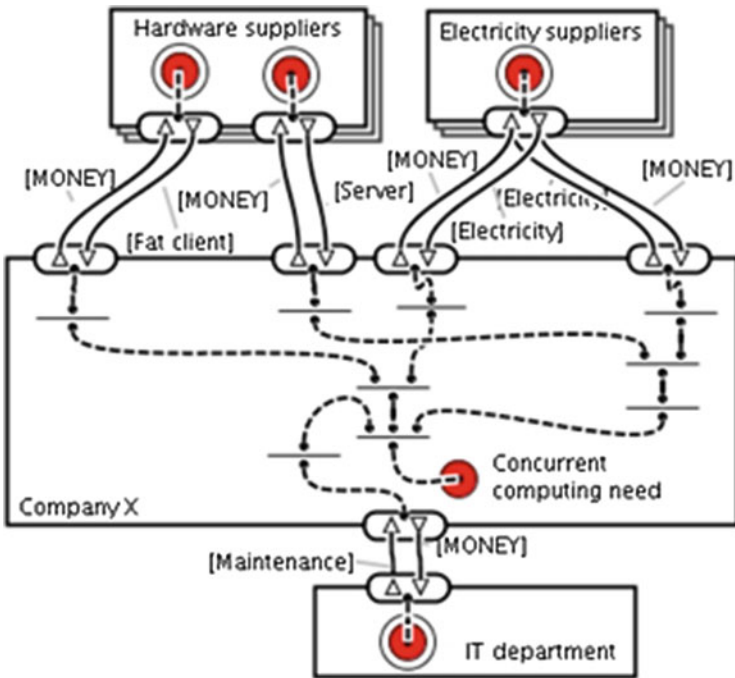
Fig. 5.3   Usage of fat clients, without virtualization
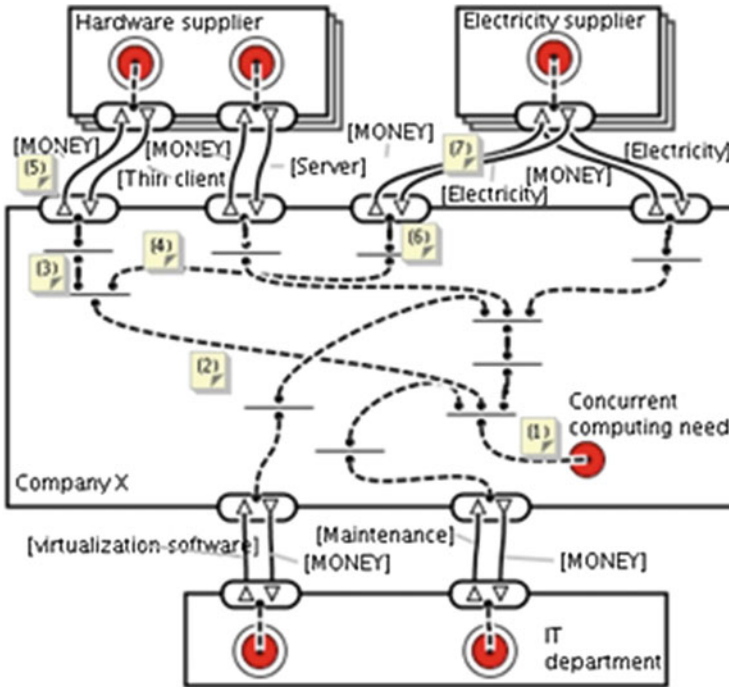
Fig. 5.4 Usage of thin clients, with virtualization

department has an additional task of providing and maintaining a virtualization software to deliver desktop virtualization service.

Value exchanges can be calculated along multiple dependence paths presented in the models. The paths start with the start stimulus "concurrent computing need" of company X. Such need can be fulfilled by a combination of three components: thin clients, a server, and maintenance service (see the AND fork in Fig. 5.4 labelled with (1)). To give an example of the dependent paths, consider the value exchanges related to the thin clients (2), which consists of acquisition of thin clients (3) and energy consumption of these thin clients (4). The acquisition of thin clients requires value exchange with the hardware suppliers (5), and the use of these thin clients requires electricity, which requires another value exchange with the electricity suppliers (7). Since the electricity is charged per month whereas the computing need is charged for 3 years, the fork (6) automatically normalizes costs in 1-month fractions.

After modeling the actors and value exchanges between them, we assigned parameters (with assumptions) to each value exchange in order to estimate the costs. The parameters we assigned for the two situations are listed in Table 5.1.

With the provided parameters, the e^3value modeling tool generated a spreadsheet that calculates the amount spent and the revenue gained by each actor after 3 years.

Table 5.1 The parameters assigned to AS-IS and TO-BE situations

Attribute	AS-IS situation	TO-BE situation
Number and type of clients	50 fat clients	50 thin clients
Price for each client	600 euro	400 euro
Number of servers	1	1
Energy consumption of a client per month	180 W × 10 h × 22 days = 39.6 kW	20 W × 10 h × 22 days = 4.4 KW
Energy consumption of a server per month	250 W × 24 h × 30 days = 180 kW	400 W × 24 h × 30 days = 288 kW
Price of energy per kWh	0.5 euro	0.5 euro
Maintenance cost per client per year	50 euro	25 euro
Maintenance cost per server per year	400 euro	400 euro
Desktop virtualization software license per year	n/a	400 euro

From the report, we could quantify the economic benefits that we should expect by reusing the green ICT practice:

- Energy consumption cost of client workstations is decreased by 88.8 %.
- Energy consumption cost of servers is increased by 37.5 %.
- Acquisition of thin clients requires an investment of 20,000 euro.
- Desktop virtualization software license requires an investment of 1,200 euro.

In this example, estimation results show that the applied practice would lead to an overall 47 % reduction of expenses and 20 % reduction of electricity consumption. However, when using the e^3value management tool, these measurements and a comparative analysis have to be manually calculated by users, which converts this procedure into a tedious task. Therefore, there is a need for automatization to help users in this decision-making process.

5.4.1 Findings

In general, we have been able to model all relevant fields in the original MJA document using e^3value according to our model. This gives us confidence on its suitability in modeling green strategies. The next step, however, would be to go back to the companies, or involve other ones, to check if our way to align economic impact and environmental benefits is effective and sufficient to decide on the best-fitting strategy.

For the sake of readability, the green practice in our experiment has been modeled in a simplified yet realistic context:

1. We limited the number of actors, including only the ones that are essential and highly relevant to the green practice. In real life, thin clients and fat clients can be purchased from multiple vendors in multiple times and at potentially

different prices. To simplify the models, we assumed that all the equipment is purchased from a set of vendors concurrently with a fixed price.

2. We simplified the calculation of electricity tariff. Electricity prices may vary depending on regions, countries, and distribution networks of the same country, type of customers, and type of contracts. In this experiment, we assumed that electricity is provided by a set of providers of the same type and with a fixed rate. However, rates are all taken from real providers.

3. We constructed the relation between IT maintenance and the company in a simplified manner. In reality, the way in which IT services are arranged can be quite complex, and the cost for IT maintenance can be charged differently. In this experiment, we assumed an average maintenance cost per hardware per year. In addition, often a company already has a number of computers in use; when deciding to apply desktop virtualization, the disposal cost of legacy hardware equipment should also be considered. Customization is needed when modeling the value exchanges and estimating the expected ROIs in real scenarios.

Further, the case study allowed us to collect the following additional observations on the codification as well as a list of issues that should lead to further improvements in our strategy model:

- Advantages of using $e^3 value$ in alignment with our green strategy model:

    - **The understandability of green actions is improved**:
        As we mentioned earlier, the green actions provided to us were documented by domain experts who made assumptions that readers have sufficient background knowledge to understand. However, when shared with other data centers or presented to a third party (e.g., our university for research purposes), the documented green actions are often not completely understandable and are less usable. Modeling the green actions makes it easier to share and communicate them since assumptions and domain knowledge embedded in the descriptions become explicit. For instance, in our description of the scenario, the consequences of using fat/thin clients with/without virtualization are less clear to the reader before modeling. Only after modeling, it is clear that from an environmental perspective, it would use less energy (lower carbon footprint) and maintenance, and from an economic perspective it saves costs for energy consumption and saves costs for maintenance. Knowing these consequences is essential for companies to understand and select the green action.

    - **Searching and selecting green actions for specific purposes become easier**:
        Very often, a company would search for green actions to achieve certain environmental goals, which is also the purpose of sharing green actions among multiple data centers under the MJA agreement. When action effects and economical impacts are explicit, they can be used as criteria for companies to search for green actions that fulfil their business requirement. Further,

explicit action effects and economical impacts may aid companies to justify and reason about the selection of certain green actions.

– **The completeness of the documentation of green actions is improved**:

The model encourages the author of a green action to document and, most importantly, to think about the environmental effects and economic impact that the green action may bring. In the future, when the green strategy model is commonly used for documenting green actions, the authors are guided with what type of critical information to provide. As a result, the chance that the documentation of green actions is complete is much higher.

– **Dependences between green actions are modeled**:

When modeling the green actions, we observed that some green actions are dependent or related to each other. For example, one green action could be to use adiabatic cooling for the resources of a data center, meaning that "as a complement to the direct free cooling we can evaporate water into the airflow to remove the heat in the air." Another similar green action is described as "Moistening and drying air are expensive and energy intensive. Use equipment that operates between 20 and 80 % of relative humidity." The former action results in high humidity of the air, and the latter action proposes to use devices that may tolerate wider ranges of humidity so that no extra efforts are needed to moisten or dry air. Obviously, the latter action provides a solution for the problem that the former action introduces. If such a relation is explicitly modeled, it is easier to justify the economic impact of both green actions as a whole.

– **Advantages of visualizing value exchanges**:

The results show that the models in our experiment well simulate the value exchanges under the simplified context and make the economic value of the green practice explicit. The e^3value technique provides a graphical overview of a resource exchanging network of a company. The visualization of participants and their relations in terms of value exchanges aid the analysis of the economic viability of the network. Therefore, the e^3value technique helps to consider a green practice in the context of the business model of a company; it encourages the alignment between business strategies and environmental ICT solutions.

Explicitly modeling the resource exchanges related to green practices within the business model of a company also urges ICT technicians to be aware of economic value of certain ICT solutions. Technicians often consider only quality attributes (e.g., performance, security) when proposing ICT solutions to meet the business needs of a company. The short- and long-term economic impact of the ICT solutions, however, often gets little attention, as long as the solutions meet the budget planned. Using the e^3value technique, ICT technicians are able to compare alternative ICT solutions, especially from the economic perspective, and decide the one that suits best the company's needs. For instance, desktop virtualization can be implemented in many different ways: by using thin clients and storing the "virtualized desktop images" on a central server (as we modeled in our experiment) or by running

multiple virtual machines on local hardware such as laptops without a server. While the former requires a central image management software, the latter requires the realization of desktop virtual machines. These two solutions may require different actors and different value exchanges. With the help of the e^3value technique, technicians are able to compare the economic influence of different solutions and thus make informed decisions.

– **The e^3value technique versus spreadsheet applications**:

One could argue that without using the e^3value technique, a spreadsheet application, such as an Excel sheet that records and calculates the cost, would also be sufficient. We agree that using Excel (or similar software tools) would be computationally equivalent to the e^3value modeling tool in terms of the calculation of costs. In fact, the report generated by the e^3value modeling tool is in the form of Excel spreadsheets. However, e^3value models are different from spreadsheet applications, which focus only on numbers and calculations. The e^3value modeling tool, instead, provides a graphical interface both for illustrating the interrelated financial dependencies between actors for filling parameters by end users. The e^3value model cannot be replaced by any spreadsheet applications specifically because it helps to achieve the following two goals:

1. To support communication of green practices among different types of stakeholders. While technical stakeholders would be comfortable in working directly with formulas and textual calculation (like in Excel), there is the need to communicate about a practice with business people and strategic decision makers.
2. To facilitate reuse of the same green practice in different organizations having different ways of implementing them (e.g., because of different departments involved or different factors that are variable in one company and constant in another). Whenever a practice is reused, its contextualization changes. While applying the changes in a visual model is straightforward (assuming one knows the modeling notation), applying the same changes in a textual calculation (like in Excel) is error prone and hinders reuse.

• For demonstration purposes, we show that it is feasible to use the e^3value technique to estimate the economic impact of green practices. However, there are a few issues for further improvement:

– **Support for differentiation between positive and negative impacts**:

In the case study, we observed that each green action can have multiple economic impacts, which may be positive (i.e., contributing to reduce costs and increase benefits) and/or negative (i.e., require investments to put the actions in place). Currently, both of the two types of economic impacts are codified by one element (i.e., economic impact) without discriminating between positive and negative effect. To assess the ROI of a strategy and align it to the organization business objectives is of course necessary to gain

a clear understanding of both positive and negative economic impacts. Therefore, in our opinion the discrimination of positive and negative impacts should be supported by the model.

– **Include references to the application of green actions**:

We noticed in the MJA document that sometimes reference documentation, a case study, or examples are given to show the application or usage of a green action. In our opinion, the information about the practice of a green action is very relevant to give the reader an instrument to get a better understanding on a green action and, therefore, should be supported by the model.

– **Support for model customization**:

The models can be further customized to a real-case scenario with actual actors and pricing and, most importantly, a real-life business model. Similarly, the energy consumption of hardware devices should be measured instead of estimated to improve the accuracy of the cost estimation.

– **Support a comparative analysis of economic implications for short- and long-term investments**:

We noticed that short-term investments (e.g., acquisition of hardware devices) and long-term costs (e.g., energy consumption) should be distinguished and analyzed in order to provide a thorough estimation of economic impact of green practices.

5.5 A Web-Based Calculator for Green ICT Practices

Our findings emphasize the need for a quantitative assessment of the economic benefits of green ICT. For this reason, we developed and released a green ICT Web calculator, a Web application able to estimate the ROI of applying a green ICT practice. In this section, we explain our approach and the implementation of the calculator.

Our first step was to collect and elicit green ICT metrics from both practice and the literature. We performed a systematic literature review [3] that resulted in 66 green metrics, classified in five categories: energy, performance, economics, utilization, and pollution. Examples of relevant metrics identified in this study are energy consumption, energy savings, and client/server energy costs.

Another study we performed [10] was more focused on metrics commonly used in industry. This study evaluated practices in four different focus areas: embedded system software, generic software, data centers/high-performance computing, and hardware. Relevant practices identified in this study were total cost of ownership (TCO), power usage effectiveness (PUE), and energy from renewable sources.

Finally, we performed a case study in partnership with a multinational telecommunication organization, where we designed a framework called "value of energy" [9]. In this framework, we defined several metrics for data management in the cloud that can be applied to calculate the economic value of data management practices. Some examples of value of energy metrics are effective power (power used to store valuable data in the organization), wasted power (power used to store

obsolete data that can be deleted or archived), and future amount of data (the expected amount of data in the future, according to data growth statistics). Using this previous work as reference, we carried out additional research to analyze and build a consistent set of metrics for green ICT.

5.5.1 Formalization of Green ICT Metrics

As a result of the aforementioned studies, we elicited several metrics to describe the environmental and economic benefits of green ICT practices. Below is a list of the relevant metrics embedded in the Web calculator according to the implemented practices. However, as shown above, this list is part of a considerable knowledge base we built that allows us to easily extend the Web-based calculator with additional practices:

- Electricity (kWh)
- Carbon emissions (g/kWh)
- Capital expenditures—CAPEX ()

  - Hardware
  - Software
  - Service

- Operational Expenditures—OPEX ()

  - Electricity cost

- Equipment lifespan (years)
- E-waste (kg)

Carbon emissions are used to estimate savings in CO_2 emissions when adopting a green ICT practice. To obtain this value, the electricity metric is multiplied by a number which represents the CO_2 grams per kWh emitted in the electricity plants of the Netherlands (normalized average), namely, 597 g/kWh. This value has been calculated by means of the measures shown in Table 5.2.

CAPEX is the amount of money (in euro) to be invested in capital expenditures before the adoption of a green ICT practice. It can be further divided into Hardware, Software, and Service (maintenance, periodic licenses) costs.

Table 5.2 Emissions and types of power plants in the Netherlands

Type of fuel	Emissions (g/kWh)	Plant power (MWe4)
Gas	430	4,500
Coal	900	3,943
Nuclear	6	485
Renewable	0	37

OPEX is the amount of money (in euro) to be spent on operational expenditures during the adoption of a green ICT practice. This includes electricity cost that is derived from the electricity metric according to the relative energy tariff.

Equipment lifespan is used to express how long IT devices can be functional and, thus, when they should be replaced. The replacement will involve a periodical payment.

E-waste expresses the quantity of IT material that has to be disposed of. This metric has an inverse relationship with the equipment lifespan metric.

5.5.2   The Application

The *Web calculator*[4] is an online Web application that helps decision makers calculate the cost benefits of green ICT practices. To ease this achievement, the Web calculator has been partially integrated with our online library of *green ICT practices*,[5] thus allowing users to immediately calculate an economic estimation of applying a specific practice in their organizations. Up to now, the thin client practice from the green ICT library has been implemented in the Web calculator. As part of our future work, additional practices will also be modeled and integrated.

The Web calculator has been developed as a PHP Web application. Along with the PHP programming language, CSS style sheets and JavaScript are used to improve user interface in terms of usability and aesthetics.

The Web calculator consists of two modules: model management and selection processing of models. The first module offers features to find, upload, and retrieve models. The second module is divided into four substeps in which users select previously loaded models, configure them, and calculate investments and expenses according to the modeled practice. After the execution, charts of the results are available to users to provide visualization of the economic benefits.

Figure 5.5 shows the execution flow of the application. The various phases are described in detail in the remainder of this section.
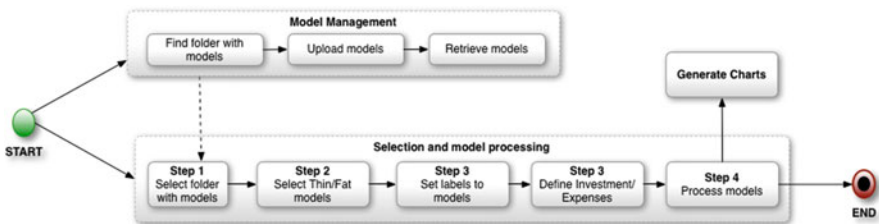
Fig. 5.5   Usage diagram of the Web calculator
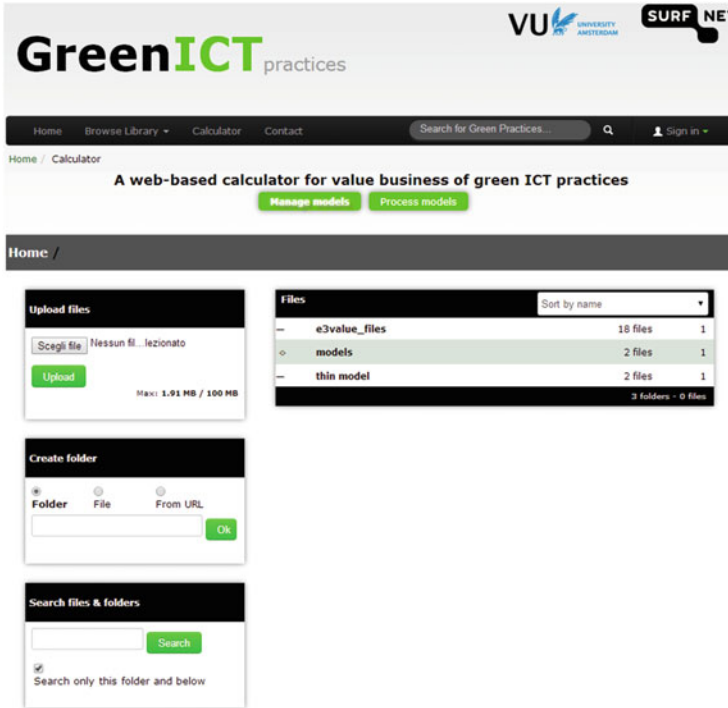
Fig. 5.6 Model management interface

5.5.2.1   Features of the Calculator

Model Management

Through the e^3value modeling tool, it is possible to generate Excel spreadsheets that implement the models for green ICT practices. The file manager of the Web calculator helps users to organize the different models in a meaningful structure for easier manipulation. In Fig. 5.6, we show a screenshot of the model management interface. In the list below, you can find all the available file management operations:

- View and sort files/folders
- Upload new files
- Create files/folders
- Search in files/folders

Thanks to this feature, the Web calculator is easily extensible, allowing users to estimate the economic effects of new green ICT practices. However, the Excel spreadsheets have to be generated from the e^3value modeling tool in order to be properly parsed by the Web calculator.

Selection and Processing of Models

Models can be selected and configured from within the Web calculator interface. Users are able to give each model a meaningful label for comparison, and, more importantly, they can customize the *value exchanges*, which represent the actions between the actors of the model. For example, the electricity provider (actor) exchanges electricity for money with the company (actor). These value exchanges can be either an investment or a monthly expense depending on whether exchanges occur once or every month. In the previous example, electricity is a monthly expense because companies calculate electricity as a monthly cost. After defining these properties, users can process the models and visualize the results through charts.

The main feature of the Web calculator is model execution. Before the execution, users can customize the parameters related to the green ICT practice selected. For the desktop virtualization practice, such parameters can be the number of clients/servers, the cost of the equipment, the electricity consumption, and the cost of electricity. This allows users to tune the calculation of the economic benefits of the practice according to their specific situation.

The Web calculator loads the user-defined parameters into the equations of the model and calculates the results. Figure 5.7 shows a screenshot of the model execution interface. By default, the application shows two types of potential savings: one-time savings and monthly savings. One-time savings include the investment savings, while monthly savings include electricity consumption (in kWh), CO_2 emissions, and other operational costs.
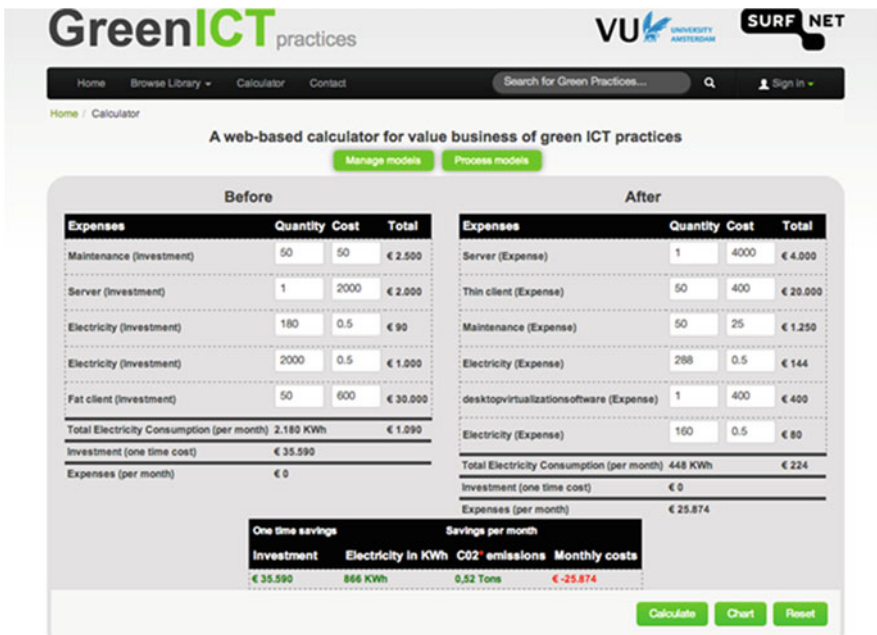
Fig. 5.7 Processed models
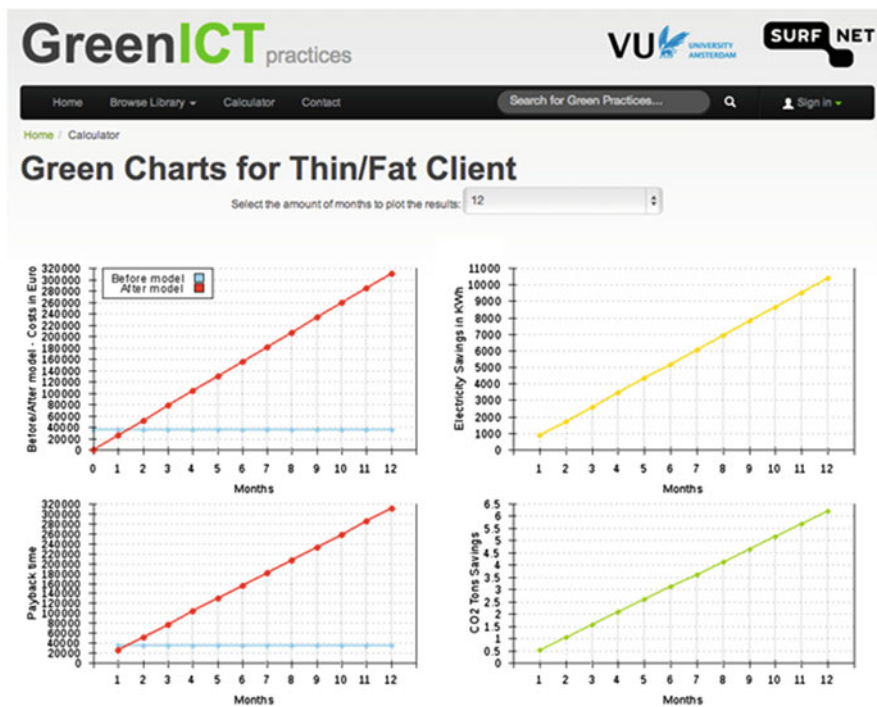
Fig. 5.8 Visualization of results

Chart Generation

The Web calculator also features the possibility to visualize the results of the model processing phase. Figure 5.8 gives an example of the Chart Generation feature for the desktop virtualization practice. Charts display the calculated metrics for a given period of time which by default is 12 months. The charts show the difference in terms of monthly costs and payback time between the AS-IS (fat clients) and TO-BE (thin clients) situation.

Moreover, two additional visualizations demonstrate the electricity savings in kWh and the CO_2 emission savings. For example, in Fig. 5.8, it can be observed how applying the desktop virtualization practice reduces the costs (comparing the AS-IS and TO-BE lines in the top left chart) and shortens the payback time (bottom left chart). The right-side charts show the progression of CO_2 and electricity savings: for example, after 5 months of applying the desktop virtualization practice, we estimate to reduce CO_2 emissions by more than 2.5 tons. The period of time can be modified by the user, and the charts are automatically updated.

5.6   Conclusions

To reduce energy costs and contribute to global environmental goals, organizations consider green ICT practices increasingly often. Sometimes they even add them as part of their organizational strategies. However, experience shows that if green ICT practices are not in line with business and organization strategies, they are easily neglected or withdrawn in times of crisis.

To aid organizations in the selection of green ICT practices and aligning them to their business strategies, we applied the e^3value technique to estimate the economic benefits of applying green ICT practices. Such economic benefits can be influenced by various aspects including investment cost, size of companies, pricing, and duration. The application of the e^3value technique allows to perform trade-off analysis to select among different green ICT practices, particularly from an economic perspective. When costs are quantified and ROI is estimated, informed decisions can be made before actual investments.

Most green ICT practices do not yet particularly address software-specific aspects. Our approach allows to clearly separate the role of software (e.g., virtualization software in the experiment presented here) from the role of other factors and calculate its direct and indirect economic impact.

Our contribution is twofold: First of all, we provide a rationale for promoting sustainability in organizations and a stimulus to identify and formalize new green ICT practices for achieving more profitable results. Secondly, we provide an educational tool to explain environmental benefits of regreening ICT, as well as the relation with economic investments, gains, and ROI.

The Web calculator represents a first step towards calculating the economic benefits of green ICT: it provides estimations of the economic impact of green ICT practices, allowing organizations to reuse these practices through informed decisions. Our future work will be devoted to empirically validate the estimations of the Web calculator through industrial case studies in collaboration with our partners SURF[6] and Green IT Amsterdam.[7]

---

[6] http://www.surf.nl/en/about-surf/subsidiaries/surfnet

[7] www.greenitamsterdam.nl/

References

1. Asif M, Muneer T (2007) Energy supply, its demand and security issues for developed and emerging economies. Renew Sustain Energy Rev 11(7):1388–1413. doi:10.1016/j.rser.2005.12.004
2. Boudreau M, Chen A, Huber M (2008) Green IS: building sustainable business practices. In: Watson RT (ed) Information systems: a global text. Global Text Project, Athens, GA
3. Bozzelli P, Gu Q, Lago P (2013) A systematic literature review on green software metrics. Tech. rep., VU University Amsterdam
4. Corbett J (2010) Unearthing the value of Green IT. In: ICIS, p 198. Association for Information Systems
5. Davidson E, Vaast E, Wang P (2011) The greening of IT: how discourse informs IT sustainability innovation. In: Proceedings of conference on commerce and enterprise computing, pp 421–427. IEEE
6. Gordijn J, Akkermans H (2001) E3-value: design and evaluation of e-business models. IEEE Intell Syst 16(4):11–17
7. Gordijn J, Yu E, van der Raadt B (2006) E-service design using i* and e3value modeling. IEEE Software 23(3):26–33
8. Gu Q, Lago P (2013) Estimating the economic value of reusable green ICT practices. In: Safe and secure software reuse, Lecture Notes in Computer Science, vol 7925. Springer, Berlin, pp 315–325
9. Gu Q, Lago P, Potenza S (2013) Delegating data management to the cloud: a case study in a telecommunication company. In: International symposium on the maintenance and evolution of service-oriented and cloud-based systems (MESOCA), vol 7, pp 56–63. IEEE Computer Society
10. Gude S, Lago P (2010) A survey of Green IT – metrics to express greenness in the IT industry. Tech. rep., VU University, Amsterdam
11. Harmon R, Demirkan H, Auseklis N, Reinoso M (2010) From green computing to sustainable IT: Developing a sustainable service orientation. In: 2010 43rd Hawaii international conference on system sciences (HICSS), pp 1–10
12. Henkel M, Perjons E (2009) Ways to create better value models. In: Proceedings of the 3rd workshop on value modeling and business ontologies (VMBO09)
13. Kurp P (2008) Green computing. Commun ACM 51(10):11–13
14. Lago P, Jansen T (2011) Creating environmental awareness in service oriented software engineering. In: Maximilien E, Rossi G, Yuan ST, Ludwig H, Fantinato M (eds) Service-oriented computing, vol 6568, Lecture Notes in Computer Science. Springer, Berlin, pp 181–186
15. Liu L, Wang H, Liu X, Jin X, He W, Wang Q, Chen Y (2009) GreenCloud: a new architecture for green data center. In: Proceedings of the 6th international conference industry session on autonomic computing and communications industry session. ACM, pp 29–38
16. Mattern F, Staake T, Weiss M (2010) ICT for green: how computers can help us to conserve energy. In: Proceedings of the 1st international conference on energy-efficient computing and networking, e-energy '10. ACM, New York, pp 1–10. doi:10.1145/1791314.1791316
17. Mingay S (2007) Green ICT: a new industry shockwave. Tech. rep., Gartner. URL http://www.ictliteracy.info/rf.pdf/Gartneron GreenIT.pdf
18. Mitchell RL (2008) Get up to speed on Green IT. Tech. rep., Computerworld
19. Morrison E, Ghose A, Dam H, Hinge K, Hoesch-Klohe K (2011) Strategic alignment of business processes. In: 7th international workshop on engineering service-oriented applications. Springer, Berlin
20. Murugesan S (2008) Harnessing Green IT: principles and practices. IT Prof 10:24–33. doi:10.1109/MITP.2008.10
21. Omer AM (2008) Energy, environment and sustainable development. Renew Sustain Energy Rev 12(9):2265–2300. doi:10.1016/j.rser.2007.05.001

22. Park JK, Cho JY, Shim YH, Kim SJ, Lee BG (2009) A proposed framework for improving IT utilization in the energy industry. World Acad Sci Eng Tech 58:387–393
23. Sarkar P, Young L (2009) Managerial attitudes towards Green IT. An explorative study of policy drivers. In: Proceedings of PACIS, pp 1–14
24. Vereecken W, Van Heddeghem W, Colle D, Pickavet M, Demeester P (2010) Overall ICT footprint and green communication technologies. In: Proceedings of the 4th international symposium on communications, control and signal (ISCCSP). IEEE, pp 1–6

Chapter 6
Green Software Quality Factors

Juha Taina and Simo Mäkinen

6.1 Introduction

Software quality and quality software have been leading factors since 1968 when the first software engineering conference was held in Germany [20]. In almost 50 years, the software engineering community has got a very good and realistic view of what is quality in software and software engineering. We now know how to build, maintain, and execute quality software.

More than 10 years after the first software engineering conference, the first important use of the term *sustainable development* was presented [11]. At that time, the International Union for Conservation of Nature (IUCN) published the report 'World Conservation Strategy: Living Resource Conservation for Sustainable Development' [12]. In 1987, the World Commission on Environment and Development (WCED) gave perhaps the most commonly used definition for sustainable development: 'the needs of the present without compromising the ability of future generations to meet their needs' [5]. Since then, sustainable development has gained growing interest among researchers, politicians, economists, environmentalists, and other interest groups.

A few years after the WCED report, the first attempt to combine computer technology and sustainable development was introduced. In 1992, a voluntary programme called Energy Start started. It is an umbrella of voluntary programmes aimed at reducing climate change by promoting the development and use of energy-efficient equipment [3].

J. Taina (✉)
Faculty of Science, University of Helsinki, Helsinki, Finland
e-mail: Juha.Taina@helsinki.fi

S. Mäkinen
Department of Computer Science, University of Helsinki, Helsinki, Finland
e-mail: Simo.V.Makinen@helsinki.fi

The idea of considering sustainable software and software engineering alone without the hardware aspects was not fully acknowledged until the year 2010. At that time, new terms *green and sustainable software* (green software or sustainable software depending on the view) and *green and sustainable software engineering* (green software engineering) started to emerge.

The field of green software and green software engineering is still young. Naumann et al. presented a definition for green and sustainable software engineering as recently as 2013 [19]:

> Green and Sustainable Software is software, whose direct and indirect negative impacts on economy, society, human beings, and environment that result from development, deployment, and usage of the software are minimal and/or which has a positive effect on sustainable development [. . .] Green and Sustainable Software Engineering should produce Green and Sustainable Software in a sustainable way.

While the definition states what green software is, it does not mention the quality of green software. Since we already have a working software quality model, it is worth to expand it to support green software quality.

For a good-quality model for green and sustainable software, we need to understand the properties of green software. Two viewpoints help to define this: green ICT and software engineering for sustainable development.

Green ICT is the study and practice of using computing resources efficiently [14]. It is a huge field that includes everything in a software system life cycle from hardware manufacturing to minimising computer-related waste.

Resource usage is a remarkable issue in green ICT. For example, one supercomputer can consume enough energy to power nearly 10,000 homes and costs ten million dollars a year to operate [10]. Each PC in use generates about a ton of carbon dioxide a year [17].

At the moment, green ICT concentrates mostly on minimising hardware resource usage. There is still much to be done there, and savings can be remarkable. For example, in a moderate climate air-cooling a data centre constitutes about 30 % of the total energy it consumes [23]. According to IBM, 'up to 50 % of an average air-cooled data center energy consumption and carbon footprint today is not caused by computing but by powering the necessary cooling systems to keep the processors from overheating'.[1] In the United States in 2009, approximately 25 % of TVs, computer products, and cell phones that were ready for end-of-life management were collected for recycling. Cell phones were recycled at a rate of approximately 8 %.[2]

In 2007, Gartner released the statistic that 2 % of global carbon emissions were from the ICT sector [9]. While this is a huge number, it is still 'only' 2 %. For example, it is estimated that the manufacture of Portland cement accounts for 5 % of all human-generated greenhouse gas emissions [2]. In a recent commentary, Carlos Ghosn, CEO of the Renault-Nissan Alliance, claimed that 23 % of

---

[1] http://www.zurich.ibm.com/st/server/zeroemission.html

[2] http://www.epa.gov/osw/conserve/materials/ecycling/faq.html

worldwide greenhouse gas emissions come from the auto industry.[3] Eventually, we need to find a balance between carbon emission and carbon absorption. Reducing ICT emissions is not enough to reach the goal.

A larger field called *software engineering for sustainable development* (SE for SD) addresses issues and questions of where and how software and software engineering can help sustainable development. While green ICT is included in SE for SD, it is not limited to green ICT.

In a workshop 'Software Engineering and Climate Change', 2009, the participants defined that SE for SD consists of at least the following[4]:

1. *Software support for green education*: Use software to support education and general knowledge about sustainable development and climate change.
2. *Green metrics and decision support*: Define software processes and tools for environment-friendly software design, implementation, usage, and disposal.
3. *Lower IT energy consumption*: Allow software to support or be part of green ICT.
4. *Support for better climate and environment models*: Let environmental scientists do their research with better software.

Since 2009, the requirements for SE for SD have grown substantially. Now SE for SD is considered to support all sustainable development. From the SE for SD point of view, good software helps to reduce waste and resource requirements while bad software increases them. Sometimes software can be downright ugly: badly written, difficult to use, full of unwanted features, and resource intensive. On the other hand, we can have elegant software that not only reduces resource requirements and minimises waste but also supports sustainable development and sustainable development knowledge.

6.2 Green Quality

Common conceptions of software quality include the notion that software should meet its explicit requirements, serving the needs of the customers and users appropriately with the expectation that the development of software lives up to the professional standards of the field [8]. Properly engineered, high-quality software fulfils these requirements. Software which is of high quality in these terms might or might not embed green and sustainable values. It depends on the specified software requirements, the preferences of the customers or users, and the current state of professional standards. While software quality can be valued in varied ways, quality indicators from products, processes, and resources hint at the relative

---

[3] http://www.project-syndicate.org/commentary/carlos-ghosn-describes-what-is-needed-to-improve-automobiles-safety-sustainability-and-affordability

[4] http://www.cs.toronto.edu/wsrcc/WSRCC1/index.html

quality of software. *Green quality*, however, is only indicated by some quality indicators, either indirectly or directly.

Software quality can be characterised on several levels ranging from higher-level abstractions to more concrete, lower-level characteristics which can be measured. Quality factors [7] or quality characteristics [1] are meaningful software-related properties like efficiency, reliability, and usability which cannot be measured as such due to their general nature [7]. Generally, quality attributes are properties from products, processes, and resources that are of interest in terms of software being developed or operated.

Quality attributes can be either internal, in which case the attribute does not rely on its environment as such, or external, when the effect of the environment on the attribute is strong and affects the result of the observation or monitoring of the attribute [7]. Efficiency, reliability, and usability are all external attributes: efficiency of a software product is reliant on the hardware it is executed on and on the resource utilisation of the system; reliability depends not only on the internal consistency of software but also on the way software is being operated in a software system; and perceptions of usability might differ between various user groups. Internal attributes such as size, however, can be measured in a more consistent manner without much interference from the external environment. It would be unlikely that the size of a code product measured in a specific manner would change given that the code remains unchanged.

Quality factors or characteristics are different internal or external quality attributes which are considered central to software development or its operation. The factors or characteristics need to be broken down in order to understand which dimension of the attribute or property is being addressed; these lower-level constructs are called quality criteria [7] or quality sub-characteristics [1]. Quality criteria or sub-characteristics are specific enough so that they can be measured, unless further division is required. Measurement leads to metrics or measures which characterise the quality criteria or the sub-characteristics with a value [1, 7]. For instance, error tolerance is a quality criterion for the quality factor of reliability, and its measure could be mean time between failures (MTBF) that is measured by the time between observed failures in a software system. Similarly, the length of a program is a quality criterion for size, and lines of code would be one way to measure it.

6.3   Quality Models

Quality models group quality factors together, forming a set of quality factors that address some quality concerns. Software quality models such as McCall's quality model [16] and the ISO standard quality model [1] are relatively generic. While several quality factors in these models are associated with resource efficiency, sustainability is not considered as a substantial quality factor.

Both of the aforementioned quality models focus on product or system quality. The models have quality factors for dimensions that deal with quality from different perspectives. For instance, there are factors that relate to how reliable and secure the product is and how easy it is to use. Factors in the models are also related to how products can be tested and maintained later in the product's life cycle. Since there are times when software systems need to interact with each other and transfer data from one system to another, compatibility and interoperability with other systems are listed as a relevant quality factor. The portability quality factor is important in situations where the product itself needs to be converted to work in another environment or transferred to a new system. From all the quality factors in these two models, efficiency is one of the most relevant factors for green quality and sustainability. Efficiency signifies the product's ability to use the available resources in a non-wasteful manner.

Besides the product quality model, the new ISO software quality standard [1] contains a separate quality model for quality in use. The model differs from the system and product quality model in that it describes potential effects the whole software system might have on its users and the environment. The effects include estimations from the users of how satisfied they are with the system and how effective and efficient the system is for the purpose it was designed for. There is a quality factor called *freedom from risk* in the quality in use model that has a sub-characteristic for environmental risk mitigation: the description of the risk factor implies that a system should limit its negative environmental effects that result from the use of the software system. Sustainability of software systems can be inferred from environmental risk mitigation as systems which generate much waste can be considered more harmful to the environment than those systems that are better at recycling. Thus, the quality in use model does acknowledge the existence of environmental factors, but sustainability is not the governing aspect in the generic quality model.

Software quality models list a wide array of quality factors which might be suitable when developing, maintaining, or using software. Whether the quality factors actually are relevant for a particular software project or a software system, or its users, depends on the instilled ideals and priorities that have been set by the customers, developers, and other involved parties. While the existing software quality models do not rule out environmental factors, green quality or sustainable software development is not specifically encouraged by the models.

The software quality model gives basis for a simple definition for quality software:

Quality software is software that supports a predefined set of measurable software quality factors.

The current software quality model is simple and elegant. It does not restrict software quality to certain predefined characteristics but allows all kinds of factors to be present. In fact, quite often software quality factors are in conflict with each other. A software designer has to decide what factors and at what level he or she wants to support in his or her software.

The current quality factor model consists of factors that improve software end-user experience and simplify software engineering. The factors have mostly been

independent of the software problem domain. (Problem domain is the scope where software is executed and its results are exploited.) However, green software and green software engineering are strongly problem domain dependent. They are connected to sustainability and sustainable development. Due to this, green software quality needs *green quality factors*: factors that define how software supports sustainable development.

6.3.1 Software Quality Measurement

Software quality can be measured in many respects with a multitude of methods. The quality factors in quality models can give a sense of direction as to which kind of elements the measurement should focus on. The metrics yielded by software measurement are associated to some quality concern, given that the metrics themselves represent the phenomenon in question adequately.

Software quality measurement can target different products, processes, and resources. For instance, in a software development project, the product qualities of code components might be of interest. It is possible to measure the complexity of code components [15] by analysing the software source code; resolving the component relationships and the specialisation degree of components can be done from the same source code [6]. These metrics can be related to the quality factor of maintainability, but they might also help to predict the probability of the occurrence of defects in software [4]. Similarly, by analysing the source code, it can be measured how well the developers have followed professional coding conventions while programming which might affect the maintainability of code products.

At first, product quality metrics such as these do not seem connected to green quality. However, reducing defects can lead to the creation of less waste. The time of the users and developers is saved if the failure of software for a specific feature never occurs in the field. Less complex software with fewer intercomponent relationships could be easier to maintain which subsequently makes future maintenance work on software easier, again reducing the waste induced by imperfect programming.

When executing the program code of a software, we can learn more about the qualities of the product. Software performance can be measured when software is running which allows profiling of the application, giving an idea where most of the computation time is spent. Good, beautiful software is sufficiently fast in performing its tasks. Slow software spends more time than necessary when, for example, going through data structures in the memory for finding the correct result. Improvements in performance lead to gains in the efficiency of a program and can result in waste reduction.

Program execution is suitable for dynamic program analysis, measuring quality factors such as performance, but it can also be used to measure the completeness of testing. Structural testing is possible when automated, machine-executed tests have been programmed to test the functionality of a program [21]. Thus, structural

testing allows to determine which parts of the program have been tested and which parts are currently not covered by any automated test. This type of coverage metrics is represented by the ratio between the amount of code tested and the total amount of code in a program. Coverage relates to the quality factors of maintainability and its sub-characteristic testability, but striving for high coverage can also be based on the desire to prevent defects in software. Fewer defects and better quality can be the rationale for structural testing, and the positive impact regarding green quality is the reduced effort and waste related to fixing the defects. Preparing an extensive suite of automated tests for structural testing might still increase the development effort in the early stages of development, though.

Regarding testing or any other process for that matter, process effort is a key metric in software engineering. Effort indicates the time it takes to perform an activity either by a human or a machine. Effort of testing could be measured by observing a person who is performing testing on a software system or one of the product components and measuring the time it took for the person to finish the testing stages. Equivalently, the effort for a machine could be the time it used for the execution of the process. All software development activities take time: gathering the requirements, programming, testing, maintenance, and releasing new software versions require effort. Knowing the effort of processes helps to point out where most of the time is spent, which might help to identify resource-intensive activities and achieve greater resource efficiency through analysis of activities. Green quality implies that resources are saved where possible. Process metrics, as other product and resource metrics, can be of use with green and sustainable software development.

6.4   Green Factor Motivation

It is clear that software can and will help in our goal to support sustainable development. New and improved software will play an ever-increasing role in control systems, optimising algorithms and education, among other things. Software is needed on all current problem domains, and totally new problem domains are constantly being introduced. In the next few years, we need to design, write, and test millions of software requirements and billion lines of code for common problem domains with new sustainable views and also for totally new problem domains.

Creating and managing all new and improved software require resources. We need more energy, hardware, supporting software, and peopleware for software creation, management, maintenance, and disposal. At the same time, we need to minimise resource usage and waste so that future generations can also benefit our software in environmental, economic, and social well-being. Thus, we need to be very efficient in software engineering and use our valuable resources with maximum efficiency.

In order to understand the characteristics of sustainable software, we first need to understand sustainable development. While it is a common term to everyone, it has several interpretations and definitions that are not necessarily compatible with each other. Sustainability implies our wishes to keep or improve our standards of living, to maintain economic growth, to give every human being equal rights and possibilities, to use our current material and energy resources wisely and with minimum waste, to save the environment of our planet, and more. No wonder that people are confused with the term.

Hopwood, Mellor, and O'Brien [11] have written a detailed summary of current trends in sustainable development. We specialise their model of sustainable development to software use, and we encourage everyone to read the original article for a detailed background of trends in sustainable development.

The several incompatible views of sustainable development are at the best confusing and at the worst negatively affecting public views of sustainability. In order to clarify the subject, Hopwood et al. use a mapping methodology based on combining environmental and socio-economic issues. The result is a two-dimensional mapping that shows how and how much various actors see sustainability to be related to human equality and environment.

We present a summary of the mapping in a matrix in Table 6.1. The matrix shows what kind of sustainable development is considered and a typical example of this kind of an approach. The equality axis (bottom-up) implies level of importance of human well-being and equality. The environmental axis (left–right) covers the priority of the environment. Techno-centred implies approaches where new and improved technology is enough for sustainable development. Eco-centred implies approaches where current technology is not considered a working solution for sustainable development.

The higher the concern of equality and anxiety about the environment are, the more extreme methods are suggested in the name of sustainable development. Most governments and major environmental and socio-economical organisations are close to the middle of the mapping. Their view to sustainable development is that it can be achieved by improving current methods and techniques. According to this view, we can solve our socio-economical and environmental problems with our current technology, economy, and political models—without sacrificing reached living standards.

Table 6.1 Approaches of sustainable development

High concern of equality Little environmental concern – Early communism	High concern of equality Techno-centred – Social reform	High concern of equality Eco-centred – Eco-socialism
Concern of equality Little environmental concern – Arab Spring	Concern of equality Techno-centred – EU environmental policy	Concern of equality Eco-centred – The Club of Rome
Little concern of equality Little environmental concern – Early capitalism	Little concern of equality Techno-centred concern – Natural resource management	Little concern of equality Eco-centred concern – Extreme environmentalism

Groups that in the mapping are somewhat right or top from the middle have a stronger view where they do not trust current methods to be sufficient. According to such groups, a fundamental reform to current methods and techniques is required to achieve sustainability. Finally, groups at extreme right or top consider that no reform is enough, but we need to completely redefine our socio-ecological, economical, and political models to achieve sustainability.

So, when we say that software will support sustainable development, whose sustainable development is it supporting? This is an interesting and unsolvable question that affects software quality. For example, let us assume that we are more concerned of the environment than social issues. Then software that supports extreme environmentalism is by definition greener than software that supports traditional government policy of technology-centred sustainability. For many people, this would sound illogical and downright wrong.

A software quality model needs to be objective, so it is not our task to define what kind of sustainable development is acceptable. The nature of the green software definition sets green software and green software engineering close to the centre of sustainable development mapping. It is technology-centred and does not have high or low concern of equality.

The previous analysis of sustainable development and the earlier definition for green software create a starting point to define our green factor model. Let us start with the sustainable development matrix.

The two major forces of sustainable development are environment preservation for future generations and socio-economical equality. While the emphasis of the two forces varies in different definitions, they are practically always present.

In the definition of green and sustainable software, it is mentioned that 'direct and indirect negative impacts on economy, society, human beings and environment [..] are minimal'. The definition already acknowledges the environmental and socio-economical aspects, so any green factor model should also acknowledge them.

At a large scale, solving the environmental problems and preserving environment for future generations lead to two requirements:

1. We need to recycle all nonrenewable resources.
2. We need to get rid of all waste.

These are the ultimate solutions to environmental problems and in fact formulate the same requirement from two different views. For example, global warming is a waste problem. We dump too much greenhouse gases into the atmosphere. The lack of pure fresh water is both a waste problem and a recycling problem. Water is polluted, and nonrenewable freshwater sources are globally used.

While the ultimate requirements eventually need to be fulfilled, implementing them is currently beyond our abilities (at least without severely breaking socio-economical equality and well-being). We want to start with somewhat easier requirements that may eventually lead to satisfied ultimate solutions. We formulate new requirements as follows:

1. We need to be resource efficient.
2. We need to minimise waste.
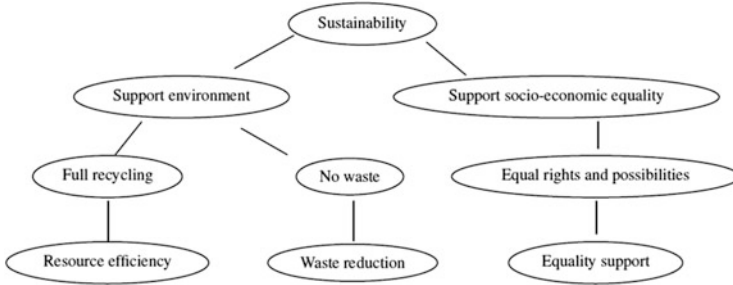
Fig. 6.1 Sustainable development pattern

Resource recycling simplifies resource efficiency since it is usually more efficient to reuse resources than produce new ones. Minimising waste is the first and necessary step towards getting rid of waste.

Socio-economical equality gives importance to human well-being and equality. It is beyond this chapter—and our knowledge—to define exact requirements for this. We only state that according to socio-economical equality, every human being should have equal rights, obligations, and possibilities. This leads to the following requirement:

1. We need to support equal rights and possibilities.

Thus, green software and green software engineering have three goals (see Fig. 6.1):

- Resource efficiency
- Waste reduction
- Equality support

The level of support depends on how well one wants to support environment and socio-economic equality. As such, the model is compatible with the earlier sustainable development analysis although a natural starting point is to start close to the middle of the matrix in Table 6.1.

6.5   Green Software Factor Model

From the definition of green software and our previous analysis, we can extract two properties for green software factors (green factors): (1) object of support and (2) type of support.

The first property is *object of support*. It defines who or what is the object of sustainable green software actions. We identify three cases:

1. *The object is software*: We affect software architecture and algorithms with sustainable processes. This is a 'what software gets' case. It is a view to green factors. Green software engineering is based on this case.
2. *The object is a software system*: Software affects its software system via CPU operations. This is a 'what software does' case. It is a software execution view. *Green ICT* (software-wise) is based on this case.
3. *The object is a stakeholder*: Software affects its stakeholders via its outputs to the software system. This is a 'what software delivers' case. It is a software system and client view. Sustainable software is based on this case.

All three cases are important and have a different view to quality. Quality is in human actions to software, in software results, and in software result effects. A good software factor model supports all views.

The second property is *type of support*. It defines what direct or indirect effects green software has on sustainability. The type of support comes directly from the previous analysis of sustainability. Again, we identify three cases:

- *Support for resource efficiency*: It defines how green software directly or indirectly supports efficient concrete, abstract, and human resource allocations and use. This is a 'how do we get full potential' support. It is related to all three objects of support.
- *Support for waste reduction*: It defines how green software directly or indirectly minimises waste. This is a 'how do we not waste useful resources' support. It is related to all three objects of support.
- *Support for social equality*: It defines how green software indirectly supports socio-economical sustainability. This is a 'how do we help others' support. It is related to software and software system client objects of support.

Note that support for social equality is not related to the case where a software system is the object. Current software systems do not need social equality.

The previous analysis translates directly to a green factor model. The type of support translates to software factors since factors define how software behaves. The object of support translates to factor goals since goals define objects for behaviour. This leads to three factors: *resource effectiveness*, *triftness*, and *social sustainability* (Fig. 6.2).

Resource effectiveness is a common factor for all aspects that evaluate software resource efficiency in software engineering, software execution, and software stakeholder processes. In software engineering, resource efficiency is related to software life cycle, including software design, management, maintenance, and disposal. Software execution resource efficiency is related to software execution and software platform usage. Software client process resource efficiency is related to how software stakeholders benefit from software and its software system.

Triftness is a common factor for all aspects that evaluate how software reduces waste. Again, *triftness* can be a factor of software engineering, software execution, and software usage.
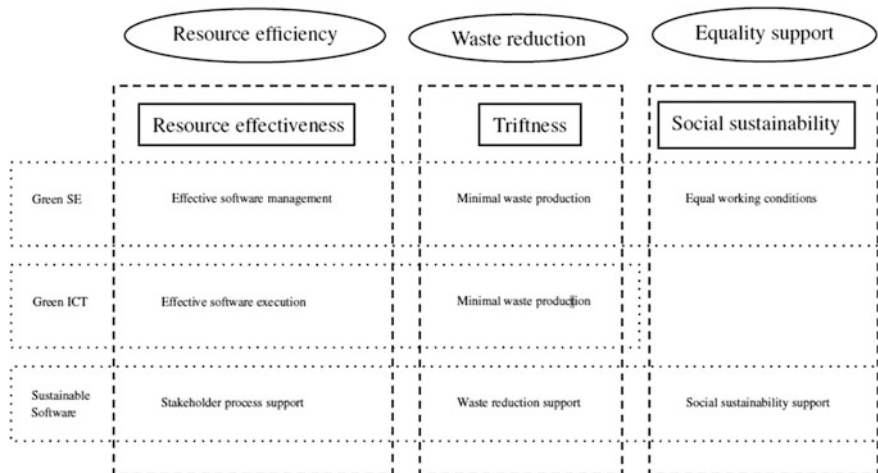
Fig. 6.2 High-level factor model

Social sustainability is a common factor for all aspects that evaluate how software supports social equality. *Social Sustainability* is a common factor for all aspects that evaluate how software supports social equality. Social Sustainability can be a factor of software engineering and software benefit.

Resource effectiveness and *triftness* are related but are not the same factor. We can be effective on different ways and create different types of waste. For instance, time is a valuable resource. We can minimise this resource in software development by hiring the best engineers all over the world. However, the solution probably does not minimise waste since the hired engineers most likely need to commute a lot and over long distances in order to get software done.

At a very long time scale, *social sustainability* can probably be reduced to *resource effectiveness* and *triftness*. However, currently it is a separate factor. For example, we could have a software system that helps in Third World education. It clearly supports social equality but is not necessarily resource efficient or does not necessarily minimise waste. We could also have a software system that helps in building working sanitary systems (a huge issue in social equality). Eventually, the sanitary systems will reduce waste but not at the moment and not directly due to our software.

With the three factors, we can give a factor-based definition for green software:

Green software is a software that follows *resource effectiveness* and *triftness* in execution; supports its stakeholders' *resource effectiveness*, *triftness*, and *social sustainability*; and is built with support for *resource effectiveness*, *triftness*, and *social sustainability*.

The definition is compatible with the definition of green and sustainable software by Naumann et al. [19] that we quoted earlier.

With the three factors, we can define how green software and green software engineering behave. Green software is developed and managed with sustainable

methods and executed with maximum efficiency and minimum waste and supports sustainable development.

The definition and related factors are abstract, and as such they do not give much support when evaluating green software. For example, let us have a very well-written spamming software. It follows *resource effectiveness* and *triftness* in management and execution. It probably wastes resources of some of its stakeholders, but it is a matter of opinion of how much. It can even claim to support social equality since it sends spam to everyone regardless of their social status. Hence, someone could easily claim that it is green software.

The problem in the previous example is in the generality of the definition. Spamming software may be efficient in its own execution, but it clearly steals common resources such as stakeholders' network and permanent storage resources. It also affects its stakeholders' resource efficiency by at least forcing them to somehow filter the spam. All these details and more are hidden in the definitions. Due to this generality, we need to divide the factors into smaller sub-factors.

Fortunately, an excellent green software quality model called GREENSOFT model has already been proposed [13, 18]. The model is defined at four levels: software product life cycle, criteria and metrics that represent sustainability aspects directly and indirectly related to the software product, procedure models for different phases, and tools and recommendations for action [13].

From our point of view, the most important level of the GREENSOFT model is the criteria and metrics level and its quality model for green and sustainable software. It defines direct, indirect, and common criteria and metrics for green and sustainable software [13]. The model is compatible with our earlier factor definitions and is a good start to defining green software sub-factors.

By combining the GREENSOFT model, our sustainable development analysis and green factor requirements, we get a three-level green factor model: (1) a software object layer (software management layer) that supports software, (2) a software system object layer (software execution layer) that supports software execution, and (3) a software stakeholder layer (software system layer) that supports software end users and their stakeholders. Each layer defines relevant sub-factors.

The three-level layered model is sufficient for a green factor model. However, the layers are large and complex. Due to this, we further divide the software execution layer into two layers: (1) an execution layer and (2) a platform layer. We also divide the software system layer into two smaller layers: (1) an application layer and (2) a system layer. Finally, we add a semantic layer on top of the model: a problem domain layer.

The division to execution and platform layer is due to different resource requirements. We define the execution layer to deal with resources that are directly related to software, such as CPU, peripherals, and main memory. The platform layer, on the other hand, deals with resources that are platform specific and possibly have a relationship with resources on other platforms. Network, cloud, and database resources are typical to the platform layer.

The division to application and system layers is related to how we see software. At the application layer, we consider software a stand-alone object whose factors

Table 6.2 Summary of layered sub-factors

	Resource effectiveness	Triftness	Social sustainability
Management layer	Feasibility	Minimality	–
Execution layer	Execution efficiency	Utility	–
Platform layer	Service efficiency	Service utility	–
Application layer	Reflectivity	Reflectivity	Support for society
System layer	Collaboration	Reduction	Production sustainability
Problem domain layer	Beauty	Beauty	Beauty

we can evaluate. At the system layer, we consider software to be a part of a software system. The difference is subtle but meaningful. It is relatively straightforward to calculate how a software system uses resources and generates waste, but it is much more complicated to evaluate what is the role of software in it. At the application layer, we consider software and its effects alone.

Finally, the problem domain layer is for evaluating factors that are specific to a certain problem domain. It is often difficult to compare software and software systems from separate problem domains, but it is possible to compare them within a common problem domain as long as the problem domain is well defined.

With the layer model, we can define suitable sub-factors for various approaches of green software. A summary of the sub-factors is provided in Table 6.2. We explain each layer in more detail below.

6.5.1   Management Layer

The first and lowest layer in our model is the management layer. It defines how external resources indirectly affect the software life cycle. It deals with sub-factors that are related to software engineering and management during the software life cycle. Software itself is an object of the layer activities.

The management layer is a process layer. Everything in the layer is related to software processes and software engineering. Since software process researchers and practitioners have created excellent models and processes to effectively create new software, the factors and their definitions here are closely related to what has already been researched. After all, efficient software production includes resource efficiency which itself is a core component in green software engineering.

The management layer is a very practical layer. The sub-factors and metrics here are all process related and deal with green issues in software engineering. Nevertheless, the layer belongs to green software engineering since we want our green software to be produced and maintained with sustainable methods.

The two green ICT factors, *resource effectiveness* and *resource effectiveness and triftness*, are relevant to the management layer. All software processes need to be resource efficient and minimise waste.

In our earlier work, we defined the factor *feasibility* [25]. It states how resource efficient it is to develop, maintain, and dispose software. *Feasibility* is a sub-factor *of resource effectiveness* and clearly related to the management layer. Hence, it is a good management layer sub-factor of *resource effectiveness*.

A good sub-factor of *triftness* at the management layer is *minimality*. It states how much of the resources used in *feasibility*-related tasks are directly related to tasks that have direct value to software end users. In other words, the more we have *minimality*, the more we concentrate on developing or managing software functionality that software end users find valuable.

Both *feasibility* and *minimality* are still very large factors that cover all stages in the software life cycle. As such, they are too general for most uses. Fortunately, both factors are relatively easy to divide into sub-factors with a simple procedure:

1. Identify most common resources used in the management layer.
2. Identify most common stages in the software life cycle.
3. Define suitable sub-factors for each (resource, stage) pair.

The resources and stages together create a sub-factor matrix where each cell may have several sub-factors. The matrix size and structure depend on the chosen resources and factors. There is usually no need to name the sub-factors in the matrix. The pair (resource, stage) identifies all sub-factors in that cell. A short characterisation identifies a single sub-factor from the others in the cell.

For example, typical resources available are human, hardware, energy, and facility resources. Typical software life cycle stages are requirements analysis, design, implementation, testing, installation, reuse, refactoring, and disposal. With this division, we would get a 4*8 sub-factor matrix, that is, at least 32 - sub-factors for each factor. For instance, a pair (human resources, implementation) could simply include a sub-factor of *feasibility* called *implementation efficiency* or just a vector (human resources, implementation, efficiency) (Table 6.3).

As can be seen, sub-factor matrixes grow large. Fortunately, cells often have closely related sub-factors, so we do not need to define all of them. For example,

Table 6.3 Feasibility sub-factor matrix with sub-factor examples

Resources/ stages	Human	Hardware	Energy	Facility
Requirements	Requirements analysis complexity			
Design				Facility utilization
Implementation				
Testing		Test hardware usability		
Reuse				
Refactoring			Refactoring energy efficiency	
Disposal				

most stages of facility efficiency can be covered with a sub-factor *facility utilisation*.

The sub-factor matrix helps us to define metrics for *feasibility* and *minimality*. Each sub-factor has a set of criteria for evaluating software status for the sub-factor. Some criteria are suitable for several factors and sub-factors.

A list of possible sub-factor criteria and metrics is worth a book of their own. We only include the most popular and widely used metric for green software and green software engineering: carbon footprint.

Carbon footprint (CF) measures how much carbon dioxide a product emits during its life cycle. It is the most common and arguably most important metric in sustainable development. With proper measurements, it is not only possible to have relevant measurements about software carbon emissions but also to compare the results to the carbon footprints of other physical and non-physical resources. We have used the CF metric in our analysis of software life cycle emissions [24].

The management layer is the most researched area in our model. Software engineers and managers have already optimised used processes to excellent production efficiency, waste reduction, and process and software reusability. For instance, the very idea of lean software engineering [22] is to reduce waste. Its methods clearly follow green software engineering principles. Green software engineering can add maximum resource efficiency and improved maintainability and configurability to traditional software engineering.

6.5.2   Execution Layer

The second layer in our model is the execution layer. It is the first layer where software plays an active role. The layer has factors that are related to software execution within a suitable platform. All factors, criteria, and metrics in this layer are hardware and implementation independent. They can be used in any software regardless of the problem domain.

In our earlier work, we called *efficiency* a factor that defines how software behaves when it comes to saving resources and avoiding waste. Unfortunately, *efficiency* is far too general for our layered model. In fact, it would be a super-factor that covers both *resource effectiveness* and *triftness*.

For the execution layer, we need a factor that is an execution-related sub-factor of resource effectiveness and a similar sub-factor to *triftness*. For these, we define *execution efficiency* and *utility*.

Execution efficiency defines how efficient it is to resource-wise execute software. For example, possible software execution resources include CPU, main memory, sensors, and internal peripherals. We can evaluate how effectively software uses these resources.

Utility defines how well software execution minimises its waste. For example, each CPU operation, access to main memory, peripherals, and sensors requires energy and hence has a CF. With this, we can evaluate the CF value of executing

software. We can improve *utility* by minimising the CF of each software accessed resource (e.g. by changing our source of energy) or by minimising the number of accessed resources during software execution.

Again, *execution efficiency* and *utility* are not the same sub-factor. For example, we could have software that uses a main memory database to minimise required execution time. Due to high main memory usage, it minimises execution time but requires more energy resources for the main memory. Its *execution efficiency* is good, but its level of *utility* depends on how the required energy is produced. *Utility* is much higher when renewable energy sources are used than when coal is used to generate energy.

Execution efficiency and *utility* are large factors that are of little use as is. We can use a similar technique to the one in the management layer to divide the factors into suitable sub-factors. We have a set of resources and a set of software execution stages. We can create pairs (resource, stage) and add suitable sub-factors for each pair.

For example, typical resources in software execution are CPU, main memory, sensors, and internal peripherals. Typical stages in software execution are software start, software execution, software wait, software finish, and software restart. With this division, we will get a 4*5 matrix, that is, 20 sub-factors for each factor.

At the execution layer, the main goal is to save resources. The software engineers should minimise resource requirements without losing functionality and without indirectly generating more waste elsewhere. For example, wasted CPU time is a criterion of how much of software execution is in operations that do not bring visible value for the end users. A typical example of such a code is to ensure input validity. Such a code is a waste since it does not bring visible value to the end user. Yet missing the code would create more waste because software would malfunction more easily. What is needed is to minimise CPU cycles from input validation without losing its functionality.

The execution layer factors, criteria, and metrics are practical and relatively easy to calculate. However, interpreting the numbers and especially comparing them to more common metrics such as the carbon footprint metric are not that clear. For example, we can calculate how many CPU cycles our software consumed and how much energy is required to create one CPU cycle. This allows us to calculate an estimation of how much carbon dioxide emissions our software created in the CPU cycles. Great. But since CPUs are part of hardware, they would have consumed energy regardless of our software. Do the carbon emissions count to the software CF or to the system CF? This is a matter of definition and depends on who is doing the calculations.

6.5.3   Platform Layer

The platform layer is the first layer where software architectures and architectural platforms play a role. It helps to define factors, criteria, and metrics for system services.

At the platform layer, *resource effectiveness* implies how resource efficient it is to execute system software. We could use the sub-factor *execution efficiency* here, but for the sake of distinction we define a sub-factor *service efficiency*. It defines how efficient it is to execute system software in a specific platform. Similarly, we could use *minimality* for waste analysis, but due to the separation from regular software, we want to use a sub-factor *service utility* for system waste analysis.

Service efficiency evaluates how efficiently software uses resources that are related to system software and the execution platform. While in *execution efficiency*, we were interested in concrete hardware components such as CPU and main memory. In *service efficiency*, we are more interested in external resources such as network, cloud, and databases.

Service utility evaluates how well software minimises system-level resource waste when using the resources. Again, we are interested in external resources.

A similar approach to the one in the previous layers works at this layer. We can divide *service efficiency* and *service utility* into sub-factors by creating a suitable matrix. However, at this layer the division to execution stages is not interesting. We have software that requires platform-specific resources. We need to know what resources are available and at what platform. The stage of the client software is always the execution stage so we do not need to take the stage to the analysis.

A better approach is to consider what kind of a platform we have and what resources such a platform would require from outside and how. We can even consider end users to be one such resource especially when we have a server platform. We can then create a similar matrix than in the previous layers. Only this time the cells are (platform, resource) pairs and the sub-factors of type (platform, resource, factor) triples.

With this approach, not all sub-factors at the platform layer are relevant to every software. For example, an embedded software system in a refrigerator does not have the same sub-factors as a weather prediction software in a supercomputer. Yet, resource-wise, there are similarities on both systems, such as network access.

We consider the following platforms at the platform layer: embedded platforms, mobile platforms, parallel platforms, desktop platforms, and server platforms. Other platforms can be included easily. Embedded platforms include platforms to support embedded software. Parallel platforms support software that is distributed to a very large number of nodes. Desktop platforms support normal desktop or laptop software. Server platforms support the Web and other server software.

Again, we can list resources that are relevant to the platform layer. We get at least the following: energy, network, cloud, space, end user time, and external peripheral time.

Energy resources are present at every layer. At the platform layer, we are interested in the total energy requirements of the service execution. How much energy is needed in the platform and how much indirectly outside when the service request is executed? This information is relevant to all types of platforms.

Network resources are especially important in almost all platforms. In the near future, even kitchen appliances and bathroom cabinets will have network access. These resources need to be used efficiently and with minimum waste.

Cloud resources are related to the network resources but also include other hidden resource requirements. Cloud server software itself is green since it supports *triftness* by minimising wasted computing resources (such as idle time) and *social sustainability* by offering access to its services. However, from a service software point of view, it matters how cloud resources are used at the client side. This information is relevant to all platforms that require net access and at least some external storage.

Space resources are especially important in mobile and embedded platforms where storage space is expensive. We can support *resource effectiveness* and *triftness* by minimising space requirements and sometimes also minimising space energy usage requirements.

End user time is one of the most important resources at the platform and higher layers. Here, however, the most important aspect is *triftness* because maximising human resource efficiency can lead to unwanted social side effects.

External peripheral time defines how much software uses peripherals that are outside its immediate vicinity. For instance, printers, monitors, and external permanent data storage are external peripherals, while main memory and internal permanent data storage (usually disk storage) are not. External peripherals are important since they also generate waste. We need to find a balance between external peripheral usage and wear.

With this division, we get a 5*6 matrix where each cell is a pair (platform, resource). Each cell includes one or more sub-factor of *service efficiency* and *service utility*.

The platform layer is also the first layer to support. Hence, it is a common layer between green ICT and SE for SD. The next layers are for only SE for SD.

6.5.4   Application Layer

Any software is always part of a software system. One aspect of the application layer is to evaluate how well software helps the software system and software system stakeholders to reach their objective. Green software supports its software system to reduce waste and use available resources at maximum efficiency.

The difference between the application layer and execution and platform layers is in the direction of support. In the previous layers, we analysed resource requirements and waste reduction for software *inputs*. At application layer and higher layers, we analyse software *outputs*.

For example, a typical output of a desktop software is a report. If the report is viewed on a display, its resource requirements are of peripheral resources. If it is printed, paper and ink usage generates both resource requirements and waste. If it is sent to several people, the resource usage and generated waste should be included for all receivers.

We call the effects of software to its system-level outputs *reflectivity*. It states how much and how software positively affects its system and stakeholders. *Reflectivity* could be divided into resource reflectivity and waste reduction reflectivity. However, since we are talking about indirect effects, the difference between resource efficiency and waste reduction is more on the actions of the software system and software stakeholders than software itself.

Reflectivity considers effects of all software execution stakeholders. As such, *reflectivity* is undoubtedly the most important factor in the green software factor model. The difference in resource usage and waste generation is a multiplication of all the stakeholders. For instance, consider *reflectivity* of a word processor. However, we are often more interested in the direct effects of software. We define *fit for purpose* for this purpose.

Fit for purpose defines how well software fulfils its objective. It is not a sub-factor of *reflectivity* since we can have software that has good *fit for purpose* but bad *reflectivity*. The difference comes from the level of indirection. At the first-order effects, *fit for purpose* and *reflectivity* both define how well software supports its direct end users and system. At the second-order effects, where we consider effects to other stakeholders, *fit for purpose* and *reflectivity* are equal only when the effects of software are positive.

Our earlier example of spamming software is a good example of a case where *fit for purpose* and *reflectivity* are not equal. Spamming software has good *fit for purpose* since it does its job very well. However, it has bad *reflectivity* since it steals resources from spam receivers and generates waste.

On the other hand, *fit for purpose* and *reflectivity* of good eShop software are equal. *Fit for purpose* eShop system is to sell products efficiently via the Internet. Good eShop software supports this by offering a fast interface and a positive user experience. *Reflectivity* of eShop software is to offer a positive and efficient user experience. It should maximise its customers' eShop visit efficiency and minimise time spent on secondary functions (i.e. their waste time).

Fit for purpose is a problematic factor due to its duality. We consider *reflectivity* to be more important than *fit for purpose*, but quite often direct software stakeholders are more interested in *fit for purpose* than *reflectivity*. Due to this, we wanted to include *fit for purpose* at this layer although it is not listed in Table 6.2.

Since *fit for purpose* does not include *social sustainability* and *reflectivity* is at a very general level on it, we define a factor *support for society* for *social sustainability* support.

Support for society defines how software supports socio-economical equality. Like *reflectivity*, it has first- and second-order effects. The first-order effects of *support for society* define how software supports social equality of its end users.

The second-order effects of *support for society* define how software end users can use software to support social equality.

For example, the previous eShop software has high first-order *support for society* when it allows everyone to use the eShop resources as long as they have network access. It could have second-order *support for society* if it was used to sell products that support social equality.

Exact criteria and metrics for the application layer are beyond the space of this chapter. We can use a matrix approach similar to earlier layers to first define suitable sub-factors and then define criteria for them.

For example, we can use a following procedure:

1. Identify software stakeholders.
2. Identify the first- and second-order effects to stakeholders.
3. Define criteria (or sub-factors if you are brave) for each (factor, stakeholder, effect) triple.

At the higher and more abstract levels, it often makes sense to evaluate criteria instead of sub-factors. At the application level, the deeper we get into factor details, the smaller is the problem domain where the factor has value. The first- and second-order software effects can be anything. A detailed list of sub-factors would grow rapidly and hide the big factor picture from small details.

The software layer is the most important layer in sustainable software. It is the layer where we can get the best idea of how software in general supports sustainable development. It is still a system-independent layer, and the ideas and factors at this layer do fit all software.

6.5.5 System Layer

At the previous layer, software was considered a stand-alone product with the first- and second-order effects to its stakeholders. The closest stakeholder of software is its software system. At the system layer, we consider how software in cooperation with its software system supports *resource effectiveness*, *triftness,* and *social sustainability*.

It is important to notice the difference between green software and a *green software system*. We can have green software in a non-green software system and vice versa. For example, a coal power plant software system is not a green software system since it creates much waste. However, an optimising software for minimising emissions of the plant is green software since it helps the system to reduce waste. Such software supports *triftness* when it minimises emissions and *fit for purpose* due to support to its system to produce maximum energy with minimum waste.

At the system layer, we want factors that support both green software and green software systems. Thus, while the previous coal power plant software was green software at the application layer, it is not green at the system layer.

We call the system-level support for resource efficiency *collaboration*. It defines how a software system helps its stakeholders to manage resources efficiently. A software system with high *collaboration* supports and cooperates with its stakeholders on resource efficiency. The support can be a first-order or second-order support.

At the first-order support, the software system itself helps in resource management. For example, a teleconferencing software system has high *collaboration* since it can save travelling-related resources of its end users.

At the second-order support, the software system helps its end users to support resource efficiency. For example, a research system to help in crop research will have a second-order support to farmers who use the results of the research.

We call the system-level support for waste reduction *reduction*. It defines how a software system helps its stakeholders to reduce waste. A software system with high *reduction* helps its stakeholders to directly or indirectly reduce waste. Again, the support is first or second order.

For example, a street light system that reduces unnecessary light to sky has high *reduction* and first-order support to waste reduction. It directly helps reducing serious waste: light pollution. This type of waste affects well-being, wildlife, and plants. The street light system also has second-level support to waste reduction if it helps people to sleep better and that way have a more active life.

We call the system-level support for social equality *production sustainability*. It defines how a software system helps its stakeholders to support social equality. A software system with high *production sustainability* supports its stakeholders in sustainable development.

For example, an MOOC (massive open online course) software system has high second-order *production sustainability*. With the system, MOOC teachers can support equality in learning which is an important area in social equality. It can also have high first-order *production sustainability* when it supports social equality among teachers such as a chance to teach on the MOOC without being physically present. (This is also a matter of *reduction*.)

The system layer is a useful layer due to its practical nature. At the layer, we are not that much interested in the role of software but consider software to be an integral part of the software system. This approach makes defining suitable criteria and metrics an easier task. Again, a complete list of good criteria and metrics is beyond the space and scope of this chapter. We only give an example of how it can be done.

Since a software system is a concrete entity, it is straightforward—while not always simple—to calculate its resource requirements. We need to calculate all elements of the system and see how much material, energy, and human resources their assembly requires and how much the elements have affected social equality. This is the initial state. Any waste at this time is not considered to be part of the software system.

When the software system is at the end of its life cycle, its usage has required resources, generated waste, and affected social equality. These resources can be calculated. This is the final state.

The difference between the initial state and the final state is the net summary of the software system. We probably want to estimate the final state since we seldom want to wait until the disposal time of the software system to evaluate its factors.

The next step is to evaluate the first-order effects that the system has to resource usage, waste generation, and social equality. These effects can be measured from stakeholders that are directly in contact with the software system. The results are added to the earlier measurements.

The final step is to evaluate the second-order effects to resource usage, waste generation, and social equality. These are the effects that the first-level stakeholders have on their stakeholders. Again, the results are added to the earlier measurements.

The net result gives values to a software system's *collaboration*, *reduction*, and *production sustainability*. If the results can be calculated, they are comparable with the results of any system.

The actual calculations for the second-order effects can be very difficult and sometimes even impossible. Yet some estimations are possible.

6.5.6 Problem Domain Layer

The final layer, the problem domain layer, is the conclusion of the green factor model and layered factors. Here all factors, criteria, and metrics are relative to the chosen problem domain. Due to this, we define only one factor at this level: *beauty*.

Beauty defines how software supports sustainable development. It is the ultimate green software factor. *Beauty* includes *resource effectiveness*, *triftness,* and *social sustainability* at this layer.

It is up to a problem domain specialist to divide *beauty* into sub-factors. *Beauty* sub-factors are usually not compatible between problem domains. For example, does women's education supporting software have more *beauty* than software that increases awareness of carbon dioxide emissions?

If a problem domain is clearly defined with reasonable requirements, *beauty* is measurable. We can use a simple algorithm:

1. Select a reference software in a software system.
2. Calculate whatever metric you want with reference software.
3. Calculate the same metric with evaluated software.
4. Compare results.

For example, let us have a problem domain of car braking. We can measure how much we can reuse braking energy. The reference software controls the brakes without support for reuse. The new software supports reuse. If the new software saves 15 % braking energy in the software system and the old software saves none, then we can evaluate the new software to have 15 % more *beauty* than the old software. However, the result is specific to a problem domain and in the purest form requires that both software run in the same hardware.

It is important that the hardware does not change in the measurement. If we compare two different braking systems, it is difficult to say how much of the difference is due to better software and how much is related to hardware.

Even without good absolute factors, criteria, and metrics for the problem domain layer, the layer itself and the ideas of how to measure green software factors within it are important. It matters how software supports *beauty*. We have tools and techniques to create, execute, and measure software that can be truly beautiful.

6.6   Conclusions

In this chapter, we introduced a layered model for green software and green software engineering quality. The model is based on the idea of software-supported sustainability, including software engineering.

In order to support sustainability, software has to support both sustainable environment and socio-economic equality. These sustainability characteristics can be generalised into three goals: resource efficiency, waste reduction, and support for human equality. Green software helps to reach all three goals both directly via its actions and indirectly via support to its stakeholders. Moreover, green software engineering helps in software development and management.

In order to fulfil the goals above, we need a green software quality model. Our model is based on the GREENSOFT model by Naumann et al. With the GREENSOFT model and our requirements for sustainability, we get a layered model for green quality factors. Each layer defines green software quality from a specific point of view.

Our layered model consists of the following layers: management layer for green software engineering, execution layer for green software execution, platform layer for green software services, application layer for green software applications, system layer for green software systems, and problem domain layer for sustainability-related software system problem domains. Each layer defines green software quality from a slightly different point of view.

When a sustainability specialist wants to define how green a software product is, he or she can choose the layer that best suits his or her needs. He or she can use the management layer to see how well software development supports sustainability, the execution layer to see how well software uses its resources, platform layer to see how well system software services support sustainability, application layer to see how well software supports its software system in sustainability, system layer to see how well the software system supports sustainability, and problem domain layer to see how well the software system supports its problem domain when compared with similar software systems. Each layer is important depending on an observer's point of view, but usually all layers are not equally important to the observer.

On each layer, we define a set of green software factors and sub-factors to support sustainability. However, the exact definitions of the factors are less important than the idea behind them. Some factors may live the test of time, and some

may not. Yet we are certain that the idea of layered software support to sustainability remains.

References

1. ISO (2011) Systems and software engineering – systems and software quality requirements and evaluation (SQuaRE) – system and software quality models. ISO, Geneva
2. Amato I (2003) Green cement: concrete solutions. Nature 494:300–301
3. Banerjee A, Solomon BD (2003) Eco-labeling for energy efficiency and sustainability: a meta-evaluation of us programs. Energy Policy 31(2):109–123
4. Briand LC, Wüst J, Daly JW, Porter DV (2000) Exploring the relationships between design measures and software quality in object oriented systems. J Syst Software 51(3):245–273. doi:10.1016/S0164-1212(99)00102- 8
5. Brundtland G et al (1987) Our common future: report of the 1987 world commission on environment and development. United Nations, Oslo, pp 1–59
6. Chidamber S, Kemerer C (1994) A metrics suite for object oriented design. IEEE Trans Software Eng 20(6):476–493. doi:10.1109/32.295895, URL http://dx.doi.org/10.1109/32.295895
7. Fenton NE, Pfleeger SL (1997) Software metrics: a rigorous and practical approach. PWS, Boston, MA
8. Galin D (2004) Software quality assurance: from theory to implementation. Pearson/Addison Wesley, Harlow
9. Gartner (2007) Green IT: the new industry shockwave. Presentation at symposium/ITXPO conference
10. Geller T (2011) Supercomputing's exaflop target. Commun ACM 54(8):16–18
11. Hopwood B, Mellor M, O'Brien G (2005) Sustainable development: mapping different approaches. Sustain Dev 13(1):38–52
12. IUCN, WWF (1980) World conservation strategy. World Conservation Union, United Nations Environment Programme, Word Wide Fund for Nature
13. Kern E, Dick M, Naumann S, Guldner A, Johann T (2013) Green software and green software engineering – definitions, measurements, and quality aspects. In: Proceedings of the first international conference on information and communication technologies for sustainability (ICT4S 2013), pp 87–94
14. Lamb J (2009) The greening of IT. How companies can make a difference for the environment. IBM Press, Indianapolis, IN
15. McCabe T (1976) A complexity measure. IEEE Trans Software Eng SE-2(4):308–320. doi:10.1109/TSE.1976.233837, URL http://dx.doi.org/10.1109/TSE.1976.233837
16. McCall JA, Richards PK, Walters GF (1977) Factors in software quality. Technical report for the Rome Air Development Center (ISIS), General Electric
17. Murugesan S (2008) Harnessing green it: principles and practices. IT Prof 10(1):24–33
18. Naumann S, Dick M, Kern E, Johann T (2011) The GREENSOFT model: a reference model for green and sustainable software and its engineering. Sustain Comput 1(4):294–304
19. Naumann S, Kern E, Dick M (2013) Classifying green software engineering-the GREENSOFT model. In: 2nd workshop EASED@ BUIS 2013, 13
20. Naur P, Randell B (1969) Software engineering: Report of a conference sponsored by the NATO science committee, Garmisch, Germany, 7–11 Oct 1968, Brussels, Scientific Affairs Division, NATO
21. Pezzè M, Young M (2008) Software testing and analysis: process, principles and techniques. Wiley, Chichester

22. Poppendieck M (2007) Lean software development. In: Proceedings ICSE COMPANION '07 Companion to the proceedings of the 29th international conference on software engineering, pp 165–166
23. Shein E (2013) Keeping computers cool from the inside. Commun ACM 56(7):13–16
24. Taina J (2010) How green is your software? In: Proceedings of the first international conference on software business (ICSOB 2010), pp 151–162
25. Taina J (2011) Good, bad, and beautiful software – in search of green software quality factors. Cepis Upgrade 12(4):22–27

Part IV
Software Development Process

Chapter 7
From Requirements Engineering to Green Requirements Engineering

Birgit Penzenstadler

7.1 Introduction

Requirements engineering (RE) is the early phase of software engineering where we determine the exact scope of the system and iteratively elaborate the stakeholders' needs and concerns. Step by step, these are refined into more specific requirements and constraints for the system under development.

Within the overall green software engineering process, RE ties in between business analysis (see chapter 'Green Software Economics' in this book), testing (see chapter 'Green Software Testing') and architecture (see chapter 'Green Software Construction'). Good software engineering practice mandates to perform these phases in an iterative fashion with feedback loops [31].

For describing green software requirements, the basic building blocks are to define *requirements engineering* and to define what *green* means.

7.1.1 Defining Requirements Engineering

Zave [45] provides one of the clearest definitions of RE:

> Requirements engineering is the branch of software engineering concerned with the realworld goals for, functions of, and constraints on software systems. It is also concerned with the relationship of these factors to precise specifications of software behavior, and to their evolution over time and across software families. [45], p. 315

Nuseibeh and Easterbrook explain this definition further and expand it on several tasks that become important when adapting RE for green software:

B. Penzenstadler (✉)
California State University, Long Beach, USA
e-mail: bpenzens@uci.edu

Requirements Engineering is concerned with interpreting and understanding stakeholder terminology, concepts, viewpoints and goals. Hence, RE must concern itself with an understanding of beliefs of stakeholders (epistemology), the question of what is observable in the world (phenomenology), and the question of what can be agreed on as objectively true (ontology). Such issues become important whenever one wishes to talk about validating requirements, especially where stakeholders may have divergent goals and incompatible belief systems. They also become important when selecting a modelling technique, because the choice of technique affects the set of phenomena that can be modelled, and may even restrict what a requirements engineer is capable of observing. [32], p. 35

7.1.2   Defining Green and Sustainability

Over the last decades, sustainability research has emerged as an interdisciplinary area; knowledge about how to achieve sustainable development has grown, while political action towards the goal is still in its infancy [13].

- The Oxford English Dictionary defines sustainable as 'capable of being upheld; maintainable' and to sustain as 'to keep a person, community etc. from failing or giving way; to keep in being, to maintain at the proper level; to support life in; to support life, nature etc. with needs'. The etymology of the terms originates in the French verb *soutenir*, 'to hold up or support' [4].
- Sustainability can be discussed with reference to a concrete system—such as an ecological system, a human network or even a specific software system. Software engineering for sustainability has developed as a current focus of research due to sustainability being advocated as a major objective for behaviour change on a global scale.
- The attribute *green* is widely used to denote an emphasis on environmental sustainability. At the same time, the above explanation makes it clear that overall sustainability in our daily lives can only occur when the various aspects are in balance. This has to be reflected in the software systems we create.

7.1.3   Defining Green Requirements Engineering

The term *green* or *sustainable software* can be interpreted in two ways: (1) the software *code* being sustainable, agnostic of purpose, or (2) the software *purpose* being to support sustainability goals, that is, improving the sustainability of humankind on our planet. Ideally, both interpretations coincide in a software system that contributes to more sustainable living. Therefore, in our context, sustainable software is energy efficient, minimises the environmental impact of the processes it supports and has a positive impact on social and/or economic sustainability. These impacts can occur as direct (energy), indirect (mitigated by service) or systemic and, potentially, rebound effect [24].

Requirements engineering for sustainability denotes the concept of using requirements engineering and sustainable development techniques to improve the environmental, social and economic sustainability of software systems and their direct and indirect effects on the surrounding business and operational context. In order to develop such systems, we need awareness (by education), guidance (e.g. as in this book) and creativity (to find better solutions).

Green requirements engineering consequently denotes that same concept with a specific focus on the direct and indirect *environmental* impacts of systems. However, as sustainability is an encompassing concept and one aspect of it cannot be strengthened without considering the other dimensions, we will still discuss it in the broader scope, referring to all five dimensions.

7.1.4 Five Dimensions of Sustainability

As we are convinced that a focus on environmental sustainability only makes sense when in balance with the other dimensions of sustainability, we define these further dimensions and describe how they have been represented in requirements engineering up to now.

There are different dimensions to sustainability [37]: *Individual* sustainability refers to maintaining human capital (e.g. health, education, skills, knowledge, leadership and access to services). *Social* sustainability aims at preserving the societal communities in their solidarity and services. *Economic* sustainability aims at maintaining capital and added value. *Environmental* sustainability refers to improving human welfare by protecting the natural resources: water, land, air, minerals and ecosystem services. *Technical* sustainability refers to longevity of systems and infrastructure and their adequate evolution with changing surrounding conditions.

For the general characteristics of sustainability requirements for the respective dimensions, we have found the following subtypes that are used in other requirements categorisations [41]:

- *Environmental*: Requirements with regard to resource flow, including waste management, can be elicited and analysed by life cycle analysis (LCA). Furthermore, impact effects can be analysed by environmental impact assessment (EIA). Other aspects are efficiency and time constraints. The problem is that usually only the first-order impacts by a system are considered, whereas the second- and third-order impacts are not even in the conscience of the developers because they will not be held responsible for them. The only way to change this is to change our mindset and actively include a notion of responsibility for the wider impacts our actions have on the surrounding environment.
- *Human*: Parts of *human sustainability* are covered by privacy, safety, security, HCI and usability. In addition, there is a strong focus on personal health and well-being, which still needs to be made explicit in requirements. An example

for this might be that an application suggests to take a break after a specific amount of working time.

- *Social*: A share of *social sustainability* can be treated via computer-supported cooperative work (CSCW) requirements, which reflect the interaction within user groups; via ICT for development (ICT4D) requirements; and via political, organisational or constitutional requirements, as in laws, policies, etc. What is still missing are, for example, explicit requirements for strengthening community building.
- *Economic*: *Economic sustainability* is taken care of in terms of budget constraints and costs as well as market requirements and long-term business objectives that get translated or broken down into requirements for the system under consideration. The economic concern lies at the core of most industrial undertakings.
- Technical: The technical sustainability requirements include non-obsolescence requirements as well as the traditional quality characteristics of maintainability, supportability, reliability and portability, which all lead to the longevity of a system. Furthermore, efficiency, especially energy efficiency, and (hardware) sufficiency [13] should be part of the technical sustainability requirements.

This list shows that four of the five dimensions are already supported to a considerable extent by traditional software quality characteristics and requirements and can be dealt with. The least support exists for the environmental dimension. Consequently, we especially need to consider and better support the second- and third-order impacts in the environmental dimension of software systems. One way to address this would be to describe sustainability requirements, as combinations of the five dimensions *human*, *social*, *environmental*, *economic* and *technical* times the *orders of effect* as proposed in [41].

Instead of proposing a new framework that might interfere with established practices and be negated by the average user as sustainability is only one of many objectives for a system, in this chapter we aim at presenting how requirements engineers can take sustainability considerations into account within their established practice.

7.1.5 Why Not Simply Add a Category Green Software Requirements?

A question that might arise is why we do not see an actual category green software requirements in there. The reason is that trying to establish a new category of requirements that have to be treated separately would lead to increased effort for developers, which would lead to resistance. Resistance to change does not mean that software developers would not accept to consider environmental sustainability as an objective but only that humans as well as organisations of any form are resistant to change for a variety of reasons [3].

Instead, we are following an integrative approach that shows that requirements engineering can easily accommodate the new objective of improving the environmental sustainability of software systems using its current techniques and incorporating simply a few more instantiations of known requirements types.

7.1.6 Outline

The rest of this chapter is outlined as follows: We describe how to elaborate green requirements and show where different types of sustainability requirements occur in the overall process of requirements engineering. To better illustrate these steps, we then introduce an artefact model for requirements engineering and provide example excerpts for green requirements engineering content items. Furthermore, we discuss aspects like requirements conflicts, costs, legal constraints and risks. We conclude with how much difference the consideration of sustainability actually makes in requirements engineering and point to practical consequences and related future research.

7.2 Elaborating Green Requirements

Starting from scratch for developing any type of software system, how would we elaborate *green*, *sustainable* requirements within a generic requirements engineering approach? There are a few questions to help guide the way, as summarised in Fig. 7.1.

Q1. *Does the system under consideration have an explicit purpose towards environmental sustainability?* If yes, this can be analysed in depth. If no, it can be considered whether such an aspect is desirable and feasible to add. If, again, that is not the case, then the analysis details the potentials for greening *of* that IT system (further explored in question 2) instead of greening *through* IT, but depending on the kind of system this might still lead to considerable improvements of the environmental impact of the system [33]. In case the system is widely used, that is worth the effort.

Q2. *Does the system under consideration have an impact on the environment?* Any system has an impact on the environment, as any system is applied in a real-

Guiding Questions for Green RE:
1. Does the system have an explicit sustainability purpose?
2. Which impact does the system have on the environment?
3. Is there a stakeholder for environmental sustainability?
4. What are the sustainability goals and constraints for the system?
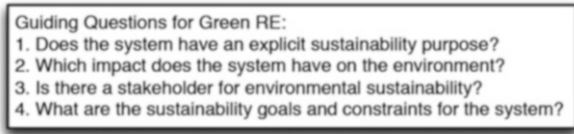
Fig. 7.1 Guiding questions for green requirements engineering

world context of some kind, which is situated within our natural environment. Consequently, it has to be analysed as to what are the direct (first order), indirect (second order) and systemic as well as potential rebound effects (third order). This potentially includes a very large scope, especially for the third-order effects, but systemic thinking [27] facilitates such an analysis process and may lead to significant insights.

Q3. *Is there an explicit stakeholder for sustainability?* In case there is an explicit stakeholder who advocates for environmental sustainability, I already have a significant voice that issues objectives, constraints and considerations to support the quality in the system under consideration. In case there is no such advocate, it can be decided to establish such a role. Otherwise, the least representative for sustainability that should be established is a domain expert responsible for providing information on applying environmental standards, legislation and regulations.

Q4. *What are the sustainability goals and constraints for the system?* Independent of whether the system has an explicit purpose for supporting environmental sustainability or not, there certainly are a number of objectives that pertain to the different dimensions of sustainability that may be chosen to apply. For example, a social network might not have an explicit environmental purpose, but it certainly has objectives supporting social sustainability. Furthermore, any system will at least have some constraints with respect to the environment, as stated in question 2.

For the description of how to elaborate green requirements, we limit ourselves to a few concepts that are commonly agreed on as content items or information elements for gathering and refining requirements, all depicted in the overview in Fig. 7.2. These are *business processes*, *domain models*, *stakeholders*, *objectives*, *constraints*, *system vision* and *usage model*, as well as *quality requirements*, *process requirements*, *deployment requirements* and *system constraints*. There are a number of potential starting points for green requirements engineering with related elicitation and analysis activities as illustrated in Fig. 7.2:
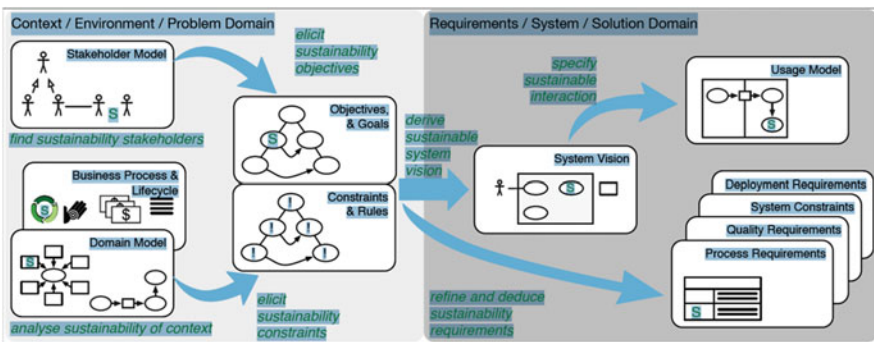
Fig. 7.2 Overview of content items and information flow for green requirements engineering

- In case there is a relation of the business process of the system under consideration to sustainability or environmental issues (see Q1), the *business process model* is the first piece of information that may explicitly include green concerns in the form of supporting business processes or services. If that is not the case, then there will still be elements in the *domain model* that can be related to sustainability concerns, due to the impacts caused by the system (see Q2). This is denoted by the activity *analyse sustainability of context*.
- If the business context and application domain lack adequate root elements for a sustainability analysis, the *stakeholder model* may be used as the starting point (see Q3), characterised by the activity *find sustainability stakeholders*. Whichever the system under consideration, the stakeholder model should include a sustainability advocate, at least as a representative for legal constraints.
- In either case—a system with an explicit sustainability concern as well as without such a mission—the *objectives and goals* should feature sustainability as one major quality objective (see Q4). This objective should be included in the general reference goal model of a company used as a basis for instantiation for a particular system and then refined according to the system specifics. Apart from *eliciting sustainability objectives* from the stakeholders, it is also necessary to *elicit sustainability constraints* from the domain model for the *constraints and rules*, which includes sustainability-related constraints for any kind of system, for example, environmental standards.

From these different starting points, the sustainability requirements and constraints are propagated throughout the content items in requirements engineering as illustrated in Fig. 7.2. This includes the activities *derive sustainable system vision*, *specify sustainable interaction* and *refine and deduce sustainability requirements*. The following sections walk through these stages and describe the development of the respective content items.

7.2.1 Analyse Sustainability of Context

Whenever we are faced with a system that has an explicit contribution to sustainability either by improving our ways to analyse the environment and reporting feedback or by enabling and incentivising sustainable behaviour in its users, we can analyse the contextual elements this is related to in the business processes and the domain model. Examples for such systems are the carbon footprint calculator,[1] the Story of Stuff Project[2] or car-sharing systems like Zipcar[3] or DriveNow.[4] If it is not a purpose for environmental sustainability, there might still be a purpose for social

---

[1] http://coolclimate.berkeley.edu/carboncalculator

[2] http://storyofstuff.com/

[3] http://www.zipcar.com/

[4] http://www.drive-now.com/

sustainability, for example, different types of local community tools or social networks.

Whether sustainability is a concern or not, either way the system will cause some kind of impact on the environment, which can span from the first-order impacts to the third-order impacts. The system environment and the wider context are usually analysed using a domain model. This can also serve as the basis for a life cycle analysis [19] of the system under consideration. For the example of the car-sharing system, the first-order impacts would be the resources that the system itself (the application and the back-end servers) consumes. The second-order impacts would be the resources the car-sharing system triggers in its application domain, that is, the cars that are being shared on the road and that do consume a considerable amount of resources but decrease the overall consumption of resources through cars. The third-order effects might be a decrease in the number of individually owned cars and eventually less cars, but this remains to be observed in the long run. For the example of a social network, the first-order impacts are again the resources that the system itself consumes (front-end and back-end components); the second-order impact is the resource consumption caused by interaction with the system, for example, meeting friends and attending events; and a desirable potential third-order effect could be a global society that feels better connected and acts as global citizens, while a potential negative third-order effect might be a decrease in the participation of people in real-life events or reduction of social interaction to an online derivative.

7.2.2   Find Sustainability Stakeholders

Stakeholders are the basis for requirements engineering. They pursue goals, include the users of the system under development and issue constraints. One major pitfall for requirements engineering is to have an incomplete list of stakeholders, consequently resulting in incomplete requirements or constraints. To ensure smooth development, it is crucial to involve the stakeholders early on; to elicit desires, information and feedback from them; and to satisfy their information and communication needs. Typically, successfully dealing with stakeholders involves identification, classification, analysis and communication management. In the context of green requirements engineering, the goal is to elicit stakeholders that advocate for sustainability and that are domain experts for life cycle analysis, environmental concerns, legislation for environmental regulations or environmental standards.

Definition A stakeholder is a person or organisation who influences a system's requirements or who is impacted by that system [12].

There are different possible approaches to identifying stakeholders for sustainability [38], and Table 7.1 provides a generic list of sustainability stakeholders that may be used for reference. Most likely the best way to make sure all have been

Table 7.1 A generic list of sustainability stakeholders

Dimension	Stakeholder	Description/rationale
Individual	User	The user is affected by the system in various ways. For example, users of online learning courses educate themselves through software
	Developer	The developer is heavily involved in creating the system. Aspects like sustainable pace and growth of the developer must be considered
	Employee represent.	The mental and physical safety of individuals needs to be maintained. Employee representatives watch rights of employees involved
	Legislation (indiv. rights}	Systems must respect the rights of their users. A legislation representative is a proxy for privacy and data protection laws
Social	Legislation (state authority)	The state has a strong interest in understanding a system's influence on the society. Contrary to the individual rights legislation representative, the state authority representative speaks from the perspective of the state as a whole
	Community represent.	In addition to the state authority, other communities such as the local government (e.g. the mayor) or non-government clubs might be affected by a software system. A complete analysis must take their views into account
	CRM	The customer relationship manager (CRM) is in charge of establishing long-term relationships with their customers and creating a positive image of the company
	CSR manager	Some companies created the dedicated position of the corporate social responsibility (CSR) manager, who develops a company-specific vision of social responsibility
Economic	CEO	The chief executive officer integrates sustainability goals into a company's vision
	Project manager	It is very important to have the project manager agree in what ways the project should support sustainable aspects as he decides on prioritisation with conflicting interests
	Finance responsible	As sustainable software engineering often also affects the budget, many financial decisions have to be made to implement a sustainable software engineering model in a company
Environm.	Legislation (state authority)	Environment protection laws are in place to ensure sustainability goals. These laws must be reflected in the model
	CSR manager	The CSR manager is often also responsible for environmental aspects
	Activists/ lobbyists	Nature conservation activists and lobbyists (e.g., WWF, Greenpeace, BUND)
Technical	Admin	The administrator of a software system has a strong motivation for long-running, low-maintenance systems, making his work easier
	Maintenance	The hardware maintenance is interested in a stable, long-term strategy for installation of hardware items
	Customer	Users are interested in certain longevity of the systems they are using. This refers to the user interface and the required software and hardware

identified is a mix or iteration of these approaches, for example, in the order they are presented in:

1. *Phases*: Analysing the aspects and development phases of software systems development to find the responsible roles. This approach is an easy way to set up early elicitation meetings with the most important, rather obvious, stakeholders (see Table 7.1).
2. *Reference list*: Instantiating generic reference lists of stakeholders (see Table 7.1) for the concrete project context. This second step takes standard roles into account that have been included in reference models and enhance the initial quick list of stakeholders. A reference list for sustainability stakeholders is provided in Table 7.1.
3. *Context*: Inspecting the business and operational context of the system under development and understanding which concrete roles are involved. This step makes sure that the specifics of the project under consideration are all met and special roles are considered. A simple reference model that is being used in software engineering to map out stakeholders is the so-called onion model [1] with its four concentric spheres: product, system, containing system and the wider environment, which has been instantiated for a green system purpose [26].
4. *Goals*: Iteratively analysing and refining a generic goal model and deducing the related roles. This approach is especially suitable for finding passive stakeholders that do not have an active interest in issuing own goals but whose constraints have to be adhered to, for example, legislative representatives.

We document the stakeholders in a *stakeholder model* that allows to list and describe all stakeholders involved in a project. Stakeholders comprehend individuals, groups or institutions having the responsibility for requirements and a major interest in the project.

The stakeholder model is the checklist for ensuring that goals have been elicited and constraints have been collected from all stakeholders. Furthermore, the stakeholders have to validate the elicited requirements.

7.2.3 Elicit Sustainability Objectives, Goals and Constraints

The next step is to elicit the sustainability objectives and goals from the stakeholders and to deduce any sustainability constraints from the business processes and/or the domain model.

An objective or a goal is a discretionary abstract characteristic, which the system shall fulfil with regard to its operational environment or the development process of the system shall fulfil. Objectives and goals are issued by stakeholders and can be differentiated into various types that help structuring the ways of processing them during requirements engineering.

Definition A *goal* is an objective the system under consideration should achieve. Goal formulations thus refer to intended properties to be ensured; they are optative statements as opposed to indicative ones and bounded by the subject matter [18].

To facilitate goal elicitation, we distinguish three subcategories that refer to different levels of abstraction in systems development: Business goals are all business-relevant (strategic) goals as well as goals with a direct impact on the system or project. Usage goals are a direct relation to the functional context and usage of the system (user perspective) for behaviour modelling. System goals are system-related goals that determine or constrain system characteristics.

In order to consider the sustainability perspective during goal modelling, we consult the generic reference model for sustainability (see excerpt in Fig. 7.3; see [35, 36] for an encompassing description) that represents the sustainability dimensions by sets of *values*. Values are approximated by *indicators*, supported by *regulations* and contributed to by *activities* [36].

One option for application would be instantiating the generic sustainability model for a specific system. This is feasible in a case where sustainability is the major purpose of the system under consideration. For most systems, sustainability will be one amongst a number of objectives; therefore, it is more suitable to develop one overall goal model for the system and to detail the submodel for the objective of sustainability by using the sustainability dimensions and the generic sustainability model as a reference. This means to analyse the generic sustainability model and to decide for each value within the dimensions whether it is applicable to the system under development and, if so, to select those related activities which can be operationalised as goals for the system.
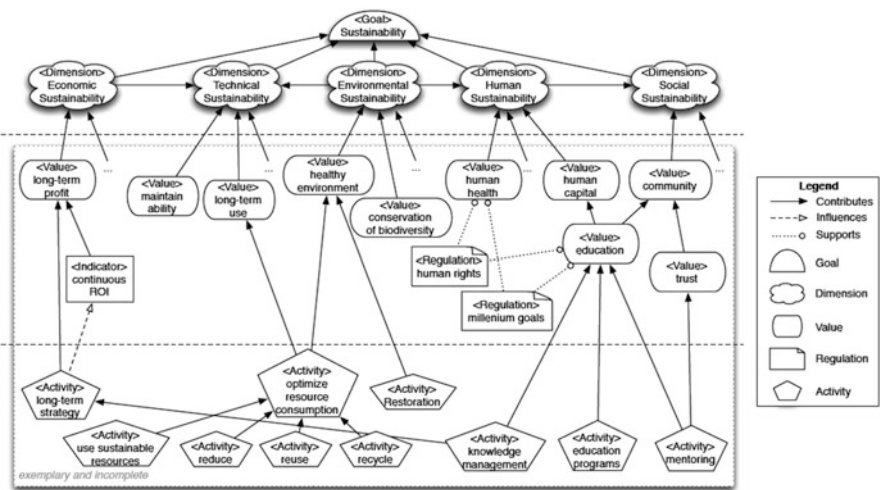
Fig. 7.3 Illustrative excerpt of the generic sustainability reference model [35, 36]

7.2.4   Derive Sustainable System Vision and Usage Model

The next steps are to derive a sustainable system vision and to specify sustainable system interaction in a usage model.

Definition A *system vision* is a common vision of the system under consideration agreed upon by all stakeholders that have an active interest in the system.

One frequently used method to create system visions that are easy to communicate is *rich pictures* [30]. A rich picture is a cartoon-like representation that identifies all the stakeholders, their concerns and some of the structural and conceptual elements in the surrounding work context. The choice of an easy-to-understand medium instead of a more formal and detailed one arises from the need that stakeholders of various domains and disciplines have to understand the vision.

The system vision is usually coupled to a milestone with the scope of an early draft of a common idea of the system. It can be used as an early basis for estimations and planning of the subsequent development process. Furthermore, it is a detection basis for moving targets.

In case the purpose of the system is closely linked to sustainability, this shall become very clear in the vision. In case it is a minor aspect, it may still be expressed as one of the concerns.

7.2.5   Refine and Deduce Sustainability Requirements

Finally, detailed sustainability requirements and constraints are refined and deduced in four categories: process requirements, deployment requirements, system constraints and quality requirements. Further concerns for the system or the project may be managed in a risk list.

Process requirements denote demands with regard to the conducted development, for example, using a green software engineering process. Deployment requirements specify demands with respect to the installation of the system and launching it into operation, for example, the migration of the data of the legacy system to the green data centre used for the system under development. System constraints detail restrictions on a system's technical components and architecture as well as related quality attributes, for example, hardware sufficiency, that is, that the system shall run on the old hardware without resource-intensive upgrades. Quality requirements describe the demands for individual quality attributes across a system's functionality, the satisfaction criteria of those requirements, the qualitative or quantitative metrics and how the metric will be evaluated.

By going through these steps, we have obtained detailed sustainability requirements that can be traced back to their respective origins in the business process, the

domain model, the stakeholder model or the goal model. From here on, the responsibility for ensuring that the requirements are designed into the system and eventually implemented moves on from the requirements engineer to the system architect and the designers. In the following section, we provide a number of examples for content items that illustrate the elaboration of the above-described steps by using a domain-independent artefact model for requirements engineering. As the artefact model itself is not specifically coupled to the approach of green requirements engineering, we chose to first present the approach and then illustrate it by means of excerpts from those artefacts.

7.3 Exemplary Application in Artefact-Oriented Requirements Engineering

In this section, we provide an illustrative application of green requirements engineering using an artefact-based approach to requirements engineering. Green requirements engineering may as well be applied with an activity-based approach though—the choice was taken for illustrative purposes, as the artefact-based approach provides an overview that is explicitly structured according to the work results. That way, the result excerpts can be seen in context with other requirements engineering work results.

Independent of the requirements engineering approach, an underlying system model provides the foundation for documenting the requirements to a system in a way that ensures that the requirements can be consolidated into a consistent system requirements specification. Therefore, we first introduce a basic system model and then the requirements artefact model.

7.3.1 Foundation: A Basic System Model

The basic system model underlying the presented approach considers a system to be composed of subsystems (components) interacting with other systems in its context.

According to Broy et al. [5, 6], a well-founded systems modelling theory provides firstly the appropriate modelling concepts such as the concept of a system and that of a user function, with (1) a concept for structuring the functionality by function hierarchies, (2) concepts to establish dependency relationships between these functions and (3) techniques to model the functions with their behaviour in isolation including time behaviour and to connect them into a comprehensive functional model for the system, and secondly a concept of composition and architecture to capture

1. The decomposition of the system into components that cooperate and interact to achieve the system functionalities
2. The interfaces of the components including not only the syntactic interfaces but also the behaviour interfaces
3. A notion of modular composition which allows us to define the interface behaviour of a composed system from the interface behaviours of its components

This is detailed in views of the system on three different abstraction levels: its surrounding business and operational context, the system itself as a black box and the system as a white box [5, 6].

Basing requirements engineering on a system model allows for early model orientation that facilitates the consolidation of objectives, requirements and constraints issued by the different stakeholders of a system. The documentation of the gathered information is performed via an artefact model.

7.3.2 AMDiRE: Artefact Model for Domain-independent Requirements Engineering

The various influences on processes and application domains make requirements engineering (RE) inherently complex and difficult to implement. When it comes to defining an RE reference model, we basically have two options: we can establish an activity-based RE approach where we define a blueprint of the relevant RE methods and description techniques or we can establish an artefact-based approach where we define a blueprint of the RE artefacts rather than a blueprint of the way of creating the artefacts. In the last 6 years, we have established several artefact-based RE approaches and empirically underpinned the advantages of applying those approaches in industry [11, 29, 34, 39]. Those approaches remain, however, complex as they encompass various modelling concepts and, in particular, incorporate their particularities of the different application domains, such as the one of business information systems. For this reason, we consolidated the different approaches and established the AMDiRE approach, that is, the artefact model for domain-independent requirements engineering. AMDiRE includes a detailed artefact model that captures the basic modelling concepts used to specify RE-relevant information, tool support and a tailoring guideline that guides the creation of the artefacts (see Fig. 7.4).

The above-explained system model and its context need to be documented in an adequate way in a system requirements specification. For this, we follow the artefact-oriented approach AMDiRE (Artefact Model for Domain-independent Requirements Engineering) [28]. For the purpose of this chapter, we use a reduced
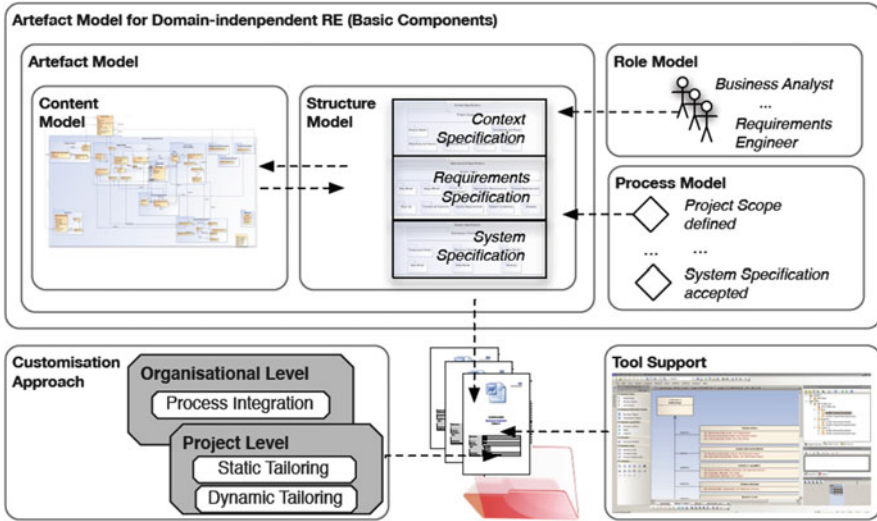
Fig. 7.4 Components of the AMDiRE approach

version of the model as depicted in Fig. 7.5. For the full AMDiRE model, please refer to [28].

7.3.3   Example Content Items for Illustration

For illustration purposes, this chapter uses an example system that most readers are likely to be somewhat familiar with: the online social networking service Facebook.[5]

How would it change the requirements of a software system if sustainability had been considered all along the way? For Facebook, the hypothesis is that the user interface would be a little simpler to require less energy and thereby run well on older devices (environmental sustainability), and the privacy policy and settings would have been available from the start as opposed to years later due to user complaints (individual and social sustainability).

In addition, we provide examples from systems that have an explicit sustainability objective in their system vision. One of them is a car-sharing system; another one is the Story of Stuff Citizen Muscle Boot Camp [21], a massive open online course system that shall educate users about how to take action for environmental causes.

---

[5] Disclaimer: The examples provided within this chapter are not the results of an official collaboration with Facebook, but rather the re-elicitation by the author.
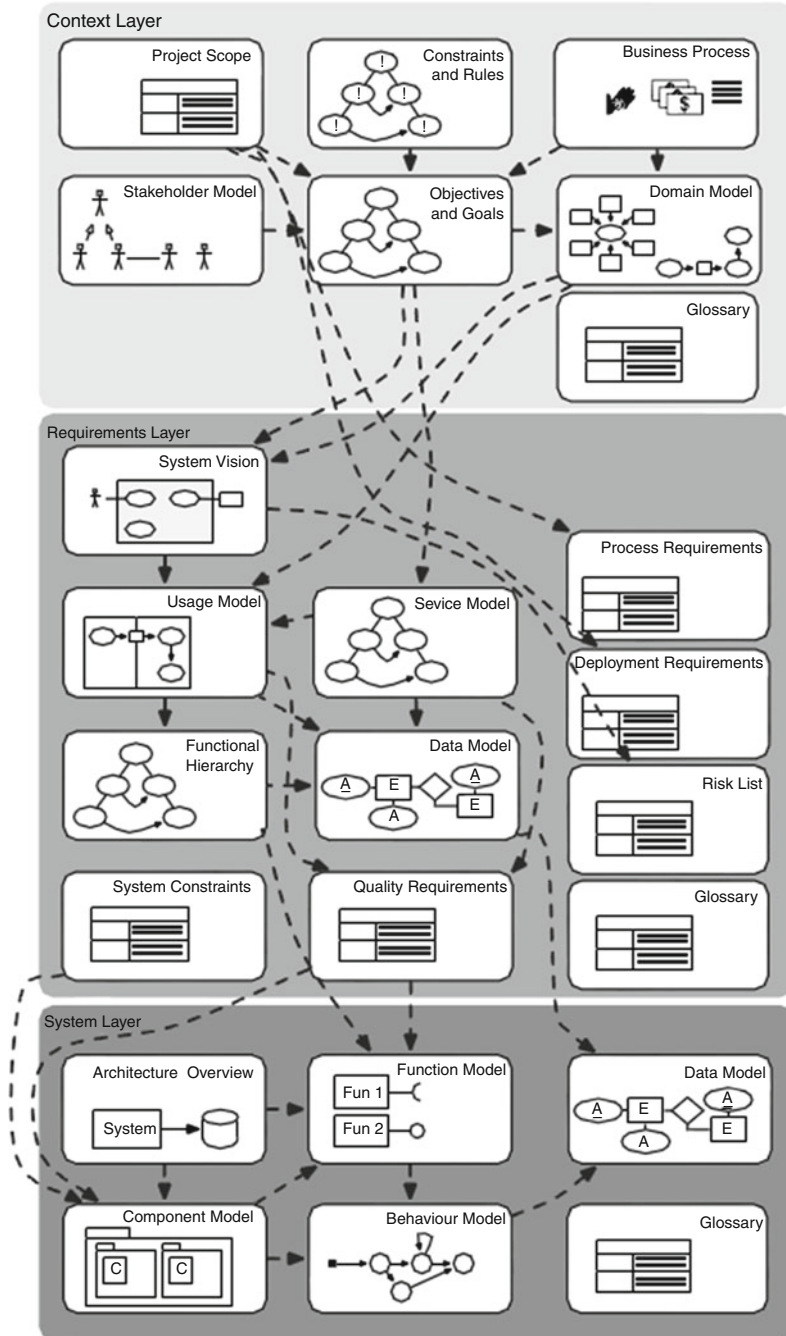
Fig. 7.5 A pictogram of the AMDiRE artefact model

7.3.4 Stakeholder Model

Stakeholders are documented in the *stakeholder model*. This content item describes all stakeholders involved in a project, comprehending individuals, groups or institutions. User groups are a specialisation of stakeholders interacting with the system. The means to document stakeholder models are UML actor hierarchies, informal hierarchical graphics or natural text.

The stakeholder model is the basis for rationale and a major source of requirements and future user groups and actors. The artefacts predominantly affected by the stakeholder model are the goal model, the domain model (business processes), the system vision and the usage model (use cases and scenarios).

The example box lists the stakeholders for Facebook, including the ones that advocate sustainability or serve as domain experts on its various aspects.

> **Example**
> *Stakeholder list for Facebook*
>
> - *Product owner: Facebook, Inc. executive board, shareholders, department heads, managers*
> - *Domain experts: Social media expert, communication expert, psychologist, business analyst, marketing expert, environmental specialist, sustainability consultant*
> - *Partners and suppliers: Plug-in providers, third software integrated in Facebook (e.g. games)*
> - *Developers: Requirements engineer, architects, implementer, tester, quality assurance, green product champion*
> - *Post-implementation support: Maintenance, managers, user forum moderators*
> - *Regulatory bodies: Legislative representatives for national and international laws w.r.t. privacy, data storage, trade and the environment, green certifying body*
> - *Users: Customers (who advertise on the platform), content users (the standard 'user' of an online network), green user, anti-green user, administrators*
> - *Competitors: Twitter, LinkedIn, Google Plus, YouTube, Foursquare, Reddit, Pinterest, Tumblr, Flickr, Instagram, Myspace, Meetup, Diaspora, etc.*

7.3.5 Objectives and Goals

The AMDiRE content item *objectives and goals* is denoted in, for example, KAOS [25] or i* [44]. Each goal, whether it is a *business goal*, a *usage goal* or a *system goal*, is issued by a *stakeholder*.

Goals satisfy a *statement of intent* [25], they build a hierarchy and they can influence each other in terms of conflicts, constraints or support. Each usage goal is related to a business goal and each system goal to a usage goal. Furthermore, system goals demand one or more *quality attributes* [2]. Goals are decomposed into sub-goals, and goals can support another goal (contribute to it) or be in conflict with another goal (either directly stating the opposite or requiring a trade-off).

In the Facebook example below, the objectives in the three categories were elicited by considering the dimensions of sustainability and how they can be reflected with regard to the system.

Example
 Illustrative Facebook goals related to sustainability dimensions

- *Business goals*

    – *Long-term evolution (economic and technical sustainability)*
    – *ROI (economic sustainability)*
    – *Large market share (economic sustainability)*

- *Usage goals*

    – *Connect people (social sustainability)*
    – *Share content (individual and social sustainability)*
    – *Trigger communication (individual and social sustainability)*
    – *Spread news (individual, social, economic sustainability)*

- *System goals*

    – *High availability (individual, social, economic and technical s.)*
    – *High reliability (individual, social, economic and technical s.)*
    – *Long-term maintenance (social, economic and technical s.)*

As an example with explicit sustainability concerns, Fig. 7.6 depicts the goal model for the car-sharing system.

7.3.6   System Vision and Usage Model

From the goal model, we derive a sustainable system vision and the respective usage model.
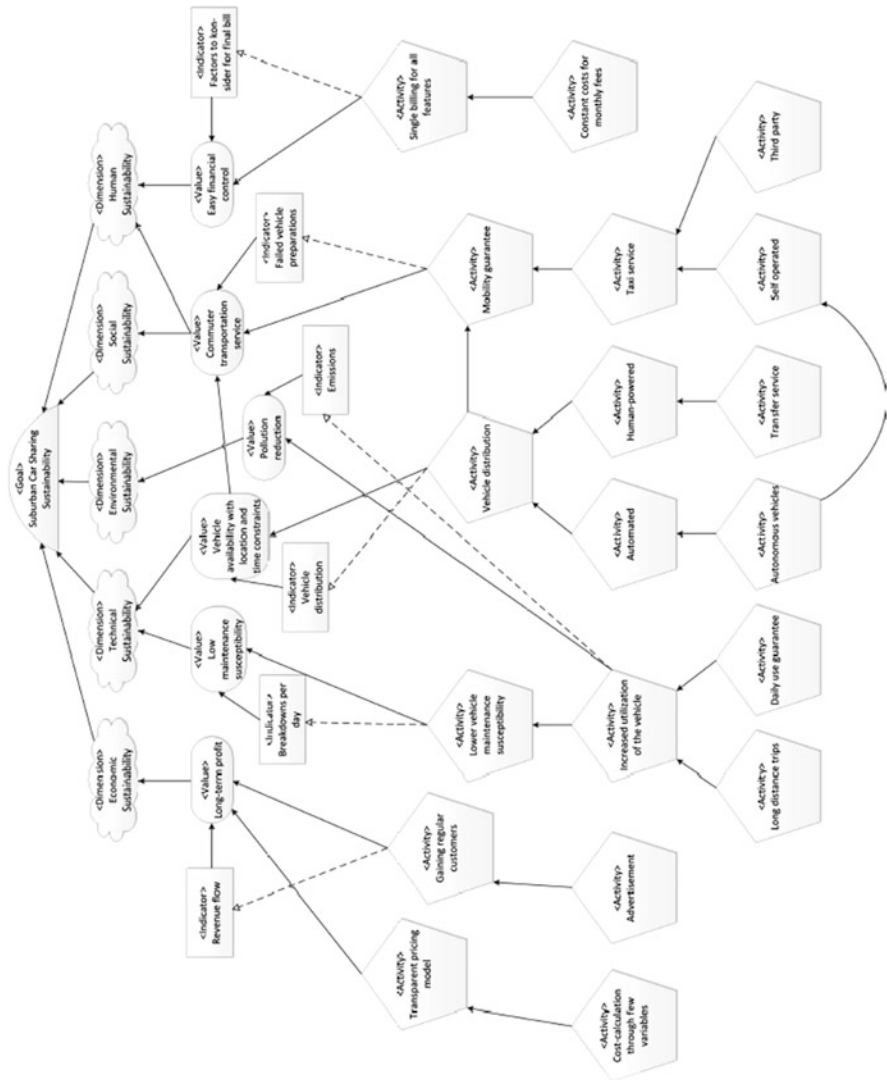
Fig. 7.6 Sustainability goals of the car-sharing case study [10]

7.3.6.1   System Vision

The system vision comprehends the system context of the *system under consideration*, which is intended to realise a number of *features*. A feature is, in our understanding, a prominent or distinctive user-recognisable aspect, quality or characteristic of a system that is related to a specific set of requirements, whose realisation enables the feature [7, 20].

Facebook does not have an explicit sustainability purpose. However, the vision could be extended such that the features are used for environmental causes, for example, organising green events or environmental campaigns. Instead of illustrating this extension, we provide two examples for systems that have an explicit sustainability reference in their system visions.

For the explicit sustainability purpose, examples are provided from the car-sharing system case study and the Citizen Muscle Boot Camp. Figure 7.7 depicts the system vision elaborated in discussion with various stakeholders from the respective industry domains. Figure 7.8 shows a system vision for the online course by the Story of Stuff Project.

7.3.6.2   Usage Model

This content item details a *use case overview* in its *use cases* and *scenarios*. We distinguish *services* and *use cases*. Both concepts are means to describe (black box) system behaviour. *Use cases* describe sequences of interaction between *actors* (*realising* user groups) and the system as a whole. More precisely, a use case represents a collection of interaction scenarios, each defining a set of interrelated actions that are executed either by an actor or by the system under consideration [8]. For each use case, there is at least one *functional scenario* in which *actors* participate. A scenario inherits from a requirement (not a whole use case), and each scenario is detailed into actions, which can be *actor actions* or *system actions*, each processing *data objects*.

Functional scenarios are triggered by events. Furthermore, we include *generic scenarios*, which serve for the satisfaction of *quality requirements* as they provide a means to specify generic interactions between actors and a system not necessarily motivated by business processes, such as maintenance activities an administrator performs. Use cases and scenarios can be represented in the form of structured text (e.g. the Cockburn template [8]), in UML use case diagrams, in diagrams and in message sequence charts.

As denoted earlier, the Facebook example does not have a specific sustainability purpose, but the service may well be used for environmental causes. Therefore, in the following example box, we specify a use case where a campaign for environmental sustainability is posted as an event.

Fig. 7.7 The car-sharing system vision

Example
Use case for creating a green event on Facebook
    Primary actor: standard content user
    Goal in context: the purpose of this feature is to enable users to create an event and to invite their friends to their activities
    Preconditions: in order to create an event, a user must have a Facebook account
    Trigger: user desires to create an event to be carried out either in real life or online

1. *User Joe reaches the events interface from the home page of Facebook*
2. *The system prompts him with a form for entering the event information*
3. *Joe enters the title 'Ban Plastic Bags' campaign*
4. *He sets time and date for the event and describes the challenge of convincing local shopkeepers to switch to reusable bags and encourage their customers to return with their own bags*
5. *Joe selects a picture of a reusable shopping bag as the event background picture and clicks a button to finish creating*

Fig. 7.8 The Story of Stuff Citizen Muscle Boot Camp Online Course System

6. *The system notifies Joe of the successful operation he has performed and prompts him to invite his friends*
7. *Joe invites all friends in his local group from the same town to join him in this campaign*

7.3.7 Specific Requirements Types

In the next refinement step for the system-level requirements, the information from goals, constraints and system vision is propagated into process requirements, deployment requirements, risk list, quality requirements and system constraints.

Quality requirements characterise specific quality attributes of the system (either coupled to a specific functionality or as a cross-cutting concern). They are usually represented in the form of natural text. Quality requirements are assessed by *measurements* that can be either a *normative reference* (e.g. a GUI style guide) or a *metric*. Quality requirements constrain *system actions* and can be satisfied by *generic scenarios*. We make use of quality definition models as by Deissenböck et al. [9].

Example
High-priority quality requirements for Facebook, issued by content user

- *High availability (economic, individual and social sustainability)*
- *High usability, easy to use (individual and social sustainability)*
- *Privacy (individual sustainability)*

The *system constraints* describe logical and technical restrictions on a system's architecture, its functionality by means of single atomic actions and its quality by means of assessable system quality requirements. We consider concepts that describe the transition to logical and technical architecture layers according to [42]. Hence, we see a system as a grey box rather than as a glass box, since we restrict systems' internals, but do not consider their logical structure by interacting components, interface specifications and functions. They are usually described in natural language text.

Example
System constraint for Facebook

- *The servers shall run in green data centres*

Process requirements constrain the content and/or structure of selected artefact types and the process model, that is, the definition of the milestones regarding time

schedules, used infrastructure like mandatory tools and compliance to selected standards and approaches like to the V-Modell XT. They are mostly described in natural language text.

> **Example**
> *Process requirement for Facebook: they might decide to use the approach provided in the book at hand*
>
> - *Develop the system according to the guidance provided by software engineering for sustainability*

The *risk list* includes a description of all risks that are related to project-specific requirements, usually in the form of natural language text. The conceptualisation of requirements risks is considered on the basis of an artefact model [16, 17]. The risks are implied by the various types of requirements, and we use the risk list as an interface to risk management.

> **Example**
> *Risk list for Facebook*
>
> - *Excessive content generation by users could cause high-energy demands with peaks that cannot be satisfied by renewable energy*
> - *Users are not active enough; therefore, no community would be established*

7.4   Discussion

In this section, we reflect on the mapping of sustainability dimensions to content items and discuss requirements conflicts, cost modelling, legal constraints and risk management.

7.4.1   Which Dimensions Appear in Which Content Items?

Independent of the applied artefact model, it is interesting to take a look at the mapping of sustainability dimensions to content items. Table 7.2 gives a coarse-grained mapping of requirements content items to the sustainability dimensions from which they contain information. For example, a system vision will generally include mainly environmental and social sustainability aspects (as it abstracts from technical details), the system constraints will feature many constraints from the environmental and technical sustainability dimensions and the process requirements will include demands from the environmental, social and technical dimensions.

Table 7.2 Requirements content items and their sustainability dimensions

Content item	Sustainability dimension
Stakeholder model	All
Goals	All
System vision	Mainly environmental and social
Usage model	Social and economic
Quality requirements	Technical
Deployment requirements	Technical
System constraints	Environmental and technical
Process requirements	Environmental, social and technical
Risk list	All

7.4.2   Sustainability Requirements Conflicts

In traditional qualities considered during software engineering, we already face a number of potential conflicts, for example, between code maintenance and code performance or between the development time and the desired quality of a software system. Consequently, the question arises what kinds of conflicts exist between the five dimensions of sustainability and their related goals.

The economic dimension aligns with the environmental one in terms of resource savings (energy, materials, waste), but they may conflict when it comes to additional certifications, building a (environmentally and socially) sustainable supply chain and turning to more expensive alternative solutions in case they are more environmentally friendly. The reason for this is mainly that up to now, the negative environmental impacts that are caused by our economy are hardly charged. Therefore, the goal of environmental sustainability does not get assigned monetary value but only image value, which is likely to be ranked second. These conflicts are also discussed in [36].

Another potential conflict, at least for some systems, is a trade-off between energy efficiency and dangerous materials. This is one potential goal conflict in case energy efficiency would require using more dangerous materials. Although not a software system in itself, a light bulb might serve as an example: New energy-saving lamps are much more energy efficient than the old light bulbs but at the same time contain toxic mercury that imposes a threat when a lamp breaks, similarly as phenol, naphthalene and styrene. In the case at hand, considerate users will make sure the lamp is not in use and is in close proximity to their heads, but as legislation has already banned the old light bulbs in some countries, they will have to be used for now. Resolving such a conflict for a particular case means to assign weights to each of the goals and prioritise whether energy saving is greater or whether the risk and long-term negative impacts of the dangerous materials are greater.

7.4.3 Cost Modelling

Another aspect worth discussing is the connection between stakeholders, goals and cost modelling. The stakeholders are made explicit in the goal model by tracing back to the issuer of a goal, as the information source (e.g. a domain expert) or the issuer of a goal. With respect to assigning costs to the goals, there is a limitation, as this only makes sense for business goals, but not for values that cannot be expressed in return on investment. Some goals, for example, the protection of the environment, do not have a monetary value in themselves, and their qualitative value is hard to measure. At the same time, it is important to define measures to ensure the realisation of these goals and to show that the approach can make a difference in those resulting measures. Consequently, instead of assigning costs to the sustainability goals, their contribution to higher causes must be made explicit, for example, the contribution to objectives commonly agreed on by governments like the sustainable development goals from Rio+20[6] or Vision 2050 [43].

7.4.4 Legal Constraints

As a consequence to the fact that environmental goals have not yet been prioritised sufficiently by the economy, legislation has established a number of environmental regulations that companies have to adhere to. These regulations will still be extended in the future, which makes legislation probably the most important stakeholder representing especially environmental sustainability. Individual and social sustainability are also taken care of by law, for example, by workers' rights, which are supported and represented by worker unions.

It would be interesting to see at which point we need new laws and a different legislation to make sure that important questions of sustainability are incorporated into IT systems. Furthermore, it would be interesting to look at other examples such as functional safety and, also to a certain extent, security, where such laws exist.

7.4.5 Risk Management and Environmental Sustainability

Risks, safety and security all strongly relate to sustainability: risks need to be managed in order to enable sustainability, and safety and security are part of sustainability.

Safety is part of individual and social sustainability for preserving human life (no injuries) and the environment (no chemical or other hazardous accidents) but

---

[6] http://www.uncsd2012.org/

also has aspects in economic sustainability (a product that is not safe will not let a company reach long-term economic goals).

Security is also part of various dimensions: the technical dimension (as it is a standard quality attribute for systems), then the individual and social dimensions (as the users shall be protected) and, as a consequence of that, also the economic dimension (insecure systems will not have market success).

Both of these quality aspects have not been around forever, but they were introduced as explicit qualities for software systems after the first safety hazards and the first security threats occurred. Consequently, we can learn from this development for systematically incorporating sustainability into software engineering [40].

7.5 Conclusion

This chapter provided an overview of how green requirements engineering may be conducted within the scope of general-purpose requirements engineering by asking a few guiding questions along the way and providing plugs for additional analysis activities that inform the development of environmental issues that should be considered. This approach was supported by illustrating examples and a discussion on the different types of conflicts and traceability of information across different requirements engineering content items.

The impact of our contribution is mainly determined by the question of how much difference the consideration of sustainability actually makes in requirements engineering. If we can make a sustainability purpose explicit in a system, then the difference is significant. If such a purpose is not given, a secondary influence can be achieved by adding sustainability objectives and greening the system itself. The latter has less impact on the environment but is still feasible, especially if the system has a big user community. In the long run, the author's hypothesis is that we will not be able to end resource depletion by greening existing systems but only by disruptive change and completely transforming our systems. However, creating the mindset for that starts with acknowledging the need for incorporating sustainability as an explicit quality objective in systems development.

One open issue is the standardisation of (environmental and general) sustainability as an explicit quality objective in software development, for example, within the IEEE 830 recommendation for software requirements specifications and the ISO 25000 on software quality, informed by the ISO standard families on environmental management [14] and social responsibility [15]—maybe this book can provide a basis for triggering such a standardisation.

The path towards software engineering for sustainability (SE4S)[7] requires a mindset of awareness (by business analysts and developers), methodical guidance (as provided in this chapter) and creativity. For the latter, we need creative

---

[7] http://www.se4s.org

confidence [22] to establish the right mindset for transition engineering [23] that enables us to move towards a more sustainable global society as illustrated in Vision 2050 [43], supported by adequate software systems.

References

1. Alexander I, Robertson S (2004) Understanding project sociology by modeling stakeholders. IEEE Software 21(1):23–27
2. Boegh J (2008) A new standard for quality requirements. IEEE Software 25(2):57–63
3. Bridges W (1995) Managing transitions: making the most of change paperback. Nicholas Brealey, Boston; 3rd revised edition (3 Dec 2009)
4. Brown B, Hanson M, Liverman D, Merideth R (1987) Global sustainability: toward definition. Environ Manag 11(6):713–719
5. Broy M, Feilkas M, Herrmannsdoerfer M, Merenda S, Ratiu D (2010) Seamless model-based development: from isolated tools to integrated model engineering environments. Proc IEEE 98 (4):526–545, available at http://dx.doi.org/10.1109/JPROC.2009.2037771
6. Broy M, Stoelen K (2001) Specification and development of interactive systems: focus on streams, interfaces, and refinement. Springer, New York
7. Classen A, Heymans P, Schobbens PY (2008) What's in a feature: a requirements engineering perspective. In: Fiadeiro J, Inverardi P (eds) Proceeding of the 11th international conference on fundamental approaches to software engineering (FASE 08) in conjunction with ETAPS 08, no. 4961 in FASE/ETAPS. Springer, Berlin, pp 16–30
8. Cockburn A (2000) Writing effective use cases. Addison-Wesley Longman, Boston, MA. ISBN 13: 978-0201702255
9. Deissenboeck F, Juergens E, Lochmann K, Wagner S (2009) Software quality models: purposes, usage scenarios and requirements. In: Proceedings of the 7th international workshop on software quality (WoSQ 09). IEEE Computer Society Press, p N/A
10. Feldmann O (2012) Sustainability aspects in specifying a car sharing platform. Bachelor's thesis, Technische Universität, München
11. Fernández DM, Lochmann K, Penzenstadler B, Wagner S (2011) A case study on the application of an artefact-based requirements engineering approach. In: 15th international conference on evaluation and assessment in software engineering
12. Glinz M, Wieringa RJ et al (2007) Guest editors' introduction: stakeholders in requirements engineering. IEEE Software 24(2):18–20, http://doi.ieeecomputersociety.org/10.1109/MS.2007.42
13. Hilty L, Lohmann W, Huang E (2011) Sustainability and ICT—an overview of the field. Politeia 27(104):13–28
14. International Standardization Organization (2004) ISO 14000 – environmental management. http://www.iso.org/iso/home/standards/management-standards/iso14000.htm
15. International Standardization Organization (2010) ISO 26000 Guidance on social responsibility. http://www.iso.org/iso/home/standards/iso26000.htm

16. Islam S (2009) Software development risk management model – a goal driven approach. In: Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundation of software engineering (ESEC/FSE). ACM, New York, pp 5–8
17. Islam S, Houmb S, Mendez Fernandez D, Joarder M (2009) Offshore-outsourced software development risk management model. In: Proceedings of the 12th IEEE international conference on computer and information technology (ICCIT 09), pp 514–519
18. Jackson M (1995) Software requirements and specifications: a lexicon of practice, principles and prejudices. Addison Wesley, Reading, MA
19. Johansson B, Skoogh A, Mani M, Leong S (2009) Discrete event simulation to generate requirements specification for sustainable manufacturing systems design. In: Proceedings of the 9th workshop on performance metrics for intelligent systems, PerMIS '09. ACM, New York, pp 38–42. doi:10.1145/1865909.1865918, URL http://doi.acm.org/10.1145/1865909.1865918
20. Kang K, Cohen S, Hess J, Nowak W, Peterson S (1990) Feature-oriented domain analysis (FODA) feasibility study. Tech. Rep. CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University Pittsburgh, PA
21. Karlin B, Penzenstadler B, Cook A (2014) Pumping up the Citizen Muscle Bootcamp: improving user experience in online learning. In: 16th international conference on human-computer interaction
22. Kelley T, Kelley D (2013) Creative confidence: unleashing the creative potential within us all. Crown Business, New York
23. Krumdieck S (2011) The survival spectrum: the key to transition engineering of complex systems. In: Proceedings of the ASME international mechanical engineering congress & exposition
24. Hilty L et al (2006) The relevance of information and communication technologies for environmental sustainability. Environ Model Software 21(11):1618–1629. doi:10.1016/j.envsoft.2006.05.007, URL http://www.sciencedirect.com/science/article/pii/S1364815206001204
25. van Lamsweerde A (2009) Requirements engineering: from system goals to UML models to software specifications. Wiley, New York. ISBN 13: 978-0470012703
26. Mahaux M, Heymans P, Saval G (2011) Discovering sustainability requirements: an experience report. In: 17th international working conference on requirements engineering: foundation for software quality
27. Meadows D (1997) Leverage points – places to intervene in a system. http://www.donellameadows.org/wp-content/userfiles/Leverage_Points.pdf (1999). A shorter version of this paper appeared in Whole Earth, Winter 1997
28. Mendez D, Penzenstadler B (2014) Artefact-based requirements engineering: the AMDiRE approach. Requirements Eng J (to appear 2014)
29. Méndez Fernández D, Penzenstadler B, Kuhrmann M, Broy M (2010) A meta model for artefact-orientation: fundamentals and lessons learned in requirements engineering. In: Model driven engineering languages and systems, vol 6395, pp 183–197
30. Monk A, Howard S (1998) Methods & tools: the rich picture: a tool for reasoning about work context. Interactions 5(2):21–30
31. Nuseibeh B (2001) Weaving together requirements and architectures. Computer 34(3):115–117. doi:10.1109/2.910904, URL http://dx.doi.org/10.1109/2.910904
32. Nuseibeh B, Easterbrook S (2000) Requirements engineering: a roadmap. In: Proceedings of the conference on the future of software engineering. ACM, New York, pp 35–46
33. Penzenstadler B (2012) Supporting sustainability aspects in software engineering. In: 3rd international conference on computational sustainability (CompSust)
34. Penzenstadler B, Eckhardt J, Fernandez DM (2013) Two replication studies for evaluating artefact models in RE: results and lessons learnt. In: Proceedings of the 3rd international workshop

on replication in empirical software engineering research (RESER '13), IEEE, 2013, Baltimore, MD

35. Penzenstadler B, Femmer H (2012) A generic model for sustainability. Tech. rep., Technische Universität, München

36. Penzenstadler B, Femmer H (2013) A generic model for sustainability with process and product-specific instances. In: First international workshop on green in software engineering and green by software engineering

37. Penzenstadler B, Femmer H, Richardson D (2013) Who is the advocate? Stakeholders for sustainability. In: 2nd international workshop on green and sustainable software (GREENS), at ICSE

38. Penzenstadler B, Femmer H, Richardson D (2013) Who is the advocate? Stakeholders for sustainability. In: 2nd international workshop on green and sustainable software (GREENS), at ICSE, San Francisco, CA

39. Penzenstadler B, Fernandez DM, Eckhardt J (2013) Understanding the impact of artefact-based RE – design of a replication study. In: Proceedings of the 7th international symposium on empirical software engineering and measurement (ESEM '13), IEEE, 2013, Baltimore, MD

40. Penzenstadler B, Raturi A, Richardson D, Tomlinson B (2014) Safety, security, now sustainability: the non-functional requirement of the 21st century. IEEE Software Spec Issue Green Software

41. Raturi A, Penzenstadler B, Tomlinson B, Richardson D (2014) Developing a sustainability non-functional requirements framework. In: under review for GREENS'14

42. Wiegers K (2003) Software requirements, 2nd edn. Microsoft Press, Redmond, WA. ISBN 13: 978-0735618794

43. World Business Council for Sustainable Development: Vision 2050 (2010) A new agenda for business. http://www.wbcsd.org/WEB/PROJECTS/BZROLE/VISION2050-FULLREPORT_FINAL.PDF

44. Yu E (2011) Modelling strategic relationships for process reengineering. Soc Model Requirements Eng 11

45. Zave P (1997) Classification of research efforts in requirements engineering. ACM Comput Surv 29(4):315–321

Chapter 8
Towards Green Software Testing

Macario Polo

8.1 Introduction

Along the life cycle, testing activities are needed time and time again: during the initial development, to detect and fix errors in the first release, and later, depending on the maintenance type, both to detect possible errors introduced in new functionalities and to check that the previous version remains stable after the maintenance intervention. So, testing is an essential workflow to ensure software quality, though it is also time consuming, costly and energy demanding. This chapter discusses how different approaches of test design, test execution and the selected test requirement may impact the costs related to testing. The chapter also includes a theoretical model about the consumption of energy depending on the selected approach.

8.2 Test Requirements

Software testing consists of a series of activities which are carried out along the whole software life cycle. It is a destructive process whose main goal is to find errors within the system under test ('SUT'). Software testing has three main steps:

1. Test design
2. Test execution
3. Result analysis

M. Polo (✉)
Department of Information Technologies and Systems, University of Castilla-La Mancha, Ciudad Real, Spain
e-mail: Macario.Polo@uclm.es

Test design is carried out taking into account one or more test requirements which should be fulfilled when test cases are executed. In general terms, a test requirement refers to some coverage criterion that measures how the test cases run through some attribute of the SUT: executing all the statements in the source code, all the decisions and all the conditions; reaching MC/DC (modified condition/decision); killing all the source code mutants; executing all the scenarios of all the functional requirements; etc.

The size of the test suite strongly depends on the selected test requirement; in fact, it is very different to calling once each method in a class that executes with true and with false every condition in a decision statement.

Consider, for example, the code in Fig. 8.1, which is a Java implementation of the well-known Triangle-type determination problem [5]: it consists of a class whose constructor receives three integer numbers as parameters, which are the lengths of the three sides of a triangle.

The *calculateType* method assigns the *triType* field a number between 1 and 4 representing whether the object corresponds to a scalene, isosceles or equilateral triangle or whether it is not a triangle. This problem is commonly used for teaching testing, and very often it has been used in software testing literature as a common benchmark to evaluate testing techniques. The idea is to write test cases to get a *complete* test of the class.

The concept of *completeness* varies from a coverage criterion to another. For example, if a tester considers to get all the possible outputs of the program (which corresponds to the Myers cause–effect graph technique), just the four test cases in Fig. 8.2 are required: if they are executed with JUnit, all of them give a pass verdict, and, thus, the tester could be convinced of the SUT quality.

However, these four test cases do not visit or only partially visit the lines highlighted in Fig. 8.3. By *partial visit*, we mean that not all possible conditions of a decision are evaluated: for example, the second condition $(x+y>z)$ of the decision $(triType==1 \ \&\&x+y>z)$ is only evaluated when the first one $(triType==1)$ is true, because both conditions are separated by an *and* logical connector.

The same test suite, if executed with a mutation tool, gets a mutation score of 69.29 %, which is far from the desired 100 %. The first mutant generated by the Bacterio tool [12], for example, holds a fault which remains undiscovered (Fig. 8.4).

8.3 Impact of the Test Requirement

A coverage criterion *C1* subsumes another criterion *C2* if for every program, any test set *T* that satisfies *C1* also satisfies *C2* [3]: going through all the program statements subsumes going through all the methods. Except in the case of an improbable program with no branches, the number of test cases required for the latter is less than in the former.

```java
package benchmarks;

public class Triangle
{
    private int x, y, z;
    private int triType;
    public static int SCALENE = 1, ISOSCELES = 2, EQUILATERAL = 3, NOT_A_TRIANGLE = 4;

    public Triangle(int x, int y, int z) {
        this.x=x; this.y=y; this.z=z;
        this.triType=0;
    }

/**
 *
 * @return 1 if scalene; 2 if isosceles; 3 if equilateral; 4 if not a triangle
 */
    public void calculateType() {
        if (x==y) {
            triType=triType+1;
        }
        if (x==z) {
            triType=triType+2;
        }
        if (y==z) {
            triType=triType+3;
        }

        if (x<=0 || y<=0 || z<=0) {
            triType=NOT_A_TRIANGLE;
            return;
        }
        if (triType==0) {
            if (x+y<=z || y+z<=x || x+z<=y) {
                triType=NOT_A_TRIANGLE;
                return;
            } else {
                triType=SCALENE;
                return;
            }
        }
        if (triType>3) {
            triType=EQUILATERAL;
        } else if (triType==1 && x+y>z) {
            triType=ISOSCELES;
        } else if (triType==2 && x+z>y) {
            triType=ISOSCELES;
        } else if (triType==3 && y+z>x) {
            triType=ISOSCELES;
        } else {
            triType=NOT_A_TRIANGLE;
        }
    }

    public int getType() {
        return this.triType;
    }
}
```

Fig. 8.1 A class implementing the Triangle-type determination problem [5]

Besides the white box coverage criteria (statements, MC/DC, mutation score, etc.), it is often interesting to measure the degree in which parameter values are used in test cases.

Suppose, for illustration, a single class which offers a function to convert magnitudes between different measures (left side of Fig. 8.5). Its convert (sourceUnit:String, targetUnit:String, magnitude:double):double function is capable of converting temperatures (Celsius, Fahrenheit, Kelvin), lengths (Meters, Yards, Inches, Kilometers, Miles) and weights (Kilograms, Pounds, Ounces). With these

```java
public class TriangleTestAllOutputs extends Test-
Case {
  public void testEQUILATERAL() {
      Triangle t=newTriangle(5, 5, 5);
      t.calculateType();
      assertTrue(t.getType()==Triangle.EQUILATERAL);
  }

  public void testISOSCELES1() {
      Triangle t=newTriangle(5, 5, 6);
      t.calculateType();
      assertTrue(t.getType()==Triangle.ISOSCELES);
  }

  public void testSCALENE() {
      Triangle t=newTriangle(5, 4, 6);
      t.calculateType();
      assertTrue(t.getType()==Triangle.SCALENE);
  }

  public void testNOT_A_TRIANGLE1() {
      Triangle t=newTriangle(5, 5, 10);
      t.calculateType();
      assert-
True(t.getType()==Triangle.NOT_A_TRIANGLE);
  }
}
```

Fig. 8.2 A test suite for the Triangle problem, built with the cause–effect technique

values in `Courier font` for the source and target units, some numerical values for the magnitude and a single spreadsheet, one can quickly generate a good number of combinations to test the function (right side of Fig. 8.5).

For this very simple function, the problem is the huge number of test cases which can be generated: there are 11 fixed values for `source` and `target`, one additional for invalid units (e.g. `nautical mile` or `pear` or `apple`) and at least 13 values to consider the boundary values (e.g. -273 °C, which is the Kelvin absolute zero) and representative values of the equivalence classes of `magnitude`. With these values, a completely exhaustive testing would produce $12 \times 12 \times 13 = 1{,}872$ test cases.

This strategy (generating all combinations) is almost never used due to the huge number of test cases produced, even for testing small programs or functionalities (consider that, besides the test case generation, tests must be executed and enriched with an oracle, that is, one or more statements to check whether the system passes or fails each test case).

For input parameter values, there exist different coverage criteria. Since the goal of testing is finding errors and many of them appear when the program is executed with unforeseen value combinations, common criteria are *pairwise* and *n-wise*. Pairwise is fulfilled when all the pairs of values of any two parameters are used at least in a test case. If we talk of tuples or *n* elements instead of pairs, we will get

```
public void calculateType() {
    if (x==y) {
        triType=triType+1;
    }
    if (x==z) {
        triType=triType+2;
    }
    if (y==z) {
        triType=triType+3;
    }

    if (x<=0 || y<=0 || z<=0) {
        triType=NOT_A_TRIANGLE;
        return;
    }
    if (triType==0) {
        if (x+y<=z || y+z<=x || x+z<=y) {
            triType=NOT_A_TRIANGLE;
            return;
        } else {
            triType=SCALENE;
            return;
        }
    }
    if (triType>3) {
        triType=EQUILATERAL;
    } elseif (triType==1 &&x+y>z) {
        triType=ISOSCELES;
    } elseif (triType==2 &&x+z>y) {
        triType=ISOSCELES;
    } elseif (triType==3 &&y+z>x) {
        triType=ISOSCELES;
    } else {
        triType=NOT_A_TRIANGLE;
    }
}
```

Fig. 8.3   Visited, unvisited and partially visited statements by test cases in Fig. 8.2

```
public Triangle(int x, int y, int z) {        public Triangle(int x, int y, int z) {
    this.x = x;                                   this.x = Math.abs(x);
    this.y = y;                                   this.y = y;
    this.z = z;                                   this.z = z;
    triType = 0;                                  triType = 0;
}                                             }
```

Fig. 8.4   The first mutant of the Triangle class (*right*) has a fault inserted by the ABS mutation operator

Converter

```
+Converter()
+convert(sourceUnit : java.lang.String, targetUnit : java.lang.String, magnitude : double) : double
-convertMeters(magnitude : double, targetUnit : java.lang.String) : double
-toKgs(sourceUnit : java.lang.String, magnitude : double) : double
-fromKgs(magnitude : double, targetUnit : java.lang.String) : double
-toMeters(sourceUnit : java.lang.String, magnitude : double) : double
-fromCelsius(magnitude : double, targetUnit : java.lang.String) : double
-toCelsius(origen : java.lang.String, unidades : double) : double
```

	A	B	C	D
1	sourceUnit	targetUnit	magnitude	Expected result
2	C	C	0	
3	C	C	100	
4	C	C	-100	
5	C	C	-300	
6	C	F	0	
7	C	F	100	
8	C	F	-100	
9	C	F	-300	
10	C	K	0	
11	C	K	100	
12	C	K	-100	
13	C	K	-300	
14	K	C	0	
15	K	C	100	
16	K	C	-100	
17	K	C	-300	

Fig. 8.5 The converter class and some test data combinations

n-wise coverage. For example, the `convert` function tested with pairwise requires only 172 test cases (less than 10 % than all combinations).

For a given set of parameter values, *all combinations* is the generation strategy with the highest chance to find errors but also the one which produces more test cases, usually of an unmanageable size. Although the size of *pairwise* suites is lower and their chance of finding errors is high, it is still lower than *all combinations*. In general, the most effective technique for cost saving is the reduction of test suites, whilst a given white box coverage is preserved. We will deal hereinafter with this.

Actually, a coverage criterion can be considered of *white box* depending on the 'zoom' used to test the system: the state machine of a system is a white box view of such a system, even though not all the branches in the code of the operations called by transitions or states are called. Consider the *Account::transfer* operation in Fig. 8.6, which transfers the *amount* passed as the first parameter to *targetAccount* passed as the second: proposing a test case traversing that transition does not imply that all the statements in the code are reached.

8.4 Understanding the Cost of Test Execution

Readers who have visited repositories of big projects may have seen the commonly called *nightly builds* (Fig. 8.7). They are nearly deployable versions of the respective systems that are ready to be tested. Since automated test execution may be an unattended process, which does not require any human intervention, test cases are often executed at night; the next morning, testers receive the result report, and if errors are present, they are reported to the developers.

Since the goal of testing is to find errors in the SUT, at a first glance, a test case is better than another if the former finds more errors than the latter or, in case no errors are found, if the former reaches more coverage than the latter. For this, it is

```
public class Account {
    ...
    private double balance;

    public int transfer(double amount,
                        Account other) {
        if (balance-amount>=0) {
            if (other.isBlocked())
                return BLOCKED;
            if (pendingCharges())
                return PENDING_CHARGES;
            other.deposit(amount);
            this.withdraw(amount);
            return SUCCESS;
        } else {
            return INSUFFICIENT_BALANCE;
        }
    }
}
```
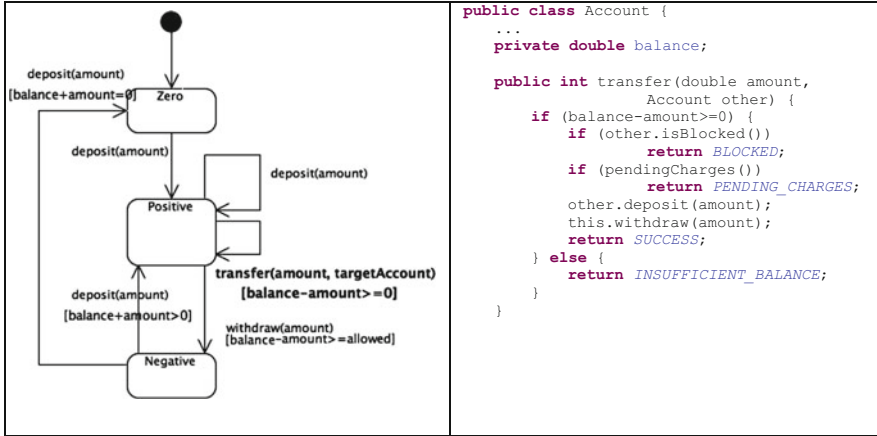
Fig. 8.6 Behaviour of a supposed banking account
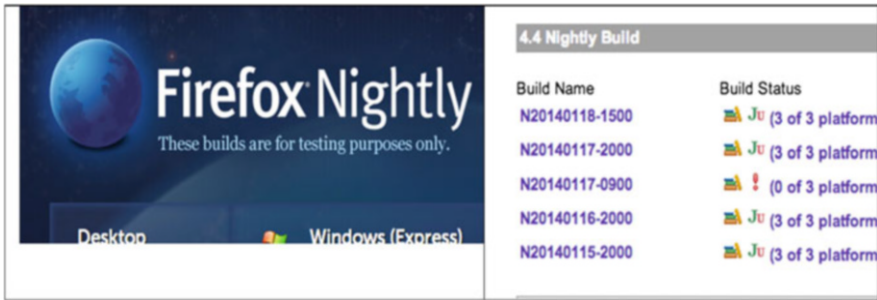
Fig. 8.7 Nightly builds in the websites of Firefox and Eclipse

important, however, to take into account the presence of redundant test cases and the intersections of the errors discovered or the areas covered by the test cases. Actually, in regression testing, these two characteristics of the test cases are strongly considered for selecting the test cases that should be included in a regression test suite.

In regression testing, old test cases are re-executed against the SUT when a new version has been released: their goal is to find faults that the maintenance intervention has likely introduced. Since test cases must be executed but also maintained and kept updated in order to consider the new characteristics, function parameters, etc., of the new version, it is important to keep them in test suites of a reasonable size—ideally just a few test cases—but with the same ability to find errors of a large test suite.

Mutation [1] can help us to understand these ideas. A mutant of the SUT is a copy of the SUT itself which holds an artificial change inserted in it. These changes are inserted by tools via mutation operators that try to imitate common faults that programmers may commit when they write their programs. The most used

Fig. 8.8 One of the equivalent mutants obtained for the Triangle problem

operators are AOR (arithmetic operator replacement), ROR (relational operator replacement), UOI (unary operator insertion), ABS (absolute value insertion) and LCR (logical connector replacement) [6]. Some of the changes these operators insert actually produce faulty versions of the SUT, whilst others are code optimisations or de-optimisations which are impossible to detect by test cases: the statement highlighted in Fig. 8.8, for example, is one of these 'equivalent mutants'.

Mutation has traditionally been used to evaluate the quality of test cases: in fact, P being a program, $M = \{M1, \ldots M100\}$ a set of 100 faulty versions of a program, and TS_A and TS_B two test suites for P, TS_A will be better than TS_B (with respect to mutation) if it finds more faults on M than TS_A (i.e. in mutation vocabulary, if TS_A kills more mutants than TS_B). The quality of a test suite is measured as the mutation score, which is the percentage of nonequivalent mutants killed (Fig. 8.9).

From the execution of a test suite against a set of mutants, the tester gets what is called a 'killing matrix': rows are labelled with mutants and columns with test cases. A filled-in cell represents that the test case in the column has found the fault inserted in the mutant in the row (i.e. the test case has killed the mutant). For the sake of understanding, Fig. 8.10 shows the two killing matrices of two test suites that exercise the same SUT. Assuming that equivalent mutants have been removed, TS_A is better than TS_B because their respective mutation scores are 0.9 and 0.4. However, TS_B finds the fault inserted in m_{10}, which is not discovered by the other.

Regarding individual test cases, tA_2 is the best in both test suites, since it finds five of the ten artificial errors. Note also that tB_3 is a useless test case, since it does not discover any error. On the other hand, m_7 is a very good fault, in the sense that it has not been discovered by any test case.

A very interesting observation is in the presence of redundant test cases: from the mutation testing point of view, a test case is redundant with respect to another if all the mutants it kills are also killed by another test case; so, $K(t)$ being the set of mutants killed by test case t, t_i is redundant with respect to t_j if $K(t_i) \subseteq K(t_j)$. In the test suite TS_A of Fig. 8.8, tA_3 subsumes tA_4, since the two mutants killed by the latter (m_4 and m_6) are also killed by the former (m_4, m_5 and m_6). In TS_B, no test case subsumes any other.

This approach is extensible to any other test requirement or coverage criterion: think about source code lines, conditions or decisions: a *test requirement/test case* matrix can be built to extract a reduced-size test suite. Since test cases must be

Fig. 8.9 Mutation score

$$MS(P,T) = \frac{K}{M-E}$$

P : program under test
T : test suite
K : number of killed mutants
M : number of generated mutants
E : number of equivalent mutants

Fig. 8.10 Possible killing matrices of test suites TS_A and TS_B

	TS$_A$					TS$_B$			
	tA$_1$	tA$_2$	tA$_3$	tA$_4$		tB$_1$	tB$_2$	tB$_3$	tB$_4$
m1	X	X			*m1*	X			
m2	X	X			*m2*				
m3		X			*m3*		X		
m4			X	X	*m4*		X		X
m5			X		*m5*				X
m6		X	X	X	*m6*				
m7					*m7*				
m8	X	X	X		*m8*				
m9	X				*m9*				
m10					*m10*		X		

executed, validated and maintained, it is essential to keep under control the test requirement achieved by each test case.

8.4.1   Algorithms for Test Suite Reduction

The optimal test suite reduction problem (i.e. the problem of extracting a subset TS^R of a test suite TS whose size is the minimum possible whilst the test requirements of TS^R are the same as TS) is NP-hard, and thus, methods for test suite reduction are based on greedy algorithms, which find good solutions in a polynomial time.

In a recent paper, Polo et al. [11] describe an effective algorithm to reduce a test suite based on mutation. Consider the killing matrix on the left side of Fig. 8.10: this greedy algorithm starts by adding the test case in TS_A that kills more mutants than TS^R_A (in this example, tA_2); then, it removes $K(tA_2)$ from the set of mutants and continues iterating with the next test case killing more mutants. The algorithm stops when all the mutants TS are also killed by TS^R. Thus, applied to that killing matrix, the algorithm evolves according to Fig. 8.11.

But the test requirement for a test case selection can be any one: coverage of sentences, blocks, paths, number of mutants killed or, even, a combination of two or more. In this sense, Harrold, Gupta and Sofa [2] give a greedy algorithm (usually referred to as *HGS*) for reducing the suite of test cases into another one, preserving

	tA$_1$	tA$_2$	tA$_3$	tA$_4$
m1	X	X		
m2	X	X		
m3		X		
m4			X	X
m5			X	
m6		X	X	X
m7				
m8	X	X	X	
m9	X			
m10				

$TS^R = \{tA_2\}$

	tA$_1$	tA$_3$	tA$_4$
m4		X	X
m5		X	
m7			
m9	X		
m10			

$TS^R = \{tA_2, tA_3\}$

	tA$_1$	tA$_4$
m7		
m9	X	
m10		

$TS^R = \{tA_2, tA_3, tA_1\}$

	tA$_4$
m7	
m10	

$TS^R = \{tA_2, tA_3, tA_1\}$

Fig. 8.11 Getting a reduced test suite from a whole killing matrix

the fulfilment of the test requirements (in general, two different coverage criteria) reached by the original suite. The main steps of this algorithm are as follows:

1. Initially, all the test requirements are unmarked.
2. Add to TS^R those test cases that only exercise a test requirement. Mark the requirements covered by the selected test cases.
3. Order the unmarked requirements according to the cardinality of the set of test cases exercising one requirement. If several requirements are tied (since the sets of test cases exercising them have the same cardinality), select the test case that would mark the highest number of unmarked requirements tied for this cardinality. If multiple such test cases are tied, break the tie in favour of the test case that would mark the highest number of requirements with testing sets of successively higher cardinalities; if the highest cardinality is reached and some test cases are still tied, arbitrarily select a test case from among those tied. Mark the requirements exercised by the selected test. Remove test cases that become redundant as they no longer cover any of the unmarked requirements.
4. Repeat the above step until all testing requirements are marked.

   Gupta continued improving this algorithm with other collaborators:

- With Jeffrey [3], he added 'selective redundancy' to the algorithm. 'Selective redundancy' makes it possible to select test cases that, for any given test requirement, provide the same coverage as another previously selected test case, but that adds the coverage of a new, different test requirement. Thus, maybe T^R reaches the all-branches criterion but not def-uses; therefore, a new test case t can be added to T^R if it increases the coverage of the def-uses requirement: now, T^R will not increase the all-branches criterion, but it will do with def-uses.

Table 8.1 Reduction of the test suite size in some small benchmark programs

Program	LOC	# of mutants	|TS|	|TSR|
Bisect	31	44	25	2 (8 %)
Fourballs	47	168	96	5 (5.2 %)
Mid	59	138	125	5 (4 %)
TriTyp	61	239	216	17 (7.8 %)

- With Tallam [13], test case selection is based on concept analysis techniques. According to the authors, this version achieves the same size or smaller size reduced test suites than prior heuristics as well as a similar time performance.

As a proof of the impact of test suite reduction on cost savings, Table 8.1 shows some reductions for several small programs that have commonly been used in software testing literature [6–10]: the third column represents the size of the original test suite used to test the corresponding program; the fourth is the size of the reduced suite.

8.4.2   Test Case Execution Algorithms

For illustrative purposes only, Fig. 8.12 shows an excerpt of the killing matrix produced by a test suite composed by 343 automatically generated test cases for the Triangle-type determination problem. For this figure, we generated 114 nonequivalent mutants using the Bacterio tool. In Bacterio, cells with X represent killed mutants, and cells with O denote that the corresponding test case has visited, but not killed, the statement mutated in the corresponding mutant. To produce this matrix, Bacterio has executed each of the 343 test cases against the 114 mutants generated: this means that it has made $343 \times 114 = 39{,}102$ executions.

We already know that test suite reduction is an excellent tool to avoid the costly tasks of test case execution and re-execution. To allow reduction algorithms (such as those reviewed in the previous section) to get their best results, a complete execution of all test cases against all mutants is required. Note in the zoomed part of Fig. 8.9 that the mutant 100 is 'killed and killed and again killed' by one test case and another: from the mutation testing point of view, if a test case discovers the fault inserted in a mutant, this one can be removed from the mutant set, thus avoiding the execution of test cases that attempt to find previously discovered faults. From any other testing criterion point of view, if a test case already exercises a certain area of the program under test, writing a test to go over that very same area could be avoided (when feasible).

Without leaving the killing matrices, there exist different execution algorithms that fill in the matrix in several ways. The goal of these algorithms is to decrease the cost of test case execution by preventing the execution of test cases against already killed mutants. In the context of any other test requirement, the idea is to prevent the execution of test cases to reach a previously fulfilled test requirement. In the

Fig. 8.12 A fragment of the killing matrix for a test suite for the Triangle problem and a detail

highlighted part of Fig. 8.12 and since the first test case kills *version100*, all the subsequent executions of test cases against this version could be avoided.

Good execution algorithms fill in the *test requirement/test case* matrix by rows. Revisiting the example of the TS_A test suite in Fig. 8.10, the testing tool executes tA_1 against m_1; since it is killed, the tool stops launching more test cases against it and takes m_1 out from the mutant set. The same happens with m_2, m_8 and m_9: they are killed and no more attempts to kill them will be made. In a second iteration, the testing tool starts to execute tA_2 directly against m_3. The process continues this way until all mutants are attempted to be killed. In Fig. 8.13, only 24 executions are required instead of the 40 of a full test.

When applying this execution algorithm to the Triangle problem, filling in the killing matrix by rows requires 1,037 test case executions instead of 39,102 (i.e. it reduces down to 2.66 %). Moreover, a greedy algorithm may be applied to reduce the test suite size. Bacterio implements one of these algorithms, obtaining that only 27 test cases are required to get the same mutation score results than the 343 original test cases. So, for regression testing, the tester may only consider the 27 test cases selected. Moreover, if he or she executes them against the system with a time-saving execution algorithm for filling in by rows, only 335 executions will be needed, being 0.08 % of the original number (39,102). The new application of the same reduction algorithms will not improve the results at all.

Although our running example is small, redundancy of test cases or overlapping of test requirements is common also in large systems. Thus, an adequate policy of test case execution may reduce regression testing costs by almost 99 %, with the corresponding savings in energy.

Fig. 8.13 Killing matrix
proceeding from filling in
by rows

	TS$_A$			
	tA$_1$	tA$_2$	tA$_3$	tA$_4$
m1	X			
m2	X			
m3		X		
m4			X	
m5			X	
m6		X		
m7				
m8	X			
m9	X			
m10				

8.5   Understanding the Costs of Test Design

Test design is the task devoted to the design of test cases to fulfil some test requirement that, as already mentioned, commonly corresponds to a coverage criterion.

Testers use coverage criteria mainly for (1) discovering the areas of the system which are not exercised by test cases and (2) building test cases that, now, do go over these unexplored areas. In another sense, the coverage reached by a test suite that does not find any error in the SUT is an indirect measure of the SUT quality: if I completely execute the SUT with the most strict coverage criterion and I do not find any error, then probably the SUT is free of them.

Typically, a testing team receives a system to be tested together with a document explaining its functionalities. Probably after a first contact with a cycle of smoke testing, the team starts to iteratively design test cases for a given functionality. Testing methodologies, such as TMap Next [4], suggest to start test design with test cases having high and early probability of finding errors. In general, when a test case finds a hidden, very difficult to detect fault, that very same test case will probably find many others. Consider, for instance, the full test killing matrix already used in Fig. 8.10, which is reproduced on the left side of Fig. 8.14: there were two faults (m_7 and m_{10}) that remained hidden for the test suite. Suppose now the tester focuses on m_7 and adds a specific test case (tA_5, added on the right side of the figure) for finding it (i.e. to kill that mutant). It is very likely this new test case will also find many other errors: in this artificial example, tA_5 not only finds the fault for which detection has been designed but also m_{10} and others that were also detected by other cases in the test suite.

Considering code metrics and code coverage criteria, testers should first design test cases to reach those statements with a deeper nesting level (Fig. 8.15).

Fig. 8.14 Finding a difficult fault helps to find many others

	TS$_A$			
	tA$_1$	tA$_2$	tA$_3$	tA$_4$
m1	X	X		
m2	X	X		
m3		X		
m4			X	X
m5			X	
m6		X	X	X
m7				
m8	X	X	X	
m9	X			
m10				

	TS$_A$				
	tA$_1$	tA$_2$	tA$_3$	tA$_4$	tA$_5$
m1	X	X			
m2	X	X			X
m3		X			X
m4			X	X	X
m5			X		
m6		X	X	X	X
m7					X
m8	X	X	X		X
m9	X				
m10					X

Fig. 8.15 Reaching the highlighted statement requires reaching all the others

```
if (A) {
    m1();
    if (B) {
        m2();
        if (C) {
            m3();
            while (D) {
                m4();
                m5();
            }
        }
    }
}
```

8.6 A Theoretical Model for Testing Cost and Energy Consumption

Consider a system *S* composed of a set of classes {*A, B, C, D, E*} and a suitable test suite *TS* with *n* test cases: $TS = \{tc_1, tc_2, \ldots tc_n\}$. Suppose that, for each class in *S*, we have a set of mutants $M = \{M_A, M_B, M_C, M_D, M_E\}$. The number of mutants generated for each class is, respectively, *a, b, c, d* and *e*. This is

$$M_A = \left\{ M_A{}^1, M_A{}^2, M_A{}^3, \ldots M_A{}^a \right\}$$
$$M_B = \left\{ M_B{}^1, M_B{}^2, M_B{}^3, \ldots M_B{}^a \right\}$$
$$M_C = \left\{ M_C{}^1, M_C{}^2, M_C{}^3, \ldots M_C{}^a \right\}$$
$$M_D = \left\{ M_D{}^1, M_D{}^2, M_D{}^3, \ldots M_D{}^a \right\}$$
$$M_E = \left\{ M_E{}^1, M_E{}^2, M_E{}^3, \ldots M_E{}^a \right\}$$

Finally, for doing mutation testing with *TS* against *S*, we must generate mutated versions of *S*, each one containing one class mutant. So, the set of mutated versions *V* is

$$V=\{$$
$$\{M_A^1, B, C, D, E\},$$
$$\{M_A^2, B, C, D, E\},$$
...
$$\{M_A^a, B, C, D, E\},$$
...
$$\{A, M_B^1, C, D, E\},$$
...
$$\{A, B, M_C^1, D, E\},$$
...
$$\{A, B, M_C^c, D, E\},$$
...
$$\{A, B, C, D, M_E^1\},$$
...
$$\{A, B, C, D, M_E^e\}$$
$$\}$$

Each class contributes to the number of versions with its corresponding number of mutants. For the system in Table 8.2, composed of three classes with, respectively, 3, 4 and 2 class mutants, the number of mutant versions is $3+4+2=9$.

v_1 (the first mutated version of S) is composed of the first generated mutant of the A class and the original versions of the remaining classes B, C, D and E. The last mutated system version is composed by the original classes A, B, C and D, plus the last mutant generated for $E(M_E^e)$. Since there are a mutants for A, b for B, etc., the total number of mutant versions is $|V|=a+b+c+d+e$.

8.6.1 Testing the First Release

If we want to fill in the whole killing matrix for the first system release, we need to perform a full execution of all the test cases against all the mutant versions. The total cost of this activity is

$$cost(S, V, TS) = |TS| \times |V| = |TS| \times (a+b+c+d+e)$$

Having the full killing matrix, we can get a test suite TSR, proceeding from the reduction of TS. The size of TSR will be equal or lower to that of TS:

Table 8.2 A sample system with three classes A, B, C and, respectively, 3, 4 and 2 mutants

Original system and mutants	System versions, each with a class mutant		
$S=\{A, B, C\}$	$V_1=\{A_1, B, C\}$	$V_4=\{A, B_1, C\}$	$V_8=\{A, B, C_1\}$
A mutants $=\{A_1, A_2, A_3\}$	$V_2=\{A_2, B, C\}$	$V_5=\{A, B_2, C\}$	$V_9=\{A, B, C_2\}$
B mutants $=\{B_1, B_2, B_3, B_4\}$	$V_3=\{A_3, B, C\}$	$V_6=\{A, B_3, C\}$	
C mutants $=\{C_1, C_2\}$		$V_7=\{A, B_4, C\}$	

$$|TSR| = \alpha \times |TS|, 0 \leq \alpha \leq 1$$

The cost of using *TSR* for testing *S* is

$$cost(S, V, TSR) = \alpha \times cost(S, V, TS) = |TSR| \times |V| \leq |TS| \times |V|$$

8.6.2 Testing a Corrected Release

Suppose that users detect bugs in *S* that must be fixed. So, the development team carries out a corrective maintenance intervention and produces S_c, a *corrected* version of *S*. In general, corrective interventions affect just small pieces of the code. For this example, let us assume that *A* is the only class modified. Since the code of *A* has changed, the previous *A* mutants are no longer valid, and new mutants must be generated for this class. We can ignore the cost of mutant generation (it is actually very low compared to others) and put our focus on the costs of test execution. We need only to execute the reduced test suite against *V*. The cost of this execution is

$$cost(Sc, V, TSR) = |TSR| \times |V|$$

8.6.3 Testing a Perfective Release

The addition of a new functionality introduces major changes in the system, likely with new classes and modifications in the previous release. Supposing *E* has changed and *F*, *G* have been added, the *perfective* system is

$$Sp = S - \{E\} \cup \{F, G\} = \{A, B, C, D, E, F, G\}$$

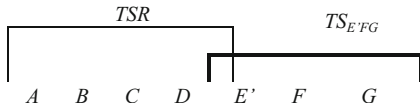So, a new set of mutants must be generated for *E'*, *F* and *G:*

$$M' = M - \{M_E\} \cup \{M_{E'}\} \cup \{M_F, M_G\}$$

Since *M* changes, also does *V*:

$$V' = V - \{v_i \mid v_i \, contains \, M_{E'}^{\,j}\} \cup V_p,$$

where V_p is the set of new versions containing mutant proceedings from the classes added or modified:

Fig. 8.16 *TSR* is valid for *A*–*E'*; $TS_{E'FG}$ is needed for *E'*, *F* and *G*

$$V_p = \left\{ v_i \middle| v_i \text{ contains mutants in } M_E{}^j \text{ or } M_F{}^j \text{ or } M_G{}^j \right\}$$

Consider (see Fig. 8.16) that *TSR* is still valid for *A*, *B*, *C*, *D* and *E'* but that new test cases are required for the changing parts of *E'*, for *F* and for *G*. That is, *TSR'* = *TSR* ∪ $TS_{E'FG}$.

With these considerations, the testing cost is

$$cost(Sp, V', TSR \cup TS_{E\ FG}) = |TSR| \times |V| + |TS_{E\ FG}| \times |V_p|$$

8.6.4 Example

Let *S* be a system with 10 classes and 10,000 class mutants. For simplicity, let us consider that there are 1,000 mutants of each class. Consider also that the initial test suite holds 200 test cases (50 mutants per test case):

$$cost(S, V, TS) = 10{,}000 \times 200 = 2{,}000{,}000 \text{ units}$$

Although it depends on how test cases have been built, a mean test suite reduction rate may be around 12 % of the original test suite. So, the reduced test suite would have $0.12 \times 200 = 24$ test cases. So, the future cost of regression testing (applicable after corrective maintenance interventions) will be

$$cost(S, V, TSR) = 10{,}000 \times 24 = 240{,}000 \text{ units}$$

Suppose the system is extended with new functionalities: one existing class *X* is modified to *X'* and three are added. The number of new versions is

$$|V'| = |V| - |M_X| + |M_X| + (3 \times 1000) = 10{,}000 - 1{,}000 + 1{,}000 + 3{,}000$$
$$= 13{,}000$$

For killing the new 3,000 versions, we require $3{,}000/50 = 60$ new test cases (note that we have no reduction at this moment for the new three classes). The total testing cost is

$$cost = 240{,}000 + 60 \times 3{,}000 = 420{,}000 \text{ units}$$

After this, a new reduction can be made that will only affect the new 60 test cases. Assuming also a reduction rate of 0.12, this results in $60 \times 0.12 = 7$ test

cases. The new whole reduced test suite will have the 24 previous test cases plus these 7. So, the execution cost of future corrective releases will be

$$cost = 240,000 + 7 \times 2000 = 254,000\, units$$

8.7 Conclusions

From the point of view of energy consumption, test case execution is the most expensive task: mutant generation (if the mutation score is used as the coverage criterion) and test suite reduction are almost insignificant, as well as test case generation (since test cases can be generated with automated tools). Test case execution, however, implies CPUs working hard for significant amounts of time. So, the application of test suite reduction algorithms and the introduction of other control mechanisms (such as controlling what test cases fulfil every test requirement, e.g. controlling what test cases cover each mutant cluster) may lead to important cost and energy savings. Although cost and energy savings may be small for a single project, data centre managers may realise the benefits of applying smart policies of test case management in their project portfolios.

References

1. DeMillo RA et al (1978) Hints on test data selection: help for the practicing programmer. Computer 11(4):34–41
2. Harrold MJ et al (1990) A methodology for controlling the size of a test suite. In: Conference on software maintenance, 1990, Proceedings, pp 302–310
3. Jeffrey D, Gupta R (2005) Test suite reduction with selective redundancy. In: Proceedings of the 21st IEEE international conference on software maintenance, 2005. ICSM'05, pp 549–558
4. Koomen T (2006) TMap Next for result-driven testing. UTN, 's-Hertogenbosch
5. Myers GJ (2004) The art of software testing, 2nd edn. Wiley, Hoboken, NJ
6. Offutt AJ et al (1996) An experimental determination of sufficient mutant operators. ACM Trans Software Eng Methodol 5(2):99–118
7. Offutt AJ et al (1996) An experimental evaluation of data flow and mutation testing. Softw Pract Experience 26(2):265–176
8. Offutt AJ, Lee SD (1994) An empirical evaluation of weak mutation. IEEE Trans Software Eng 20:337–344
9. Pargas RP et al (1999) Test-data generation using genetic algorithms. Software Test Verification Reliab 9(4):263–282
10. Polo M et al (2009) Decreasing the cost of mutation testing with second-order mutants. Software Test Verification Reliab 19(2):111–131
11. Polo M et al (2012) Reduction of test suites using mutation. In: Proceedings of the 15th international conference on fundamental approaches to software engineering. Springer, Berlin, pp 425–438
12. Reales P, Polo M (2012) Bacterio: Java mutation testing tool: a framework to evaluate quality of tests cases. In: 28th IEEE international conference on software maintenance (ICSM), pp 646–649
13. Tallam S, Gupta N (2005) A concept analysis inspired greedy algorithm for test suite minimization. In: Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on program analysis for software tools and engineering. ACM, New York, pp 35–42

Chapter 9
Green Software Maintenance

Ignacio García-Rodríguez de Guzmán, Mario Piattini, and Ricardo Pérez-Castillo

9.1 Introduction

The most important development stage (requirements, analysis, design, etc.) for each software practitioner is probably the one in which the practitioner is involved. There is one irrefutable fact, however: the software maintenance stage is the most time- and cost-consuming (up to 80 %). The time spent in software development is usually within the boundaries imposed by project planning, but the maintenance stage is not limited, and it will be running along with the software until its removal.

According to [8], software maintenance can be defined "as the totality of activities required to provide cost-effective support to software"; to a great extent, software maintenance will be more or less cost- and time-consuming depending on the quality of the product when it was delivered, as well as on the software and hardware updates, new functionalities required by the stakeholders, etc.

Software maintenance is fully described in the ISO/IEC 14764 standard [11] as a software process that "... begins with Process Implementation where planning for maintenance is performed and ends with the retirement of the software product. It includes modification of code and documentation due to a problem or need for

I. García-Rodríguez de Guzmán (✉)
Institute of Information Technologies and Systems, University of Castilla-La Mancha, Ciudad Real, Spain
e-mail: Ignacio.GRodriguez@uclm.es

M. Piattini
Department of Information Technologies and Systems, University of Castilla-La Mancha, Ciudad Real, Spain
e-mail: Mario.Piattini@uclm.es

R. Pérez-Castillo
Itestra GmbH, Madrid, Spain
e-mail: perez@itestra.com

improvement. The objective of the Maintenance Process is to modify an existing software product while preserving its integrity. . ." This standard [11] also identifies four types of software maintenance:

- *Corrective maintenance*: Modifications made necessary by actual errors in a software product. If the software product does not meet its requirements, corrective maintenance is performed.
- *Preventive maintenance*: Modifications made necessary by detecting potential errors in a software product.
- *Adaptive maintenance*: Modifications to implement new system interface requirements, new system requirements or new hardware requirements.
- *Perfective maintenance*: Providing new functionality improvements for users; it might also be reverse engineering either to create maintenance documentation that did not exist previously or to modify existing documentation.

Software maintenance could be carried out with more or less effort depending on the relative ease with which the system can be maintained, that is, on the maintainability quality attribute. According to ISO 25000 [12], maintainability is one of the six quality characteristics of a software product. Maintainability, in turn, is built up by means of five sub-characteristics (modularity, reusability, analyzability, modifiability, and testability). Many efforts in this stage will be focused on the improvement of this quality characteristic.

Despite the fact that software maintenance is a very well-known process which has a lot of research and knowledge at its disposal, new challenges that must be addressed in the maintenance phase are arising today. People are becoming sensitive to the current environmental problems that IT is helping to cause due to its direct impact. There are three main degrees of impact on the environment that IT causes [21]: first-order impacts are environmental effects that result from production and use of ICT, i.e. resource use and pollution from mining, hardware production, power consumption during usage, and disposal of electronic equipment waste; second-order impacts are effects that result indirectly from using ICT, like energy and resource conservation by process optimisation (dematerialisation effects), or resource conservation by substitution of material products with their immaterial counterparts (substitution effects); and third-order impacts are long-term indirect effects on the environment that result from ICT usage, like changing life styles that promote faster economic growth and, at worst, outweigh the formerly achieved savings (rebound effects).

Interest in designing and implementing Green IT solutions has therefore been increasing dramatically over the past few years [19]. A main driving force behind this green movement is IT's rapidly rising energy demands, due to the growing global adoption of computing services [20].

New models and proposals are appearing in the quest to produce sustainable software [21], but there is an issue here that cannot be ignored: the huge amount of complex information systems running nowadays in all organizations. Most of them have probably been developed following a *classical* approach (i.e., not a *green software engineering* [21]). As a consequence, these software systems would not be

sensitive about any kind of impact on the environment and are thus not considered as Green IT.

So the question is: "Is there any chance of all these systems becoming more environment-friendly?" Maintenance arises as a possible answer to this question. Classical maintenance activities allow the improvement of software quality characteristics [12]; maintenance, in turn, would also be helpful in improving greenability for those legacy systems that have not been developed following green approaches.

The maintenance process deals with legacy systems by means of *software refactoring*. A refactoring is a correctness-preserving transformation that improves the quality of the software without altering the semantics [17] or functionality. There is not much research about *what to do with non-green legacy systems*, but the maintenance process would be the best candidate for dealing with non-green software systems in an effort to improve their degree of greenability.

This chapter intends to outline a definition for green maintenance and discusses possible ways of undertaking the improvement of software greenability in the maintenance process. The state of the art is low, which means that it is essential to establish mechanisms to deal with existing software systems that have not been developed under green principles. The first step outlined in this work therefore consists of exploring the effects of classical maintenance issues (refactoring software flaws) on greenability. We propose not to consider green maintenance as a separate kind of maintenance but rather to integrate it with the classical type. Their aim is to find out how to combine both sorts of maintenance in such a way as to avoid side effects when applying refactoring to improve greenability or another software quality [12].

In addition, as technical debt [4, 16] influences classic software maintenance, a new concept is proposed: ecological debt. Ecological debt is intended to measure the cost of not undertaking green software maintenance to improve the greenability quality characteristic of a software system. The relationship between ecological and technical debts is analyzed, along with their relationship with classical and green software maintenance.

The remainder of this chapter is organized as follows: Sect. 9.2 analyzes classical and green maintenance, the possible relationship between them, and how they should be considered together; Sect. 9.3 discusses some of the typical software flaws that are solved in classical maintenance and how these could also be considered for green maintenance; Sect. 9.4 discusses software debt as a concept that is closely related to software maintenance and outlines how a new concept, *ecological debt*, should be considered as a possible way of measuring the greenability improvements that remain to be carried out in software; Sect. 9.5 presents a case study about how refactoring bad smells would have a direct impact on software greenability; finally, Sect. 9.6 draws a few conclusions summarizing the ideas presented in this chapter.

9.2   Greening Software Maintenance: A New Facet for a Classical Process

9.2.1   Existing Proposals

Software maintenance only focuses on preserving software functionality (or including additional functionalities when needed) through the analysis, modification, and improvement of either system source code or its associated documentation. In this definition, there is no evidence either about software greenability or of any of its areas of impact. This being so, an important question arises: "Is there any sense in stating the basis for green and sustainable software engineering when there is such a vast amount of software systems running that have not met their ROI?"

The answer should be *yes*, because in spite of the fact that most software systems have not been developed according to the basic principles of *green and sustainable engineering,* we cannot ignore the numbers: "the approximate energy consumption of U.S. data centers increased from 28 billion kWh in 2000 to 61 billion kWh in 2006. Meanwhile, estimations rose from 58 billion kWh in 2000 to 123 billion kWh in 2005 on a global scale" [21]. The current software systems must be "maintained" to obtain new versions with a higher degree of greenability; unfortunately, traditional software maintenance does not deal with this kind of quality improvement.

Despite green software maintenance being a fast-growing research area, the concept of "green software maintenance" or "sustainable maintenance" is mentioned in only a few papers [21, 24]. There is a lack of work that can serve as a reference point, and the few papers that do exist offer a description of "green maintenance" that is very far from the basis of "classical maintenance".

In [21], the authors present the *GREENSOFT* model, where green and sustainable software is described, along with the corresponding engineering process. Focusing on the software development stage of the subject of this chapter, the *GREENSOFT* model considers a *usage phase*, where impact that comes about as a result of deploying, using, and maintaining the software product is considered. As the proposal states, different activities are carried out in this stage: installation of software patches or updates, configuration of software systems, training of employees in regard to proper software usage (well-trained employees carry out their tasks faster, which entails lower power consumption), configuration of the software system to consume less power or just switching their computer to suspend mode when they leave. Other energy- and resource-consuming issues, such as the connection to services provided by other servers or the replacement of old hardware (because the updating of the software systems installed requires more powerful equipment), are considered in the usage (maintenance) phase of this proposal.

The *green software development model* [24] proposes a new approach for the software development life cycle. This proposal focuses on what should be considered in each phase of the development process (requirements gathering, design, implementation, testing, deployment, and maintenance) to produce software in an environment-friendly way. This approach considers some of the classical

maintenance issues such as fixing bugs, fine-tuning the system, implementation of new features, etc. From the green point of view, the authors propose (1) using electronic documentation instead of paper-based documentation and (2) not using reverse engineering or any technique based on disassembling the software system, since these are time- and power-consuming activities. To reduce the impact of the previous point, we need to try to (3) involve the development team in maintenance. Maintenance thus speeds up, since the maintenance team becomes familiar with the system to be maintained; the authors also propose (4) trying to avoid software system migration, since it leads to disposal of legacy hardware and devices, along with disposal of old technology, and (5) building the system in such a way as to be adaptable to new technologies, devices, and hardware equipment.

Having reviewed the scarce documentation about the topic, we see that the initial ideal of *green software maintenance* seems to be slightly distorted. The ideas presented in this section do not improve software *greenability* or reduce power consumption. The aforementioned proposals are good practices that make for a more environment-friendly maintenance stage (reducing paper consumption, training users to carry out their tasks more quickly, also using fewer resources, avoiding refactoring and reengineering, etc.). Do these practices actually improve software *greenability*, however? The answer is "no". The fact is that reducing power consumption is not directly affected by any of these useful approaches: "greening software maintenance" is not the same as "maintenance to make software greener." The second idea fits better with the maintenance concept set forth in the ISO/IEC 14764 standard [10], where maintenance is the process intended to improve the quality of a software product. Considering *greenability* as a software quality characteristic for measuring the degree to which a software product has appropriate power consumption, software maintenance is the only way to reach such improvement through software transformation. Keeping such an idea in mind, we therefore speak about *green software maintenance* or we refer to those tasks undertaken during software maintenance to improve *greenability* in an effort to obtain new versions of a given system while at the same time improving power consumption.

9.2.2 Green Software Maintenance: A First Approach

The approaches mentioned above do not lack grounds, but it is important to look at maintenance from all points of view if we are to provide a proper definition of what exactly *green software maintenance* is.

Software configuration and optimization, user training, and software management are the main activities in the software maintenance (and deployment) stage. Nonetheless, from a software engineering point of view, software evolution is also an inevitable activity (according to Lehman's laws [15]), which occurs alongside software maintenance.

This assumption leads us to the fact that software maintenance implies software evolution, so suitable tools and techniques must be provided. To enable such an evolution, the system can be *modified*, *improved*, and *manipulated* and be the subject of any kind of actions that imply dealing with the source code and the associated documentation. In the best circumstances, software evolution is carried out by accessing the module or class that must be improved and updating the corresponding documentation. However, in the real world, most systems are not in such a situation. In general, when systems have not been developed in the recent past, source code and documentation (if that documentation exists) are not aligned, and there is no evidence about *what process is being executed by this software* [22]. Such misalignment is often produced by the natural ageing of software systems [26]. Classical software maintenance is therefore in charge of dealing with all these situations, attempting to improve the software quality of existing software systems.

At this point, if the classical software maintenance process deals with a given kind of software issues that arise during the operational life of the software system, *what should we consider as green software maintenance*?

When a system has been developed under the principles of *green software engineering* [21], green software maintenance makes sense in the way that it preserves functionality as well as the *greenability characteristic* of the software system. That said, we ought to consider what is going on with the vast amount of existing software systems which have not been developed following greenability principles. In this case, green maintenance (as a facet of the software maintenance stage) must undertake all the required changes to improve the greenability quality characteristic of the system. In this sense, the green software maintenance carried out on this kind of systems would address, for example, the improvement of power consumption.

So, since the classical software maintenance perspective is well defined by means of the existing standard, the green software maintenance perspective should now be established. The following definition is inspired by the one proposed in [11]: "green maintenance is performed during all the software working cycle and ends with the retirement of the software product, undertaking at this point all the required activities to reduce the environmental impact of the retired software. It includes modification of code and documentation in order to solve possible deviation of the greenability requirements (or the implementation of a new ones), without modifying the original functionality of the source code." This definition takes into account that green maintenance must (1) be carried out throughout the whole software life cycle, (2) preserve the original functionality of the software system, (3) preserve the sustainable requirements of the system, and (4) include the new ones (e.g., for systems which had not initially been developed following the principles of green and sustainable software engineering).

It is now possible to consider a new maintenance process in which classical maintenance issues, as well as greenability requirements, are taken into account (see Fig. 9.1).
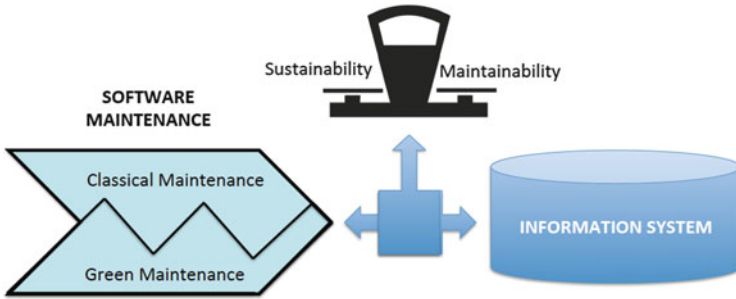
Fig. 9.1 Unified concept of software maintenance and the crossroad between quality characteristics

Unfortunately, the problem is not as easy as considering a global process with two subsets of activities, each one dealing with a different concern. As far as we know, there are no studies dealing with this precise problem, but it is possible to find some research work that addresses the issue of the relationship between green requirements and software maintainability. In [23], the authors analyze the consequences of applying refactoring to solve the god-class design anti-pattern. Although this case study is discussed in depth in the following sections, the most important conclusion is the fact that improving the software maintainability attribute [12] may worsen the greenability quality characteristic of a system and vice versa (see Fig. 9.1).

In spite of the fact that classical maintenance and green software maintenance should be considered as a whole process, there does in fact exist a crossroad where the most suitable balance between maintainability and greenability can be decided on.

At the maintenance stage, the most important quality characteristic is maintainability (*degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers* [12]). Maintainability states the degree of straightforwardness in carrying out maintenance. Maintenance is a high-level quality characteristic, and other sub-characteristics [12] to measure the different facets of maintainability are thus defined in the standard: modularity, reusability, analysability, modifiability, and testability. Good values for these sub-characteristics make maintenance easier and cheaper.

On the other hand, *greenability* can be considered as the degree of environmental friendliness of a software system, based on its power consumption. A high value for this characteristic implies low power consumption. As occurs in software measurement, the most important problem when measuring a quality characteristic has to do with thresholds. What is considered a good value? Are values of greenability valid for any kind of software, regardless of its size or working domain? While a lot of research about thresholds in software measurement has been carried out, greenability lacks these reference values.

Table 9.1 Relationship between maintainability and greenability

		Greenability	
		Relation	Implication
Maintainability	Modularity	X	– More modules imply more communication lines – Better design implies less energy and time to carry out any other task of maintenance
	Reusability	X	– Highly reusable assets are prone to be optimized, as is their greenability
	Analyzability	?	
	Modifiability	X	– If an asset is easy to modify, it is likely to keep (and not worsen) its greenability
	Testability	?	

"X" for a possible relationship, "?" when the relationship is not clear

The crossroad between classical maintenance and green software maintenance mentioned above can be transferred to a trade-off problem between maintainability and greenability. Improving a sub-characteristic of maintainability can affect greenability positively, negatively [23] or not at all. On the other hand, if the greenability of a software system is improved during maintenance, maintainability may worsen. It is essential to conduct experiments in order to figure out what kind of implication the improvement of a sub-characteristic of maintainability has on greenability. At the moment, it is only possible to predict an implication between maintainability and greenability (see Table 9.1).

Table 9.1 presents the hypothetical relationship between greenability and maintainability. Since maintainability is a coarse-grained concept, the relationship is established between greenability and the sub-characteristics of maintainability. For each sub-characteristic, whether or not there exists a possible relationship with greenability is considered, but it is not clear to what extent the relationship is positive or negative (i.e., if by increasing one, the second is positively affected or the opposite). For example, the modularity sub-characteristic ("degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components" [12]) is likely to be related to greenability for two reasons:

- Modularization implies more items to build up the system and thus more elements to intercommunicate. This fact implies a greater amount of power consumption, since a higher number of messages require more energy.
- On the other hand, a suitable degree of modularization means a better design and, thus, a quicker (and less energy-consuming) maintenance.

Another sub-characteristic that probably has a positive impact on greenability is reusability ("degree to which an asset can be used in more than one system, or in building other assets" [12]). The more a component is used, the more optimized the component will be. If we consider the optimization level in terms of greenability, this component would probably be optimized to have low power consumption. In addition, the component would have a good degree of maintainability.

In any case, Table 9.1 displays the hypothesis of possible relationships between maintainability and greenability. Experiments and case studies must be carried out to establish a validated correspondence between maintainability and greenability that would in turn be useful for understanding the correlation between classical maintenance and green software maintenance. This knowledge must exist before we can theorize about an integrated view of software maintenance that comprises classical and green views of this stage of the software life cycle.

9.3   Promising Techniques for Improving Greenability in Green Software Maintenance

From a software engineering point of view, we concluded in the previous section that green software maintenance has two main objectives: to preserve the greenability quality degree and to improve greenability when it has never been taken into account for a given system.

For our purpose, the most important challenge is the second one: *what could be done in the maintenance process to improve greenability without changing the original functionality and decreasing maintainability?*

At the moment, it is only proposals such as [21, 24] that state what could be done during the maintenance and usage stage to improve/keep software greenability. Nonetheless, these approaches do not deal with the design and the source code of software systems. The literature does not reveal any kind of source code/design modification that should be performed during maintenance in order to improve greenability. That means that it is possible that existing techniques for improving software maintainability could also be useful for improving software greenability.

Working with this assumption, it is important to figure out (1) how we can deal with source code in order to improve it without modifying its functionality and (2) what structures/problems should be detected in the effort to improve maintainability and (perhaps) greenability.

The first question has a very easy answer. In maintenance time, *refactoring* is the process for improving problems of the source code [6, 17], so system quality is also improved. Refactoring is one of the main steps in software reengineering (see the next subsection for a detailed explanation). The second question refers to all the things that deteriorate the source code (and in turn the software system). In maintenance, the system can be refactored to repair bugs, solve bad programming and design practices (bad smells and anti-patterns).

The following subsections offer an overview of what refactoring, bad smells, and anti-patterns are and why they are candidates for improving software greenability.
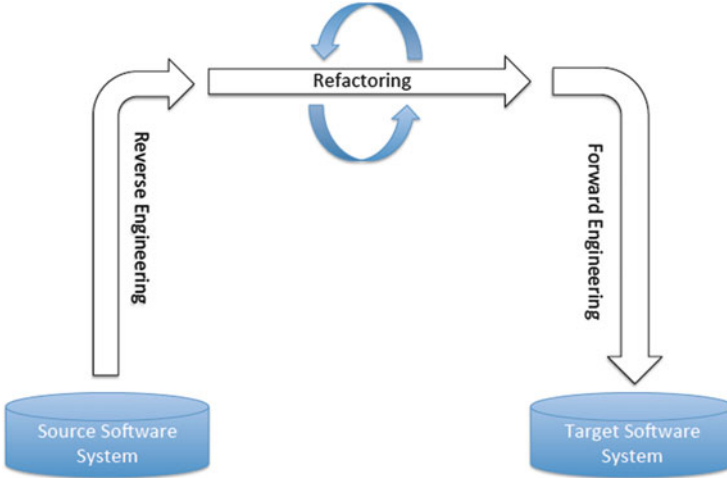
Fig. 9.2 Reengineering horseshoe model

9.3.1 Software "Greengineering": Reengineering for Greening Legacy Systems

Software reengineering [1] is a classical tool for dealing with existing software systems. The classical model of reengineering establishes three subprocesses [16]: reverse engineering, restructuration, and forward engineering. This model is also known as the *horseshoe model* [17] (see Fig. 9.2).

The reverse engineering stage allows the processing of the software system in order to create abstract representations of its structure. Once the system is abstracted, the improvement of the software system begins by means of the *software refactoring* stage. As [17] discuss, software refactoring is an updating of the concept of *software restructuration* [3] of the reengineering process and could be defined as "the process of changing a [an object-oriented] software system in such a way that it does not alter the external behavior of the code, yet improves the internal structure." In the third stage, forward engineering, the abstract representation that has been improved/modified is used to regenerate the source code implementing the system. The target is functionally equivalent to the original one but with a better quality level.

The role of software refactoring is clear in classical software maintenance, but what does this mean for a hypothetical definition of *green software maintenance*? It is important to consider that classical and green maintenance should not be separate concerns but complementary (see Fig. 9.1). The adoption of a green perspective of software development should not put software engineering to one side but rather the opposite. Focusing on the concern of this chapter, we should address the integration of "classical with green software maintenance." instead of "classical versus green software maintenance."

Software refactoring is consequently the best choice for dealing with green software maintenance, because refactoring is the only way of dealing with code if we are to provide the software system with green capabilities. The key point now is to figure out what should be refactored and what kinds of improvements reduce power consumption. Applying refactoring to provide a system with green capabilities can be understood as green software reengineering or *software greengineering*.

9.3.2 Bad Smells

Bad smells could be understood as unsightly programming styles and poor design strategies that appear in source code as a consequence of quick and uncontrolled development. In [6], the author brings together a set of code smells which frequently appear in software systems. Table 9.1 summarizes the set of code smells presented in [6], together with a brief definition. The author also considers a set of possible refactoring approaches for each bad smell, but this is beyond the scope of this chapter.

In classical maintenance, software refactoring is carried out to solve the quality problem that underlies bad smells. As the author proposes, for each bad smell, it is possible to apply a refactoring solution to transform the current state of the system into an equivalent one with a better quality and without the bad smell.

The issue is this: in classical maintenance, such transformations are desirable and required, but it is not clear to what extent there exists any kind of correlation between classical and green maintenance that would ensure an improvement of the system greenability. In this sense, experimentation is required in order to lay down which bad smells should be removed and which refactoring solutions should be applied. The crossroad mentioned in Sect. 9.5.3 establishes that there are some maintainability improvements which are inversely proportional to greenability improvement.

In addition to the bad smell and the definition, Table 9.2 includes a column called *impact*. Impact makes reference to the effect on greenability of refactoring a given bad smell. As we said before, there is no research about the effect of refactoring on greenability, so the values for the *impact* column have been hypothesized (i.e., experimentation is required in order to validate it). Three values have been designated to predict the impact of a refactoring: (1) "+," when the refactoring may have a positive effect, improving greenability; (2) "0," when it is not clear when the refactoring has a positive or negative effect on greenability; and "−," when the refactoring worsens greenability.

Table 9.2 Code of bad smells and possible impact on greenability

Bad smell	Definition	Impact
Duplicated code	A fragment of code is repeated one or more times	−
Long method	Methods with too many variables, parameters, and code	0
Large class	A class with too many instance variables. Too many responsibilities	−
Long parameter list	Method with a long and understandable list of parameters	0
Divergent change	Code is similar but not the same	+
Shotgun surgery	For a given change, a lot of changes in other places are required	+
Feature envy	A method in a class is more interested in another class than in the owner class. The method *envies* the data of the other object	+
Data clumps	The same set of data is repeated in different places (classes, methods, parameters, code, etc.)	−
Primitive obsession	Using of primitive types instead of objects, reducing understandability	+
Switch statements	The same switch statement is scattered in different places	+
Parallel inheritance hierarchies	When you make a subclass of a class, you must also make a subclass of another	+
Lazy class	A class that does not do anything except cost money to maintain and understand	+
Speculative generality	Too many sorts, hooks, and special cases are included in code for an uncertain need in the future	+
Temporary field	An instance variable in a class is not always instantiated. It makes it difficult for there to be understandability	0
Message chains	The client is coupled to a long and complex structure of navigation when requesting one object	+
Middle man	A class is delegating much of its behavior to a second class	+
Inappropriate intimacy	Two classes are tightly coupled	+
Alternative classes with different interfaces	Two methods doing the same, with different signatures. Two classes doing something very similar	0
Incomplete library class	Libraries do not usually contain all the functionality we need	0
Data class	Classes without responsibility and only fields with get/set methods	+
Refused bequest	A class does not need all the fields and methods from a parent class	+

9.3.3 Anti-patterns

According to [2], anti-patterns are "a literary form that describes a commonly occurring solution to a problem that generates decidedly negative consequences. The Anti-Pattern may be the result of a manager or developer not knowing any better, not having sufficient knowledge or experience in solving a particular type of

problem, or having applied a perfectly good pattern in the wrong context." Such anti-patterns have a negative impact on software quality and might also worsen software maintainability.

Anti-patterns could also be a possible path for dealing with software greenability. Finding and solving software anti-patterns improve software quality, but the question now is to find out to what extent software refactoring (to eliminate anti-patterns) also affects greenability.

In [2], the authors face patterns from three different perspectives: (1) software development anti-patterns or technical problems and solutions that are introduced by programmers, (2) architectural anti-patterns or common problems in how systems are structured, and (3) managerial anti-patterns or problems in software processes and development organizations. From the software engineering point of view, the most interesting kind of software anti-patterns is the first one, development anti-patterns, because they represent problems that should be addressed in software maintenance. Architectural anti-patterns could also be considered to be of interest because most of the consequences of these anti-patterns could be treated in source code (at least relieving their symptoms). On the other hand, the managerial anti-pattern is not useful if we are addressing greenability from a software engineering point of view. Managerial anti-patterns identify problems related to the human resource involved in software projects. Although certain managerial anti-patterns (those relating to human working manners) can be directly related to software greenability, they are not the focus of this chapter. Table 9.3 summarizes the most common development anti-patterns.

Table 9.4 presents the most common architecture anti-patterns. According to [18], architecture anti-patterns "focus on some common problems and mistakes in the creation, implementation, and management of architecture."

Tables 9.3 and 9.4 include an *impact* column to represent the possible impact of applying software refactoring to solve the anti-patterns. In this case, there are four possible values: "+" if the refactoring has a positive impact on greenability, "−" if the refactoring has a negative impact on greenability, "0" if it is not clear whether there is an impact on greenability, and "NA" when there is no software refactoring to solve the anti-pattern (i.e., it is more related to the process than to the product).

As we pointed out with code smells, the prediction of the impact of refactoring the anti-patterns is just a hypothesis. It is at this point that experiments with each anti-pattern (and with combinations of them) must be conducted in order to find out which of them would be suitable for improving software greenability at a maintenance stage (i.e., green maintenance).

9.3.4   Considerations

Despite the fact that we have analyzed the possible impact of bad smells and anti-patterns on software greenability separately, there is an important relationship between both concepts that must be taken into account. On one hand, anti-patterns

Table 9.3 Software development anti-patterns

Anti-pattern	Definition	Impact
The blob (*god class*)	One class contains most responsibilities, while the others hold only data and small processing	−
Continuous obsolescence	Technology evolution makes it difficult for developers to keep software interoperating properly with other products	0
Lava flow	Dead code and forgotten design is frozen in an ever-changing design	0
Functional decomposition	OO systems produced by non-object-oriented developers. The object-oriented code resembles structural language	+
Poltergeist	Very short-time life cycle classes. Such classes are usually responsible for starting processes	+
Boat anchor	Software or hardware artifact that is very costly but without any useful purpose	+
Golden hammer	Development teams apply solutions in which they are very experienced again and again, instead of exploring suitable new ones	0
Dead end	A reusable component supported by a vendor or supplier is modified. This complicates integrating such modifications in new releases	0
Spaghetti code	Software structure is made in an ad hoc way, so it is difficult to extend and maintain	−
Input kludge	Ad hoc algorithms manage program input	+
Walking through a minefield	Products are released too early, and an important number of bugs are (very probably) in the code	+
Cut-and-paste programming	Copy-and-pasted blocks of source code entail maintenance problems	−
Mushroom management	Developers are isolated from the system's end user. Requisites are received indirectly, by means of other intermediaries	0

are common (but undesirable) design decisions or poor designs taken by bad designers or constraints in the project. On the other hand, bad smells are a symptom that something is not working well. In other words, bad smells are the evidences of a possible anti-pattern. For example, if we consider *the blob* (also known as *the god-class* anti-pattern), there are several bad smells that might demonstrate the existence of an anti-pattern: the *long method*, the *large class*, and the *data class*. None of these three bad smells in a program is proof in itself of the existence of *the blob* anti-pattern, but finding any of them is a reason to suspect that this anti-pattern does indeed exist.

Table 9.4 Software architecture anti-patterns

Anti-pattern	Definition	Impact
Autogenerated stovepipe	A local system is migrated to a distributed architecture. If the design remains the same, problems appear, such as how data is transferred	+
Jumble	Vertical and horizontal elements are mixed. Software is complex and difficult to evolve and reuse	−
Stovepipe system	There is neither abstraction nor documentation of subsystems. Their integration has to be done in an ad hoc manner	+
Cover your assets	Requirements are spread over "tons" of documents. Developers have no idea about what to do with such a mess of information	NA
Vendor lock-in	A product adopts a given technology and becomes dependent upon the vendor conditions. Problems arise when the product upgrades	+
Wolf ticket	A product that meets software standards but whose interfaces may vary from the published standards	0
Architecture by implication	Overconfident architects believe that certain important architectural documentation is not needed. They have a lot of experience and then consider that something is not necessary, since it remains in their mind. Development is not possible without such information in document form	0
Warm bodies	Many programmers are assigned to a project, but only a few are quality developers	NA
Design by committee	A complex software design is usually developed by a committee of experts. Different and democratic opinions lead to complex designs that are difficult to implement	NA
Swiss army knife	Excessively complex class interface. The class attempts to serve too many uses	−
Reinvent the wheel	Lack of technological transfer between a project and a new one. The advantage of having design knowledge is made the most of	0
The Grand Old Duke of York	People's talent is not taken into account when defining system architecture. People's skills are important for deciding in which software development stage a particular person involved in the project should work	0

9.4 Technical Debt and Ecological Debt

Technical debt can be defined as "writing immature or not quite right code in order to ship a new product to market faster"; we can find it in many forms (process, scope, testing, and design) [4, 16]. Technical debt could also be understood as the "invisible result of past decisions about software that affect the future" [14]. Equation (9.1) presents a very simple equation that summarizes how technical debt could be calculated. In this formula, the *technological flaw* concept represents any kind of bad smell, anti-pattern or lack of documentation or of test cases, for example:

$$Technical\_Debt = \sum Refactor\left(Technological\_Flaw_j\right) \qquad (9.1)$$

Technical debt may be considered to be a result of decisions to trade off competing concerns during development, but the problem is that these decisions are the result of short-term thinking due to (1) pressures of the project, (2) time constraints, (3) deadlines in the contract, (4) meeting deadlines to integrate with a partner product near the release date, (5) taking advantage of good marketing opportunities (in this case, it could be seen as a matter of investment), (6) development of a prototype, and so on [16].

So, to what extent is technical debt good or bad? On the one hand, the existence of technical debt in a software system presupposes things that are not carried out or provided (documentation, tests, functionalities, etc.) or things that are definitively wrong (misaligned documentation, poor design, bad smells, etc.). On the other hand, assuming a certain degree of technical debt gives us the possibility of releasing a product to take full advantage of a market opportunity, test a first version of a product with a stakeholder who has no clear idea of the requirements or put off implementing requirements that are not essential for a first version of the system. Technical debt is therefore a trade-off problem that offers the chance of transforming debt into investment if and when debt is recognized and quantified (i.e., the technical debt may very well be worthwhile).

Technical debt is a concept that is very close to software maintenance, since any bug, bad smell or anti-pattern that has been intentionally (or unintentionally) introduced into the system must be addressed in the maintenance process. Quantifying technical debt is thus a good method for envisaging how costly maintenance will be in the near-midterm future.

While technical debt is in fact the lack of required functional or nonfunctional requirements (on purpose or unintentionally), it is possible to outline a similar situation with respect to green software development: the concept of *ecological debt*. Greenability requirements (as nonfunctional requirement) would not be an absolute value (i.e., the response time for queries) but would be allowed to move within a given range. Obviously, the greener the system is, the less its consumption of resource. But once again, the trade-off issues should be analyzed. If the long-term cost of increasing the greenness in a software system is less than planned (but with a given and acceptable degree of greenability) and is smaller than making a system highly sustainable, then it may be reasonable to include a certain value of *ecological debt* in the system. It is important to track this debt, since some legal stipulation or stakeholder requirement might very well change. If that should happen, the systems would have to be refactored, with the subsequent waste of resources, as well as the inevitable extra costs. That means (1) technical debt of the code and documentation that must be refactored and the (2) excessive (but assumed) resource consumption up until the time that the change in regulations or requirements occurred. When *ecological debt* is incurred, it is very important to know that different publications advise us to avoid reengineering (and in turn

refactoring) source code, because this is considered a very time- and resource-consuming task.

Nevertheless, not all *ecological debt* is due to decisions taken because of greenability requirements (or failing to complete requirements). Other kinds of *ecological debt* are almost unavoidable and even necessary; for example, the update strategy of a software product influences many factors of that product. These include data transfer, processing, and hardware infrastructure, all of which are required for the delivery of updates. All these issues, which may cause further consumption of power and resource [21], imply a decrease in software greenability. If we submit our systems to such policies or strategies, we are assuming an *ecological debt* that must be recognized and quantified.

Now that the concept is clear, a possible definition can be outlined. Ecological debt may be considered as "the cost (in terms of resource usage) of delivering a software system with a greenability degree under the level of the nonfunctional requirements established by stakeholders, plus the incurring cost required to refactor the system in the future" [Eq. (9.2)]:

$$\text{Ecological\_Debt} = \sum \text{Cost(resourcei)} + \sum \text{Refactor(Ecological\_Flawj)} \quad (9.2)$$

According to Eqs. (9.1) and (9.2), ecological and technological debts have a common factor, that is, the need to fix flaws. Up until now, it has not been clear whether there are specific flaws that are associated with low values of software greenability or even if the existing ones (applied in classical software maintenance) also influence greenability. As proposed in the previous section, a possible starting point for classifying which flaws affect greenability would be the examination of those flaws related to classical maintenance, validating their impact on software greenability.

Equation (9.2) points to a very important consideration when talking about ecological debt; this is the fixed (and unrecoverable) cost of the overused resources. An overused resource is seen as a software or hardware resource which has been oversized for the actual software need. For example, a very common oversized resource related to greenability is power consumption. Power consumption can be expressed as cost ($, €, £, etc.), carbon footprint (CO_2) or electrical power (w/h). The cost of these resources is a sort of investment that cannot be recovered or repaired (as can be done, on the other hand, with technical debt). It is possible to match such wasted power consumption to an economic concept—the irrecoverable expense: it is an outlay that cannot be recovered; this outlay must not influence the future decisions of the organization, since it cannot be recovered. It is nevertheless important to point out that in ecological debt, this irrecoverable expense must be taken into account in the context of the maintenance of the software maintenance strategy for the software portfolio. Technical debt (as well as the refactoring of the ecological flaws of the ecological debt) exists for the whole period during which the organization does not decide to solve it, but the irrecoverable expense factor of the ecological debt [Eq. (9.2)] is a continuous expense that will never be recovered.

This is the most important reason for reducing the level of ecological debt of a software system to the minimum. This is also the motivation which makes us want to define and formalize the green software maintenance presented in this chapter.

Earlier in this chapter, we proposed that green maintenance and classic maintenance should be considered under the same process; the same should be done with technical and ecological debt. Both forms of debt—the ecological and the technological—have in common the need to fix flaws. We can in fact state that the software system has a set of flaws which are responsible for its debt and that that set is made up of the union of the ecological and technical flaws. The issue now is: what happens if a technical flaw affects greenability, or vice versa?

For any given system, if there were not a set of (technical and ecological) flaws that clearly affects the system maintainability and greenability positively and in the same way, it would be necessary to undertake a trade-off analysis, since refactoring technological flaws would negatively affect ecological debt, and vice versa. On the other hand, if there were common flaws, it would be possible to apply a set of refactoring transformations that benefits both ecological debt and technological debt (while in turn improving greenability and maintainability). Finally (though very improbably), if ecological and technical flaws were the same, then refactoring all of them would reduce both kinds of debt and improve greenability and maintainability.

In Sect. 9.2, we advise considering green and classical maintenance under the same process, and we strongly recommend considering ecological and technical debt within the same category. Both kinds of debt are an economic load that would endanger the future of the whole software system and could therefore be considered under system debt, where technical debt and ecological debt are used as a driving force to maintain and improve the system (as well to invest in market opportunities based on the context).

9.5 Case Study: An Attempt to Improve Greenability and Maintainability at One and the Same Time

As has been discussed, there is not a great deal of evidence about how software refactoring has a positive sustainable impact in the maintenance stage. The study presented in this section addresses that question.

The case study presented here is an excerpt from a complete one [23].

9.5.1 Introduction

We have already commented that modifications in the maintenance stage could be understood as a kind of refactoring activity which is carried out through the
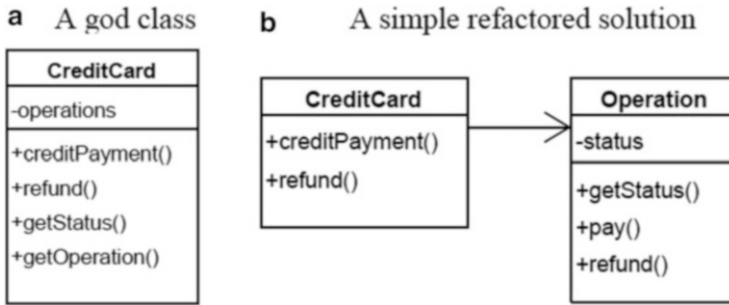
Fig. 9.3 (**a**) A god class and (**b**) a simple refactored solution

modification of the source code by means of different possible refactoring operators.

Validating all the possible refactoring solutions for all the existing bad smells and design flaws is almost impossible. This being so, we undertake an exploratory study considering the *god-class* software anti-pattern [25], as well as the side effects of applying a possible refactoring (see Fig. 9.3). This anti-pattern consists of a given class (the god class) which (1) performs most of the work of the system, (2) plays the role of a controller, and (3) is surrounded by simple data container classes.

We illustrate the core problem through the following toy example. Consider a payment system in which it is necessary to make payments and refunds. On the one hand, Fig. 9.3a shows a fragment of the class diagram for a possible architecture of this system. The *CreditCard* class can be considered as a god class, since it contains almost the whole intelligence. It retrieves each operation and checks the status (accepted or rejected) and, depending on the status, performs the payment or refund. This architecture design is poorly cohesive and highly coupled with data classes. On the other hand, Fig. 9.3b provides a simple refactored solution from which a class for operation intelligence has been extracted. The *Operation* class simply reports its status (accepted or rejected) and responds in order to *pay()* and *refund()* invocations. The *CreditCard* does all the work: it requests information from the *Operation*, makes decisions, and tells the *Operation* class what to do.

As can be seen in Fig. 9.3, the main problem that arises after refactoring the source system consists of the fact that the communication (message interchange) between the god class, the surrounding classes, and the new classes (created after refactoring the anti-pattern) increases dramatically. This consequence establishes the trade-off between maintainability (design quality) and greenability (power consumption). In the remainder of the case study, we try to find out whether this maintainability activity improves greenability or not.

9.5.2 Hypothesis, Context, and Execution of the Case Study

We put forward the following research hypothesis to be dealt with: *power consumption decreases as a result of reducing object message traffic*. The research goal is to demonstrate that when common refactoring patterns are applied under the detection of well-known anti-patterns, they lead to excessive object message traffic, and power consumption is therefore also higher.

The hypothesis was assessed through two industrial (and open source) case studies: *Informa* [9] (which provides an RSS library based on the Java platform) and *NekoHTML* [7] (an HTML scanner to parse HTML documents and enable the access to the information by means of XML interfaces).

We carried out the following steps while executing the case study:

1. Both systems under study are analyzed to detect possible occurrences of the god-class anti-pattern. The analyses were carried out with the *JDeodorant* eclipse plug-in [13] which, in turn, proposes a possible refactoring to solve the problem. Refactoring is then applied, and a new version for each system is obtained: *Informa^R* and *NekoHTML^R*.
2. The next step consists in the measurement of the traffic between objects. In order to carry out a strict comparison between the initial information systems and the refactored ones, the same execution scenario is used; it is based on the existing test cases of both systems. To quantify the object operation invocations, the source code of both systems was traced and profiled (instrumented) by means of the *Eclipse Test & Performance Tools Platform* (TPTP) [5].
3. The power consumption is also measured for both versions of each system. The execution scenario is established using the test cases provided by the software developers. To measure power consumption, the systems are now executed without any instrumentation to avoid bias as much as possible. Measurement is carried out by means of the energy logger *Voltcraft Energy Logger 4000*. This artifact measures energy consumption per second in watts (W). Processor usage is also measured.
4. Once all data has been collected, results are analyzed and some interpretations are given in order to verify the initial hypothesis.

9.5.3 Conclusions of the Case Study

The steps outlined above were followed, and the data collected by the execution of the case studies is summarized in Table 9.5. Table 9.5 provides most of the relevant architectural/design metrics of the original and refactored system under study in addition to the difference between both versions. The upper part of Table 9.5 provides (1) the number of lines of source code, (2) the number of classes, (3) the number of methods, (4) the afferent coupling as the number of classes on average outside a package that depend on classes inside the package, (5) the efferent

Table 9.5 Architectural metrics, message traffic, and power consumption during execution

	Measure	Informa	Informa^R	Dif. (%)	NekoHTML	NekoHTML^R	Dif. (%)
Architecture	#Lines of code	9,739	9,891	1.56	7,938	8,179	3.04
	#Classes	116	127	9.48	60	74	23.33
	#Methods	996	1,024	2.81	473	523	10.57
	Afferent coupling	10	10.5	5.00	5.29	5.43	2.71
	Efferent coupling	7.21	7.57	4.95	5.57	7.29	30.78
	Cycl. complexity	1.87	1.84	−1.18	3.44	3.23	−6.24
Refact.	#God classes	21	0		10	0	
	Ratio god classes	18.1 %	0 %		17 %	0 %	
	# Extrac. classes	49	0		26	0	
Execution	# Test cases	337	337	0.00	4,201	4,201	0.00
	# Errors	71	71	0.00	0	0	0.00
	# Failures	18	19	5.56	1,800	2,200	22.22
	# Messages	6,221	97,846	1,473	1,550,848	7,900,600	409
	Time (s)	57	60	5.26	22	27	22.73
Pw	Total watts	2,052.6	2,207.7	7.56	743.9	893.4	20.10
	Watts/s	36.7	37.4	1.91	33.8	34.4	1.62
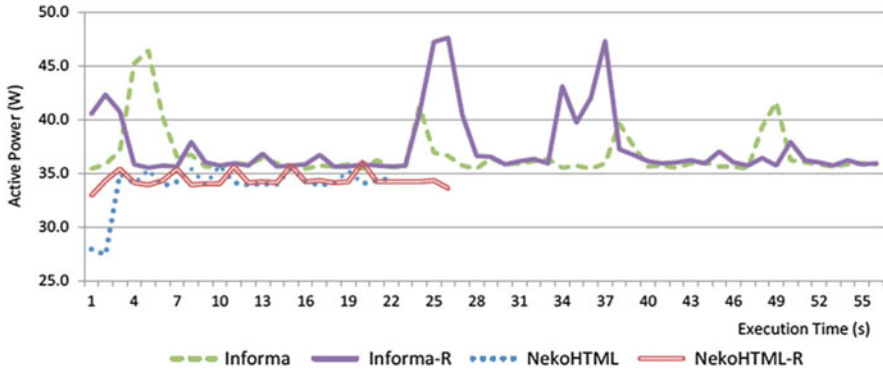
Fig. 9.4 Power consumption during execution

coupling as the number of classes inside a package that depend on classes outside the package, and finally (6) the McCabe cyclomatic complexity, which counts the number of flows through a piece of code.

Despite the huge amount of data included in Table 9.5, the most important thing to highlight for our purpose is the increment of interchange of messages and the increase in power consumption. On the one hand, data in Table 9.5 reveals that the refactored architecture produces between 14 and 4 times more messages for both systems. On the other hand, Fig. 9.4 presents the active power consumption evolution (in watts) during the execution of the original systems and the refactored ones.

The power consumption of *Informa* was 36.7 W, on average, while the consumption of *Informa^R* was 37.4 W, on average. The difference in power consumption was 1.91 %, on average. In the case of *NekoHTML*, the original system consumed 33.8 W/s, while the consumption of *NekoHTML^R* was 34.4, on average. This means an increase of 1.62 %. What is more, since the execution time was higher for the refactored systems, the increases in power consumption (in terms of absolute values) were 7.6 % and 20.1 %, respectively. The different increases in power consumption are probably due to the different execution scenarios for each system (which are based on test cases). These scenarios could lead to the execution of a different amount of parts being affected by inheritance and delegations as a result of god-class refactoring.

In spite of the several limitations and threats to the validity of this study, the experimental results have shown that an architecture in which god classes have been refactored may worsen in terms of power consumption. That is due to the excessive traffic message derived from the architecture refactoring, which in turn leads to a harmful effect on the power consumption of the refactored systems.

9.6   Conclusions

Green maintenance is a novel concept (one of many) that arises as a consequence of the "greening by IT" trend [20]. Current approaches to green maintenance and usage focus the effort of the process on stating what to do to reduce the environmental impact. However, all the activities considered are very directly oriented towards software managers, end-user training, and software retirement procedures and protocols.

A green maintenance point of view like this is correct, but it is too narrow. Many other maintenance activities are put to one side. Classical maintenance considers not only management and training activities but also many other activities related to the different sorts of maintenance: bug removal, adaptation of current systems to technological changes (software and hardware updates), quality improvement or addition of new functionalities. So, what should green maintenance take into account?

In this chapter, we try to foresee a definition, as well as possible techniques to deal with green maintenance, due to the lack of studies dealing with this topic. As with classical maintenance, software refactoring is presented as a powerful tool for dealing with software systems in green maintenance. The difference now is the target of such refactoring: the issue is to keep the greenability level or improve it when software has not been developed according to green and sustainable software engineering [21]. As far as we know, there are neither procedures nor techniques for applying and improving software greenability, as classical maintenance has to improve maintainability or other software quality characteristics.

Some studies (such as the one presented in the case study in Sect. 9.5) find out that facing classical software flaws (such as bad smells or anti-patterns) may not be a suitable solution for improving greenability. Although the ideas presented in this chapter are just the tip of the iceberg, it is essential to determine which of the typical software flaws have an impact on greenability. This is a challenging purpose, since classical maintenance and green maintenance must coexist in real maintenance (a system must be sustainable, but a system must also have good values for its quality characteristics [12]). Melding both facets of maintenance should be studied in detail, because as the case study shows, the refactoring would have opposite outcomes for the two respective kinds of maintenance. It thus becomes a trade-off problem, where software quality and greenability should be balanced depending on the stakeholders' requirements.

Also a novel concept is introduced: the *ecological debt*. As technical debt is useful as a driver for addressing software maintenance, ecological debt would also be useful not only for green maintenance but also for the whole green development cycle. Ecological debt arises in measuring the cost of avoiding the implementation (or of implementing only partially) of green requirements in development time. In the maintenance stage, ecological debt is useful as a way of measuring to what extent a refactoring of the system improves or decreases the greenability characteristic. An important concern that arises from the analysis of ecological debt is the

irrecoverable expense or the fixed cost that is incurred by a software system with a power consumption level above that recommended.

In a nutshell, a possible roadmap for setting up a practical and theoretical framework for an *integrated software maintenance* starts by analyzing the concept of green maintenance, establishing possible activities and kinds of green maintenance. Secondly, it is important to find out to what extent refactoring classic software flaws has an impact on software greenability. Thirdly, we should face up to the fact that classical and green maintenance produce a trade-off problem if we attempt to put them together into a single process. Finally, it is vital to identify green bad smells and green anti-patterns with their corresponding refactoring techniques.

References

1. Arnold RS (1992) Software reengineering. IEEE Press, Los Alamitos, CA, p 675. ISBN 0-8186-3272-0
2. Brown WH, Malveau RC, Mowbray TJ (1998) Antipatterns: refactoring software, architectures, and projects in crisis. Wiley, New York
3. Chikofsky EJ, Cross JH (1990) Reverse engineering and design recovery: a taxonomy. IEEE Software 7(1):13–17
4. Cunningham W (1192) The WyCash portfolio management system. In: Proceeding OOPSLA '92 addendum to the proceedings on object-oriented programming systems, languages, and applications (addendum). ACM
5. Eclipse (2013) Test & performance tools platform project [cited 12/07/2013]; available from: http://www.eclipse.org/tptp/
6. Fowler M (1999) Refactoring: improving the design of existing code. Addison-Wesley, Boston, MA
7. HTML, N. Neko HTML Parser v 1.9.18 (2009) [cited 12/07/2013]; available from: http://nekohtml.sourceforge.net/
8. IEEE (2014) Guide to the software engineering body of knowledge (SWEBOK®). IEEE Computer Society, p 346
9. Informa Project. Informa RSS Java Library (2007) [cited 02/04/2013]; available from: http://informa.sourceforge.net/
10. ISO/IEC, ISO/IEC 14764 (2006) Software engineering – software life cycle processes – maintenance. http://www.iso.org/iso/catalogue_detail.htm?csnumber=39064. ISO/IEC
11. ISO/IEC (2006) Software engineering – software life cycle processes – maintenance. ISO/IEC, p 56
12. ISO/IEC, ISO 25000 (2005) Software product quality requirements and evaluation (SQuaRE) 2013
13. JDeodorant. JDeodorant tool (2012) [cited 02/04/2013]; available from: http://jdeodorant.com/
14. Kruchten P et al (2013) Technical debt: towards a crisper definition report on the 4th international workshop on managing technical debt. Software Eng Notes 38(5):51–54
15. Lehman M (1980) Programs, life cycles, and laws of software evolution. In: Proceedings of the IEEE
16. Lim E, Taksande N, Seaman C, Balancing A (2012) What software practitioners have to say about technical debt. IEEE Software 29(6):22–27
17. Mens T, Tourw T (2004) A survey of software refactoring. IEEE Trans Software Eng 30 (2):126–139

18. Mowbray T (1998) Antipatterns: refactoring software, architectures, and projects in crisis. Wiley, New York
19. Murugesan S (2008) Harnessing green IT: principles and practices. IT Prof 10(1):24–33
20. Murugesan S et al (2013) Fostering green IT – guest editors' introduction. IT Prof 15(1):16–18
21. Naumann S et al (2011) The GREENSOFT model: a reference model for green and sustainable software and its engineering. Sustain Comput Informat Syst 1(4):294–304
22. Pérez-Castillo R, García-Rodríguez de Guzmán I, Piattini M (2011) Business process archeology using MARBLE. Inform Software Tech 53(10):1023–1044
23. Perez-Castillo R, Piattini M (2014) Analyzing the harmful effect of god class refactoring on power consumption. IEEE Software 31(3):48–54
24. Shenoy SS, Eeratta R (2011) Green software development model: an approach towards sustainable software development. In: Proceedings of the annual IEEE India conference (INDICON 2011). IEEE Computer Society, Hyderabad
25. Smith CU, Williams LG (2000) Software performance antipatterns. In: Proceedings of the 2nd international workshop on software and performance. ACM, Ottawa, Ontario, Canada, pp 127–136
26. Visaggio G (2001) Ageing of a data-intensive legacy system: symptoms and remedies. J Software Mainten Evol Res Pract 13:281–308

Chapter 10
Green Software and Software Quality

Coral Calero, Mª Ángeles Moraga, Manuel F. Bertoa, and Leticia Duboc

10.1   Introduction

Quality is currently one of the main goals that organisations set for themselves. A large number of organisations provide products that are similar to each other, thus permitting consumers to choose from a wide variety of brands. Bearing this situation in mind, companies attempt to develop products of better quality; their survival depends to an increasing extent on the quality of the products and services provided.

The need to resolve this issue is also present in the software industry, which has consequently become concerned about ensuring software product quality (PQ). This in turn has led to the appearance of the ISO/IEC 25000 family of standards [6], a family which is divided into five sections, one of which—ISO/IEC 25010— [7] presents various software quality models.

However, as highlighted in [3], none of these models considers sustainability or the ecological aspects of software products. From our point of view, this is also a very important weakness of the standards, since software sustainability is gaining more and more importance in society in general and in industry in particular.

While sustainability is a standardised practice in a number of engineering disciplines, there is currently no such awareness within the software engineering

C. Calero (✉) • Mª.Á. Moraga

Department of Information Technologies and Systems, University of Castilla-La Mancha, Ciudad Real, Spain
e-mail: Coral.Calero@uclm.es; MariaAngeles.Moraga@uclm.es

M.F. Bertoa
University of Málaga, Málaga, Spain
e-mail: Bertoa@lcc.uma.es

L. Duboc
Universidade do Estado do Rio de Janeiro, Rio de Janeiro, Brasil
e-mail: Leticia@ime.uerj.br

community, as noted in [11]; the way to achieve a sustainable software is mainly by improving its power consumption [2]. This, however, is a very restrictive interpretation of what software sustainability is.

The UN identifies three dimensions for sustainable development: social, economic and environmental. In the software context, they could be understood as:

- Social sustainability: This is related to software use (by whom, how and under what circumstances a software may be used).
- Economic sustainability: This is related to aspects of the software business, but not to its development.
- Environmental sustainability: This deals with aspects of energy efficiency and is the type of sustainability that is most closely related to technical aspects, that is in our case those that have to do with the sustainability of software development.

Software product development affects mainly the environment, via the consumption of resources during its use and production. The most direct (and obvious) impact of a software product is on energy consumption, but other resources may also have a negative impact on software sustainability (processor usage, network utilisation and bandwidth).

We believe that it is of prime importance to pay the necessary attention to the environmental dimension of sustainability from the software product development perspective. We term this 'green' software or software 'greenability'. Figure 10.1 provides a representation of this in a diagram form.

When a software product is developed, the requirements that the product should satisfy must be specified. Software requirements can be classified into functional requirements (FR) and nonfunctional requirements (NFR).

According to [5], FR should define the fundamental actions that must take place in the software in accepting and processing the inputs as well as in processing and generating the outputs.

In [4], NFR are defined as requirements that constrain or set some quality attributes upon functionalities.

From these definitions, we can interpret that FR are related to the 'what' of a software product and NFR can be seen as the 'how' of a software product.
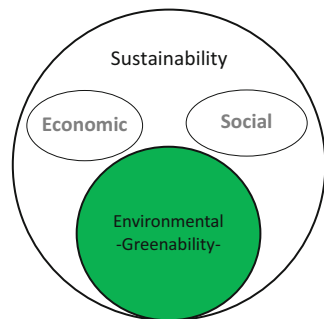
Fig. 10.1 Greenability as environmental sustainability

From our point of view, greenability is a 'how', because it is a way to improve a software product and must therefore be part of its quality.

In this chapter, we present our proposal on how to include greenability in software quality.

10.2   ISO/IEC 25010

The starting point for our work will be the ISO/IEC 25010 [7] Standard, which is the only valid international standard related to software product quality; it is widely used in the industry.

The reason for using this standard, or in general any standard, is to avoid conflicts and inconsistencies regarding the vocabulary used. But it also has another advantage: the use of a standard makes it possible to start with a widely accepted set of quality characteristics that has been agreed on by consensus.

Within the ISO/IEC 25010 [7], the software quality life cycle is presented in the standard (see Fig. 10.2).

This life cycle is divided into three parts: process, software product and effect of software product.

One of the things that affect the quality of a software product is the process quality. From the 'green' point of view, this means that it is advisable to have green development. There are various definitions of the term *sustainable software development*. For example [1]:

'Sustainable Software Development refers to a mode of software development in which resource use aims to meet (product) software needs while ensuring the sustainability of natural systems and the environment'.

Following the quality life cycle, there is a second block related to software product quality (which is what should be dealt with when developing the product).

Finally, the last block is linked to quality in use (QiU), which is related to the users and how the software product behaves with regard to its quality when it is being used.
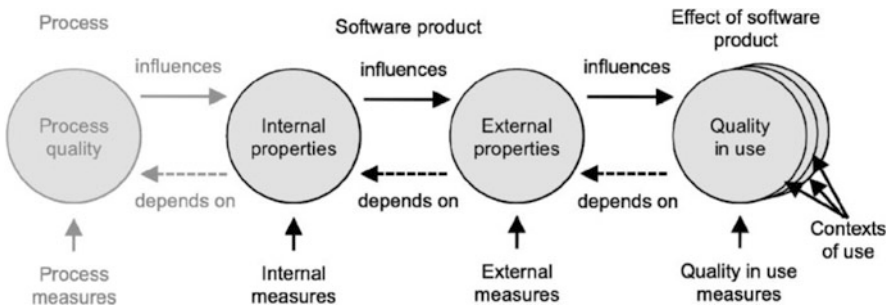
Fig. 10.2 Software quality lifecycle (extracted from [7])

Produce software in a sustainable way.
Sustainable software development

Green Process

Green
Software
product

Product greenability & Greenability in use

Consider greenability NFR
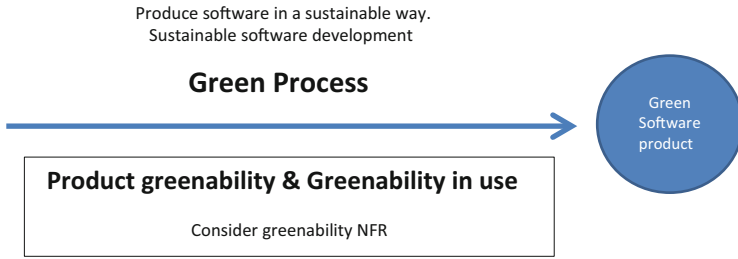
Fig. 10.3 Green aspects that influence software development

It is evident that all three aspects—process quality, product quality and quality in use—are fundamental. And, of course, it would be possible to have a green process whose result will not be a green product and the opposite—a green product that has not been developed by following a green process.

We defend the need for a green process that results in a green product (Fig. 10.3).

It is therefore necessary to include the obligation to use a green process in factories as well as to include greenability aspects in the NFR (nonfunctional requirements) of software products.

In this chapter, we will focus on product greenability as well as greenability in use, although we also give prime importance to the green process. The green process is, however, a way of doing things, and in the standard, it is not represented by means of a model, unlike the other two, that is product greenability and greenability in use.

ISO/IEC 25000 [6] defines three quality models that are shown in Fig. 10.4, in which the targets of the quality models and the related entities are presented.
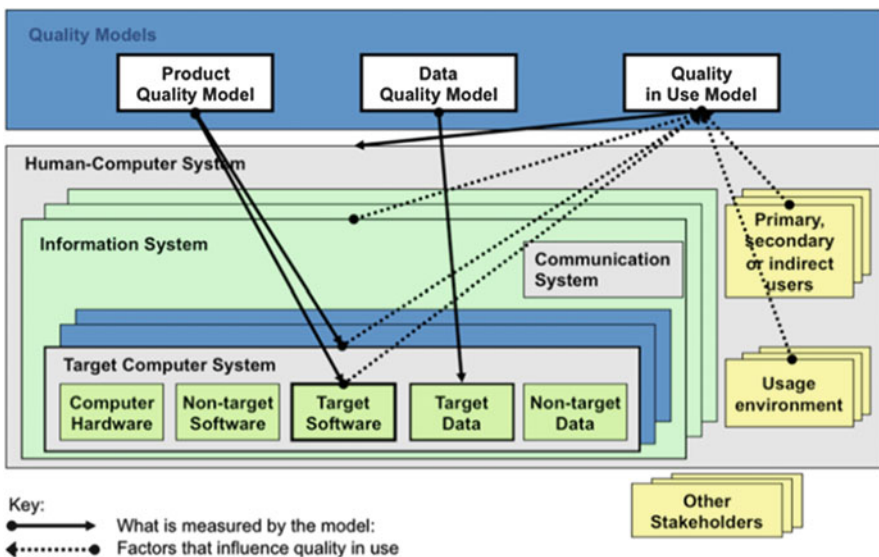
Fig. 10.4 Targets of a quality model (extracted from [6])

In the ISO/IEC 25010, a software product is defined as a 'set of computer programs, procedures, and possibly associated documentation and data. Products include intermediate products, and products intended for users such as developers and maintainers'. It is also necessary to point out that in the SQuaRE standards, software quality has the same meaning as software product quality. From this section on, this meaning will therefore be used in the same manner.

Our focus is thus on the product quality model and the quality in use model defined for the target software. The next two sections address these.

10.3 Software Product Quality and (Software Product) Greenability

As is stated in the standard, 'the product quality model is useful for specifying requirements, establishing measures, and performing quality evaluations. The quality characteristics defined can be used as a checklist in order to ensure a comprehensive treatment of quality requirements, thus providing a basis that can be used to estimate the consequent effort and activities that will be needed during systems development. The characteristics in the product quality model are intended to be used as a set when specifying or evaluating software product quality' [7].

In [7], the product quality model is composed of eight characteristics, each of which is subdivided into several sub-characteristics (see Fig. 10.5). The sub-characteristics can be evaluated on the basis of measurable attributes (for which measures can be defined).

Having presented the quality model, we shall now analyse how software product greenability should be considered in the quality model. The first thing we must check is if greenability is already included in the ISO/IEC 25010 quality model.

To do this, we shall use an example that is provided in [7] (Annex B: Example of mapping to dependability), which shows how an organisation could map its software dependability model onto the ISO/IEC 25010 model. It also attempts to show how to discover whether the model being handled by the organisation can be considered to be embedded in the quality model of the standard. This is done by matching those characteristics from the model that are being studied (which in the case of the example in the standard is 'dependability') with the characteristics and sub-characteristics of the standard, in an effort to verify whether they have already been considered or if they are comparable to those of which the standard is composed.

If we apply this procedure to greenability, then we must first define the greenability model. To do this, we must define what the greenability of a software product is and then attempt to identify the characteristics of which greenability is composed.
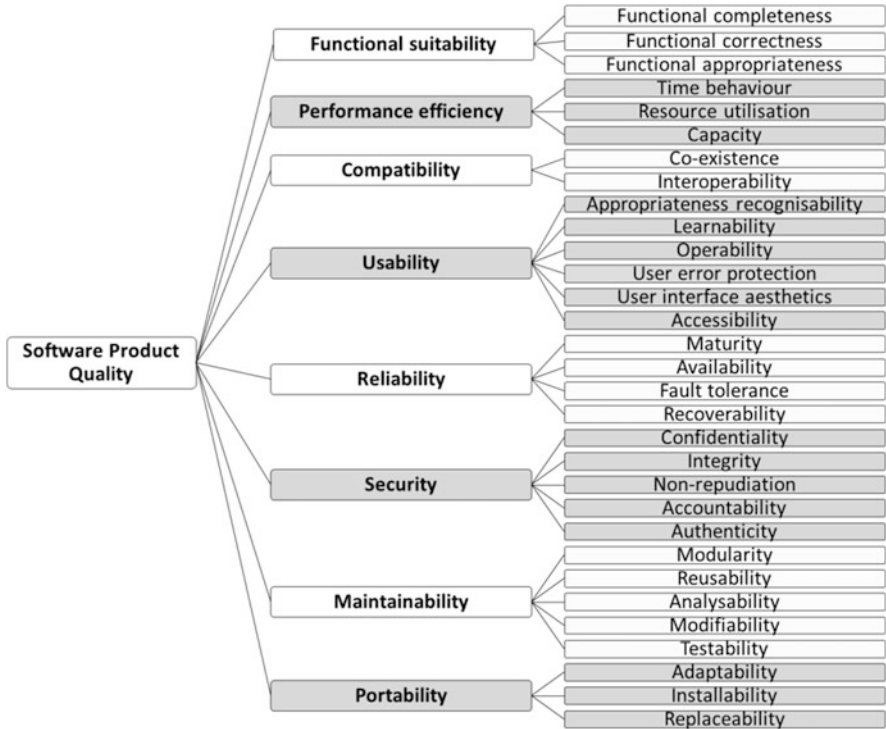
Fig. 10.5 A software product quality model in ISO/IEC 25010 [7]

In order to define what the greenability of a software product is, we use the aforementioned definition of sustainable software development, which states that the fundamental objective of a sustainable software product is to make good use of the resources while minimising their consumption as much as possible. We can therefore consider that a software product's greenability is related to the optimisation of the resources used. We can employ the definition of sustainable software development to identify the aspects of greenability.

It is, therefore, important for us to identify all the aspects related to the resources used in software development: the use of software resources (such as other applications and components), the use of hardware resources (such as disk storage), the use of human resources (e.g. development times) and the use of other material resources (e.g. print paper and ink, storage media).

In addition to these aspects, however, we must not forget that energy consumption should also be taken into account, because it is the main way in which a software can have an impact on the environment.

Having defined the characteristics of greenability, as well as those aspects related to it, we shall now create a conversion table using the characteristics from the quality model of the standard (similarly to what is done in the standard example). The results are shown in Table 10.1.

Table 10.1 Greenability and ISO/IEC 25010

ISO/IEC 25010		Greenability
Performance efficiency Performance relative to the amount of resources used under stated conditions	**Time behaviour** Degree to which the response and processing times and throughput rates of a product or system, when performing its functions, meet the requirements	Hardware resource utilisation Software resource utilisation
	Resource utilisation Degree to which the amounts and types of resources used by a product or system, when performing its functions, meet the requirements	Hardware resource utilisation Software resource utilisation Human resource utilisation Other material resource utilisation Energy consumption
	Capacity Degree to which the maximum limits of a product or system parameter meet the requirements	Hardware resource utilisation Software resource utilisation Other material resource utilisation

The left-hand and centre columns contain the 'performance efficiency' characteristic from ISO/IEC 25010, together with its three sub-characteristics. We have selected this characteristic from the standard because it is obviously the one most closely related to the use of resources. This makes it the best candidate for consideration when studying whether greenability is already included in the standard.

The right-hand column of Table 10.1 contains the aspects that have been identified for greenability and which may be comparable to the sub-characteristics of the standard.

If we focus solely on this comparison and as all the greenability aspects identified match with the sub-characteristics of the standards, we might believe that the standard already contains a characteristic that could perfectly well be considered to be software greenability.

However, if we consider that performance efficiency is the characteristic that deals with the aspects of a software product's greenability, then we are making various mistakes:

- It would not be possible to define specific requirements for performance efficiency (there would only be greenability requirements). This would be a mistake, since the intention of this characteristic is to provide a software product with aspects related to the efficiency that we wish the product to have.
- It is not possible to differentiate between the requirements that we desire for performance efficiency and those for greenability. The former are more closely related to the utilisation of resources, while the latter are related to the optimisation of this utilisation.
- The concept of greenability as a part of the quality is lost, which may make its application confusing. It would be difficult to know whether a requirement defined for performance efficiency originates from performance improvement or from the assurance of greenability (given that we consider them to be the same).
- It would be impossible to define greenability requirements that were not related to performance efficiency.
- Although all the aspects of greenability might appear to be already included in the model, we cannot rule out the possibility of future inclusions of new sub-characteristics in it, something which might not be possible in performance efficiency.

The key factor is that it is necessary to distinguish the good uses of resources both from the performance perspective and from the environmental point of view.

We therefore believe that considering greenability to be already included in the standard is not a correct option and that it is necessary to add to the model a new characteristic related to greenability.

To do this, we must take into account that when a software product is being developed, its greenability can be considered from two points of view.

We must ensure first of all that the software product is energy efficient when it works, using the resources in the most appropriate manner. This, together with the aspects identified previously for greenability, results in the proposal of the following sub-characteristics:

- Energy efficiency: Degree of efficiency with which a software product consumes energy when performing its functions.
- Resource optimisation: Degree to which the resources expended by a software product, when performing its functions, are used in an optimal manner. As in the standard, the authors consider that resources can include other software products, the software and hardware configuration of the system and materials (e.g. print paper, storage media).
- Capacity optimisation: Degree to which the maximum limits of a product or system parameter meet the requirements in an optimal manner, allocating only those which are necessary. As in the standard, the authors consider that

parameters can include the number of items that can be stored, the number of concurrent users, the communication bandwidth, the throughput of transactions and the size of the database.

On the other hand, we must ensure that the software product will last, only needing to be replaced if adapting it to the new circumstances is very difficult to achieve. We refer to this as perdurability.

The idea of making a software that is perdurable is to achieve a software product that is long lasting, modifiable and reusable, that is those aspects that make the software that has been developed last for a long time while at the same time being able to adapt to change without losing its functionality or any other features related to its quality.

In order to define perdurability, we first need to identify what it must consider, and to do so, we are going to use the standard ISO/IEC 25010 (2010). In this standard, there are three characteristics that could be related to the one we are looking for:

- Reliability: Degree to which a system, product or component performs specified functions under specified conditions for a specified period of time

  - Maturity: Degree to which a system meets needs for reliability under normal operation
  - Availability: Degree to which a system, product or component is operational and accessible when required for use
  - Fault tolerance: Degree to which a system, product or component operates as intended, despite the presence of hardware or software faults
  - Recoverability: Degree to which, in the event of an interruption or a failure, a product or system can recover the data directly affected and re-establish the desired state of the system

- Maintainability: Degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers

  - Modularity: Degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components
  - Reusability: Degree to which an asset can be used in more than one system or in building other assets
  - Analysability: Degree of effectiveness and efficiency with which it is possible to assess the impact on a product or system of an intended change to one or more of its parts, to diagnose a product for deficiencies or causes of failures or to identify parts to be modified
  - Modifiability: Degree to which a product or system can be effectively and efficiently modified without introducing defects or degrading existing product quality

  - Testability: Degree of effectiveness and efficiency with which test criteria can be established for a system, product or component, and tests can be performed to determine whether those criteria have been met

- Portability: Degree of effectiveness and efficiency with which a system, product or component can be transferred from one hardware, software or other operational or usage environment to another

  - Adaptability: Degree to which a product or system can effectively and efficiently be adapted to different or evolving hardware, software or other operational or usage environments
  - Installability: Degree of effectiveness and efficiency with which a product or system can be successfully installed and/or uninstalled in a specified environment
  - Replaceability: Degree to which a product can be replaced by another specified software product for the same purpose in the same environment

However, if we look at these sub-characteristics and their definitions in detail, we can discard many of them for not being related to long-term issues (the ones we are looking for). As a result, we obtain a final set of characteristics that could be related to perdurability. The following characteristics must be taken into account when defining the perdurability characteristic: reusability, modifiability and adaptability.

These characteristics will not be used exactly as they are defined for defining perdurability, but they will be used as a basis for doing so, taking into account the objective of perdurability. We can therefore define the new characteristic as follows:

Perdurability: Degree to which a software product can be used over a long period, being, therefore, easy to modify, adapt and reuse.

Once we have identified and defined the sub-characteristics of greenability, we could add to the software product quality model a new characteristic for greenability, proposed by the ISO/IEC 25010 (2010) and defined as shown in Fig. 10.6.

The new software product quality model would therefore be composed of nine characteristics (see Fig. 10.7).
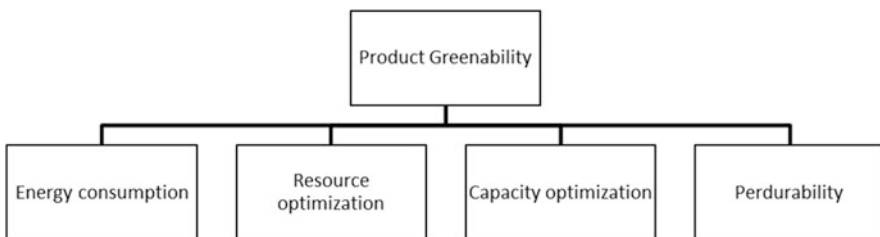
Fig. 10.6 The greenability characteristic

Fig. 10.7   The new software product quality model

We put together all the concepts and definitions of the new characteristic:

– Characteristic

   Greenability: Degree to which a product lasts over time, optimising the param-
   eters, the amounts of energy and the resources used.

– Sub-characteristics

   Energy efficiency: Degree of efficiency with which a software product con-
   sumes energy when performing its functions.
   Resource optimisation: Degree to which the resources expended by a software
   product, when performing its functions, are used in an optimal manner. As in
   the standard, the authors consider that resources include other software
   products, the software and hardware configuration of the system and mate-
   rials (e.g. print paper, storage media).
   Capacity optimisation: Degree to which the maximum limits of a product or
   system parameter meet the requirements in an optimal manner, allocating
   only those which are necessary. As in the standard, the authors consider that
   parameters can include the number of items that can be stored, the number of
   concurrent users, the communication bandwidth, the throughput of trans-
   actions and the size of the database.
   Perdurability: Degree to which a software product can be used over a long
   period, being therefore easy to modify, adapt and reuse.

At this point, it is important to explain why we do not include aspects such as
money, delivery date, etc. As stated in the standard, some software properties are
inherent in the software product, while some others are assigned to it (see Fig. 10.8).
The quality of a software product in a particular context of use is determined by its
inherent properties.

Software properties	Inherent properties	Domain-specific functional properties
		Quality properties (functional suitability, reliability, performance efficiency, usability, security, compatibility, maintainability, portability,)
	Assigned properties	Managerial properties like for example price, delivery date, product future, product supplier

Fig. 10.8 Software properties (extracted from [2])

Inherent properties can be classified as either functional properties or quality properties. Functional properties determine what the software is able to do. Quality properties determine how well the software performs. Quality properties are inherent to a software product and the associated system.

An assigned property is therefore not considered to be a quality characteristic of the software, since it can be changed without changing the software.

10.4 Quality in Use and Greenability (in Use)

The quality in use characteristics relate to the effect of the system in use and are thus a starting point for requirements; they can be used to measure the impact of the quality of the system on use and maintenance.

The software product quality characteristics can be used to specify and evaluate detailed characteristics of the software product that are prerequisites for achieving the desired levels of quality in use.

The requirements for quality in use specify the required levels of quality from the users' point of view. These requirements are derived from the needs of the users and other stakeholders (such as software developers, system integrators, acquirers or owners). The quality in use requirements are used as the target for validation of the software product by the user.

The system quality in use model (Fig. 10.9) in the ISO/IEC 25010 standard is composed of five characteristics, which are further subdivided into sub-characteristics that can be measured when a product is used in a realistic context of use.

In the case of the quality in use, we are going to use another approach to add the new characteristic to the model rather than the one used for the product quality.

However, the idea behind both is the same: to be sure that the characteristic is not already included in the standard as well as to identify and define the sub-characteristics and define the characteristic. In this case, it was also necessary to check the rest of the characteristics and even the quality in use definition itself, because in this model there are a lot of definitions with references to other elements of the model. We prefer to present both in order to show more than one way of doing things.
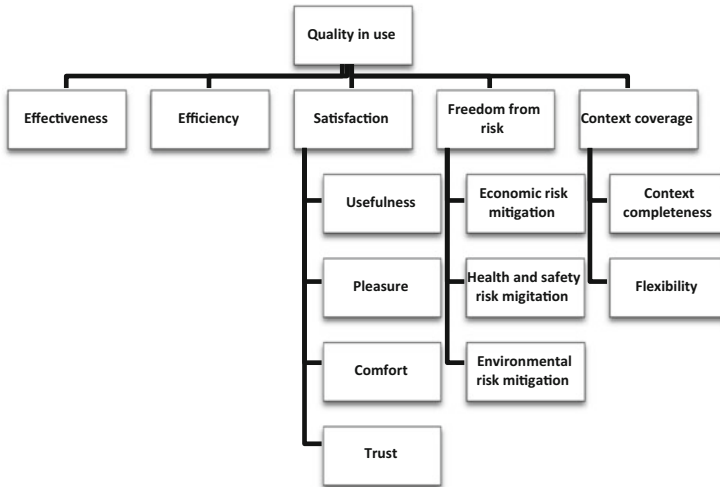
**Fig. 10.9** System quality in use model

To include a new characteristic, related to greenability, in the standard, we are going to follow a set of actions:

1. Working with the sub-characteristics
2. Defining the characteristic
3. Reviewing the quality in use characteristics
4. Redefining quality in use

10.4.1   Working with the Sub-characteristics

In order to identify the sub-characteristics which affect greenability, we have studied the quality in use model from the standard, identifying those characteristics and sub-characteristics that may be related in some way to greenability, adapting them to it. We have also considered the inclusion of new characteristics not derived from the previous step. As a result, we have identified the following sub-characteristics:

- Efficiency optimisation: Optimisation of resources expended in relation to the accuracy and completeness with which users achieve goals. Relevant resources can include time consumption, software resources, etc.
- User's environmental perception: Degree to which users are satisfied with their perception of the consequences that the use of a software will have on the environment.
- Minimisation of environmental effects: Degree to which a product or system reduces the effects on the environment in the intended contexts of use.
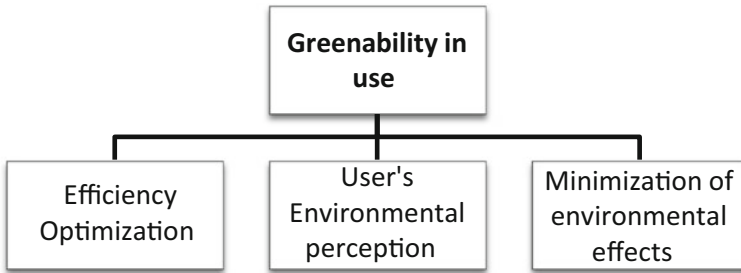
Fig 10.10 Quality in use greenability

10.4.2 Defining the New Characteristic

Once the sub-characteristics had been defined, we were able to define the new greenability characteristic as the degree to which a software product can be used by optimising its efficiency, by minimising environmental effects and by improving the user's environmental perception.

As a result of carrying out the first two steps, we obtained the new characteristic and sub-characteristics shown in Fig. 10.10.

10.4.3 Reviewing Quality in Use Characteristics

As mentioned previously, some of the definitions of quality in use models made use of other definitions of the standard; this means it is necessary to go over all of them in order to incorporate the new characteristic into the definitions. To be specific, the following need to be examined:

Context coverage: Degree to which a product or system can be used with effectiveness, efficiency, freedom from risk, greenability and satisfaction in both specified contexts of use and in contexts beyond those initially explicitly identified

Context completeness: Degree to which a product or system can be used with effectiveness, efficiency, freedom from risk, greenability and satisfaction in all the specified contexts of use

Flexibility: Degree to which a product or system can be used with effectiveness, efficiency, freedom from risk, greenability and satisfaction in contexts beyond those initially specified in the requirements
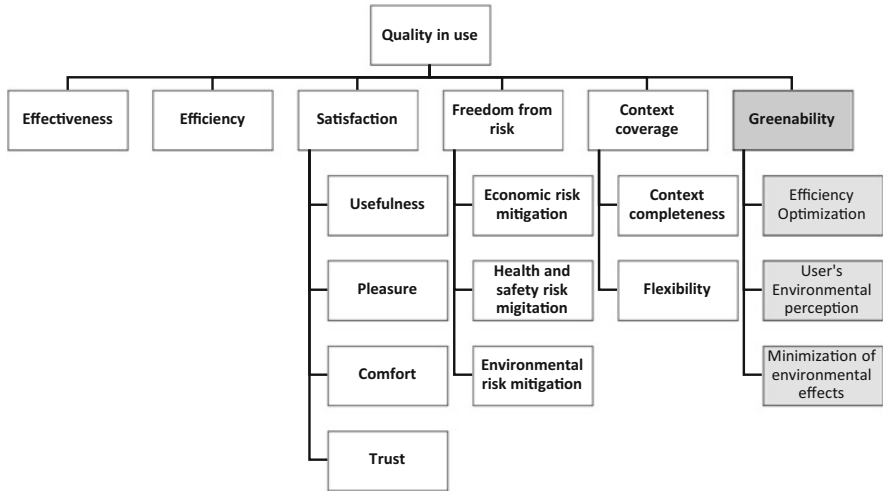
Fig. 10.11 Complete quality in use model

10.4.4   Redefining Quality in Use

Finally, the last step is to check the quality in use definition, making sure that it takes into account the newly added characteristic.

Quality in use is the degree to which a product or system can be used by specific users to meet their needs in order to achieve specific goals with efficiency, freedom from risk, greenability and satisfaction in specific contexts of use.

The final result of this process is the new quality in use model shown in Fig. 10.11.

We put together all the information related to the new characteristic:

– Characteristic

   Greenability: Degree to which a software product can be used by optimising its efficiency, by minimising environmental effects and by improving the user's environmental perception.

– Sub-characteristics

   Efficiency optimisation: Optimisation of resources expended in relation to the accuracy and completeness with which users achieve goals. Relevant resources can include time consumption, software resources, etc.
   User's environmental perception: Degree to which users are satisfied with their perception of the consequences that the use of a software will have on the environment.
   Minimisation of environmental effects: Degree to which a product or system reduces the effects on the environment in the intended contexts of use.

As in the software product quality model, we have also taken into consideration those aspects related uniquely to the inherent properties.

10.5 Linking the Software Product Model and the Quality in Use Model

As indicated in Sect. 10.2 (Fig. 10.2), there is a relationship between the product quality (PQ) and the quality in use (QiU) models. However, no clues on how to make this relationship are given in the ISO/IEC 25010 standard [7].

In the software quality field, people usually work on product quality and focus mainly on the maximisation of the product quality as a means of ensuring high level of quality in use.

This is not necessarily true in most situations, however: a product with the best quality does not necessarily guarantee that the product will fulfil the user's needs in its context of use; for example, a Ferrari is not the best car to go to work in, in most cases.

Our position is exactly the opposite; we concentrate on quality in use as the driving factor to consider when designing a software product or when selecting the product that best fits a user's needs.

This means that we start with a given level of QiU, and we wish to determine the minimum level of product quality that will guarantee such a desired quality in use.

The goal would in fact be to be able to select the smallest set of really relevant product quality sub-characteristics that ensure the required level of quality; focusing solely on these is the way to avoid superfluous costs or irrelevant features which may increase the final impact on the environment unnecessarily while also pushing up the price of the product.

In our quest to determine the relationship between the quality in use (QiU) and the product quality (PQ), we are going to use the Bayesian belief networks (or, simply, the Bayesian networks, BNs). A BN is a directed acyclic graph whose nodes are the uncertain variables and whose edges are the causal or influential links between variables. A conditional probability table (CPT) is associated with each node in order to denote such causal influence [8]. We have successfully applied this approach previously [9], and other authors have also used the BN for the assessment of software quality [10]. The overall idea is to use the BN as a way to represent the models.

To define a BN, it is necessary to:

1. Provide the set of random variables (nodes) and the set of relationships (causal influence) among those variables.
2. Build a graph structure with them.
3. Define conditional probability tables associated with the nodes. These tables determine the weight (strength) of the links of the graph and are used to calculate the probability distribution of each node in the BN.

In our case, steps (1) and (2) will be applied by using the standard, modelling the different relationships between and among the characteristics and sub-characteristics of the PQ and the QiU, in addition to the degree of dependence or influence among them. Once the network has been defined, it has to be 'trained', using a set of controlled experiments, so that it 'learns'.

The trained network can also be used to make inferences about the values of the variables in the network. Bayesian propagation algorithms use probability theory to make such inferences, employing the information available (usually a set of observations or evidences).

10.5.1   Modelling the Relationships Between PQ and QiU

The first step is to identify the relationships between the PQ and the QiU. These relationships can be modelled by determining the characteristics of the former that affect the characteristics of the latter. This is done by using the definitions provided in the standard, along with those provided in this chapter for the new characteristics related to greenability included in the standard. Table 10.2 shows these relationships by employing a matrix, in which 'X' indicates that a relationship exists.

From the information of Table 10.2, we can generate a BN (Fig. 10.12), where each characteristic is represented as a node and each X in the matrix as an arc between the nodes. Nodes from PQ are represented in the top of the BN, and the ones from QiU are represented in the bottom part.

This solution throws up a number of problems:

1. Two characteristics of PQ have the same degree of influence on one of the characteristics of QiU; for example, compatibility and usability have the same influence on satisfaction, which seems to be unreal.
2. Two characteristics of QiU are affected in the same way by one particular characteristic of PQ; for example, functional suitability has the same influence on satisfaction as freedom from risk.
3. All the sub-characteristics of one characteristic of PQ have the same influence on one characteristic of QiU; for example, all the security sub-characteristics (confidentiality, integrity, non-repudiation, accountability, authenticity) have the same influence on freedom from risk.
4. A characteristic of PQ has the same influence on all the sub-characteristics of one QiU characteristic (performance efficiency, for instance, influences the following sub-characteristics of satisfaction—usefulness, trust, pleasure and comfort, all to the same extent).

These findings show that it is better to work with each of the QiU characteristics independently. By way of example, as our present focus is on greenability, we shall make the process concentrate solely on the greenability in use characteristic (although the same process can be applied to the other QiU characteristics).

Table 10.2 Relationships between PQ and QiU

		Quality in use					
		Effectiveness	Efficiency	Satisfaction	Freedom from risk	Context coverage	Greenability
Product quality	Functional suitability	X	X	X	X	X	X
	Performance efficiency		X	X		X	X
	Compatibility			X		X	X
	Usability	X		X		X	X
	Reliability	X		X	X	X	X
	Security				X	X	
	Maintainability				X	X	X
	Portability			X		X	X
	Greenability		X	X	X	X	X

Fig. 10.12 PQ versus QiU Bayesian network

10.5.2 Modelling the Relationships Between PQ and Greenability (in Use)

We are going to follow a similar process to that set out previously but now constructing the matrix between the PQ characteristics and the sub-characteristics of greenability (in use). The result is shown in Table 10.3; Fig. 10.13 shows the corresponding BN.

In this option, we have eliminated problems 2 and 4, but problems 1 and 3 still remain, since we are still working at the level of characteristics in the case of PQ.

The solution is to work with the sub-characteristics of PQ instead of with the characteristics.

10.5.3 Modelling the Relationships Between PQ Sub-characteristics and Greenability (in Use)

Since we are working with the sub-characteristics of PQ, we need to create influence tables for each of the sub-characteristics of greenability in relation to

Table 10.3 Relationship between PQ and greenability (in use)

		Greenability in use		
		Efficiency optimisation	User's environmental perception	Minimisation of environmental effects
Product quality	Functional suitability	X	X	
	Performance efficiency	X	X	X
	Compatibility	X	X	X
	Usability	X	X	
	Reliability		X	
	Security			
	Maintainability	X	X	X
	Portability		X	X
	Greenability	X	X	X
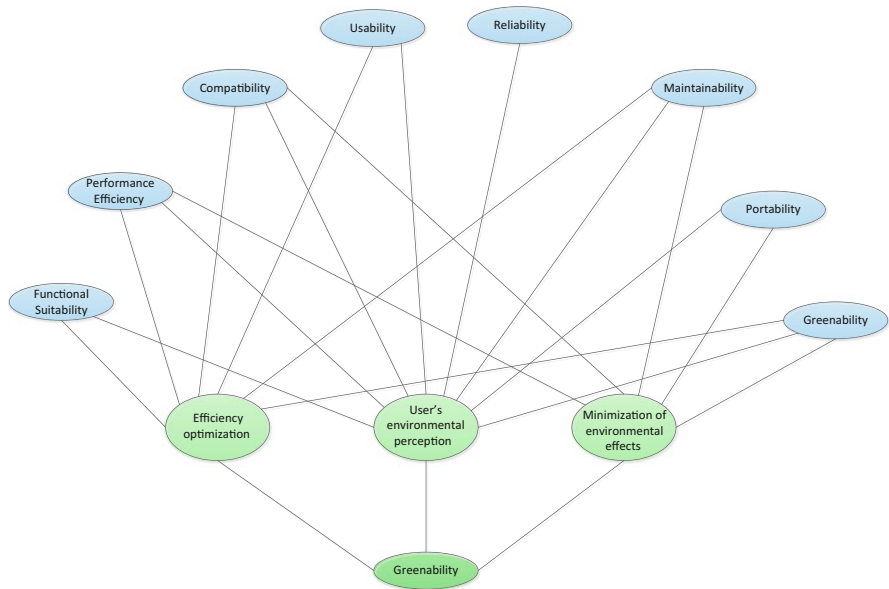
Fig. 10.13 PQ versus greenability (in use) Bayesian network

the sub-characteristics of each characteristic of PQ which have an influence on greenability (in use).

In order to simplify the information, we are going to work separately with each of the three greenability sub-characteristics.

First of all, we present the table (Table 10.4) and BN (Fig. 10.14) corresponding to the efficiency optimisation sub-characteristic.

Table 10.4 Relationship between PQ sub-characteristics and efficiency optimisation

Compatibility	Coexistence	Efficiency optimisation: X	Maintainability	Modularity	Efficiency optimisation: X
	Interoperability	X		Reusability	X
Usability	Appropriateness recognisability	X		Analysability	
	Operability			Modifiability	X
	User error protection	X		Testability	
	User interface aesthetics		Greenability	Energy efficiency	X
	Learnability			Resource optimisation	X
	Accessibility				
Functional suitability	Functional completeness	X		Perdurability	X
				Capacity optimisation	X
	Functional correctness	X	Performance efficiency	Time behaviour	X
	Functional appropriateness			Resource utilisation	X
				Capacity	X
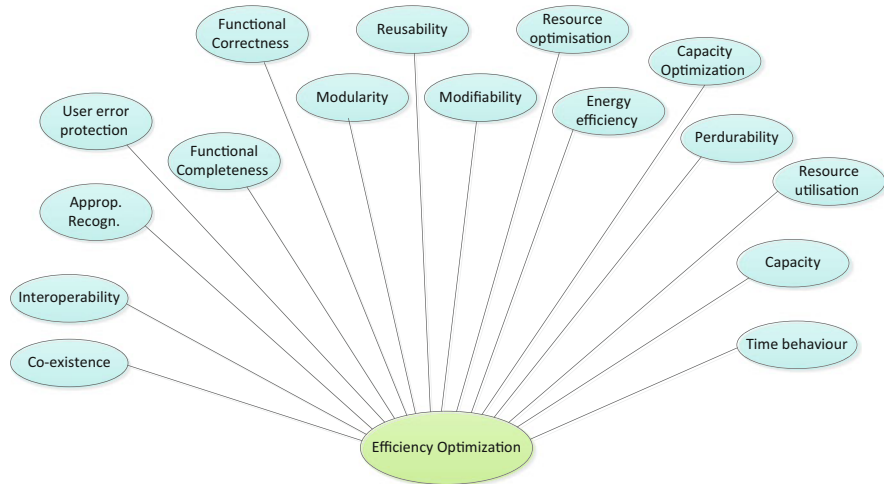
Fig. 10.14 PQ sub-characteristics versus efficiency optimisation Bayesian network

Although this BN reflects the relationships identified, it produces a very high number of entries on the final node (that of efficiency optimisation). The definition of the probability tables is therefore very laborious and cumbersome.

One practice that is commonly used to simplify the relationships in BNs is based on the introduction of synthetic nodes. In this case, we decided to introduce a synthetic node between the PQ sub-characteristics of each characteristic on the standard. We have given these nodes the name of the corresponding characteristic, followed by EO (from efficiency optimisation). Figure 10.15 shows the new BN.

The BN still has a very high number of entries on the final node, even after carrying out the above action. This means it is necessary to introduce synthetic nodes between the PQ characteristics that are conceptually related. The BN obtained (Fig. 10.16) drastically reduces the number of entries in the probability tables, thus becoming a usable BN for our purposes.

Using the same method, we obtain the BN for the other two sub-characteristics. For the sake of simplicity, we will show only the final BN with the synthetic nodes added, using UEP and MEE, respectively, in these nodes on the BN for the sub-characteristics of the user's environmental perception and the minimisation of environmental effects (see Table 10.5, Fig. 10.17, Table 10.6 and Fig. 10.18).

The definition of the three BN (one for each greenability sub-characteristic) makes it possible to study each of them independently, thus making it easier to observe the influence of PQ sub-characteristics on each of these.

This approach not only makes it easier to define the probability tables but also ensures that the network is simpler to use, thanks to the smaller number of nodes.

Once these BNs have been built, they need to be combined in a global BN in order to study greenability as a whole. Figure 10.19 shows this combination that has been made taking into account that there are several parts of the BNs that are
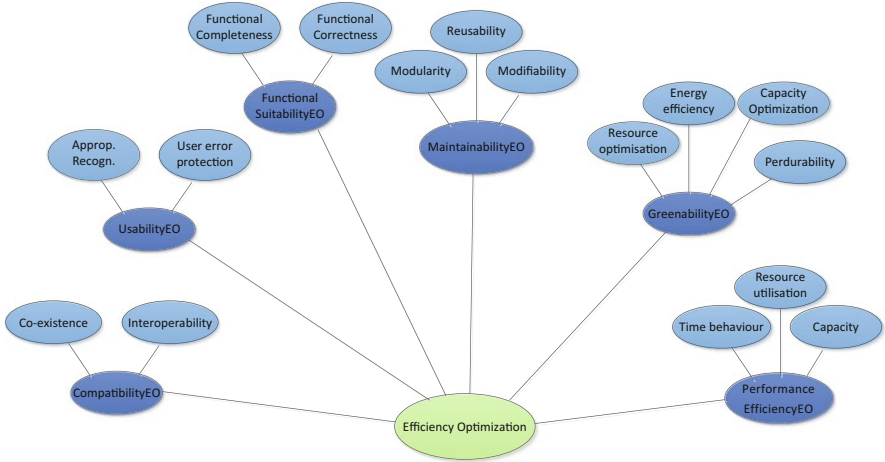
Fig. 10.15 PQ sub-characteristics versus efficiency optimisation Bayesian network with synthetic nodes

Fig. 10.16 Efficiency optimisation Bayesian network

common to two or three of them as well as the fact that it has been necessary to include synthetic nodes.

Of course, we can apply the same approach to the other QiU characteristics, obtaining different BNs that can be combined so as to achieve the complete QiU of a software product.

Table 10.5 Relationship between PQ sub-characteristics and user's environmental perception

		User's environmental perception			User's environmental perception
Portability	Adaptability	X	Reliability	Maturity	X
	Installability			Availability	
	Replaceability	X		Fault tolerance	X
Compatibility	Coexistence			Recoverability	X
	Interoperability	X	Maintainability	Modularity	X
Usability	Appropriateness recognisability	X		Reusability	X
	Operability			Analysability	
	User error protection			Modifiability	X
	User interface aesthetics			Testability	
	Learnability	X	Greenability	Energy efficiency	X
	Accessibility			Resource optimisation	X
Functional suitability	Functional completeness			Perdurability	X
				Capacity optimisation	X
	Functional correctness		Performance efficiency	Time behaviour	X
	Functional appropriateness	X		Resource utilisation	X
				Capacity	X

Fig. 10.17 User's environmental perception Bayesian network

10.6   How to Adapt the Bayesian Networks to a Specific Context

It is clear that the networks that come about in the previous section represent the influences between the PQ and the greenability of any particular software product; its definition is based on those given in the ISO/IEC 25010 and, as such, should be adapted to the specific context to which it is to be applied. In order to carry out this adaptation, we must ensure that all the characteristics of the standard are applicable to this context and that no further characteristics are going to be needed.

To do the former, we have to know what the context is and be able to establish the applicability of the characteristics to this specific context. If there is any shadow of a doubt, it is better not to eliminate characteristics, since once trained, the Bayesian networks themselves will be able to rule out any characteristics that have no influence.

In addition, it will be possible to determine whether or not to include new characteristics of the context by studying the state of the art, looking for other proposals, consulting experts, etc.

In general, if it is not a context with very well-defined features, we have to rely on the standard covering all the quality characteristics.

After these actions have been taken, we will be able to build the structure of the Bayesian network, one that is adapted and fitted to the characteristics and sub-characteristics of PQ and QiU in our context.

The second step we have to take to adapt the proposal has to do with the probability tables. The influences of some given characteristics will obviously vary from domain to domain. That means it is vital to create tables to reflect the specific reality of a particular domain.

Table 10.6 Relationship between PQ sub-characteristics and minimisation of environmental effects

		Minimisation of environmental effects			Minimisation of environmental effects
Portability	Adaptability	X	Greenability	Energy efficiency	X
	Installability			Resource optimisation	X
	Replaceability	X		Perdurability	X
Compatibility	Coexistence	X		Capacity optimisation	X
	Interoperability	X	Maintainability	Modularity	X
Performance efficiency	Time behaviour	X		Reusability	X
	Resource utilisation	X		Analysability	
	Capacity	X		Modifiability	X
				Testability	
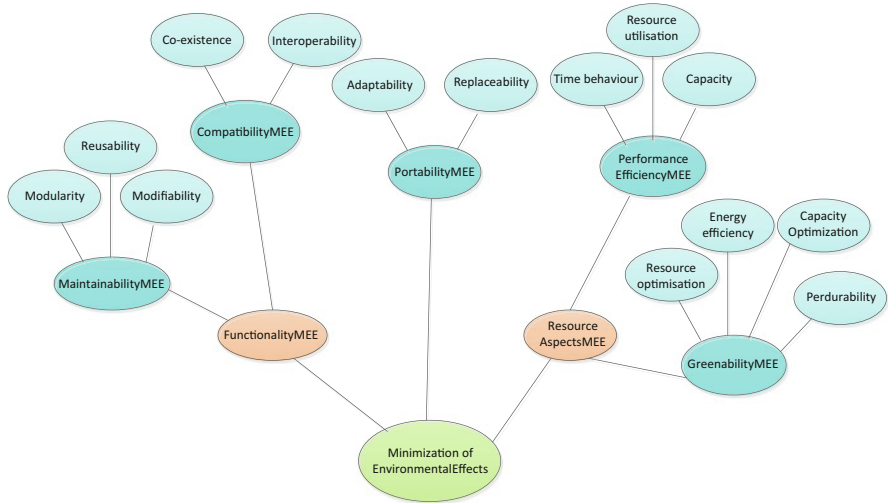
Fig. 10.18 Minimisation of environmental effects Bayesian network
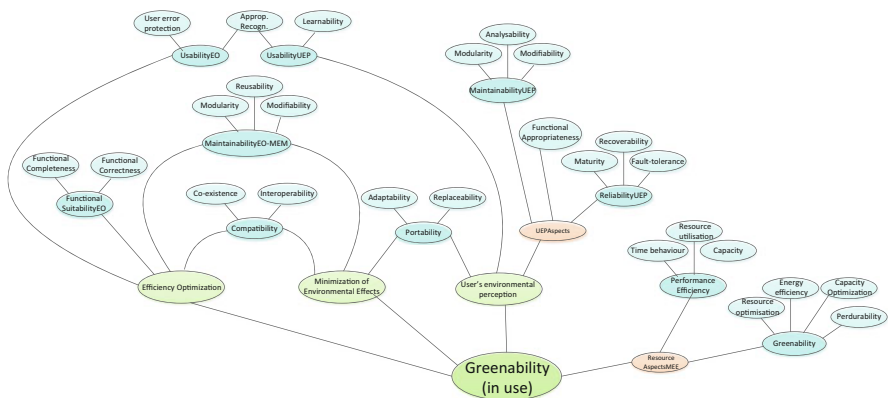
Fig. 10.19 Greenability (in use) Bayesian network

To do that, we must carry out experiments or surveys that allow us to obtain a series of data that serve as input to the network validation process.

From this validation, we will obtain a structure that is completely adapted to the context to which we want to apply it.

The final step in being able to use this network will be the definition of specific measurements for the software product we wish to measure. These measurements should be able to be calculated for the product; this will preferably be automatic, though that is not always possible. These measurements will be the ones which will serve as inputs to the external nodes of the network; their values should be changed into valid inputs to the network, and these values will be propagated through the

BN, via the nodes and by applying the probability tables, until the lower node is reached (the one of quality in use or some of its characteristics).

Once all these actions have been carried out, the network may begin to be used.

10.7 Using the Bayesian Networks

As we commented in the previous section, before being able to use the network, we must adapt it to the particular context that interests us. Once this adaptation has been performed, the network will be ready to be employed with at least three different uses.

One of these could be by using the network in a 'static' way, that is to say, just as it turns out after the validation process. The other two will be using the network in a 'dynamic' way. The objective will be different in each case.

10.7.1 Bayesian Networks 'Static' Usage

In the 'static' use case, we use the network only as a development guide for the new product. In other words, the validated network will give us information about which aspects are relevant from the point of view of the quality in use, enabling us to identify which quality aspects we should be concerned with when we develop the product in question. As is obvious, this case focuses on the product developers who wish to make a competitive product, not only from the point of view of its functional aspects but also placing special emphasis on its nonfunctional aspects (i.e. those to do with quality), thereby obtaining a competitive edge.

10.7.2 Bayesian Networks 'Dynamic' Usage

In this case, the network will be handled with two basic uses in mind; it will be essential to define measurements for the external nodes that make up the input to the network.

These measurements should ideally be automatisable, and it ought to be possible to calculate on the product and for each of the entry attributes of the network.

Once these measures are established and validated, we can use the network for two purposes: the first will be to determine the quality in use of a product once it has been created. This information will obviously let us know if our product has the quality in use or not. In this case, we are using the network in a 'top-down' way, with a given PQ (established by the defined measurements), to calculate the quality in use (calculated by the propagation of the values through the network until they reach the lowest quality in use node).

Where the quality level has not reached the desired level of quality in use, we can use the network for the second objective we have set out: to determine the minimum values of external quality that the product needs to reach the desired quality in use. This is a 'bottom-up' navigation of the network, from the quality in use up to the external nodes. We can ascertain what values the product should have for the input nodes, in other words, based on the measurements defined for the characteristics.

This means that the approach will identify for us those aspects that allow us to reach an appropriate level of PQ based on the QiU level we are looking for. Along with this, we can also find out which aspects of the PQ are non-relevant.

The first of the two uses is fundamental for those acquiring software products, since it enables them to choose the product of the highest quality among several that have the same functionality. The second of the uses, as in the case of the static use once more, has a focus that is meant to appeal mainly to product developers.

10.8 Conclusions

Greenability must be part of the quality of a software product and should be integrated into the quality model, for example that proposed in the ISO/IEC 25010 standard. This means that it will also be included in the quality in use model.

In this chapter, we have presented the extension proposed for the PQ and QiU models of the ISO/IEC 25010 standard.

Moreover, as there is a direct influence between the product quality (which includes a greenability characteristic) and the quality in use (which already integrates greenability), we have presented a Bayesian network that shows the relationships between both. In future work, we plan to work on this Bayesian network, apply it to a specific domain and construct the probability tables, so as to achieve the greenability level of a given software product. We also plan to use Bayesian networks for the greenability evaluation by means of measures and indicators. This is therefore another of our future objectives.

The final goals are, on the one hand, to incorporate greenability in the development of a software product, in the form of nonfunctional requirements, and ensure that the final products are environment friendly and, on the other hand, to define measures and indicators for the greenability of a software product, so we may use them to evaluate, detect weaknesses or improve the greenability of a software product.

Of course, we also wish to continue studying other aspects of greenability, that is the economic, and mainly the social, aspects to which we believe special attention should be paid if we are to point to and mitigate some labour situations that currently occur in the software industry and that should be rejected immediately.

References

1. Calero C, Bertoa MF, Moraga MA (2013) Sustainability and quality: icing on the cake. In: Second international workshop on requirements engineering for sustainable systems (RE4SuSy) in RE'13 (July 15th–19th), vol 995. ISSN:1613-0073, Paper 5, http://ceur-ws.org
2. Calero C, Bertoa MF, Moraga MA (2013) A systematic literature review for software sustainability measures. In: GREENS 2013: Second international workshop on green and sustainable software, pp 46–53
3. Capra E, Francalanci C, Slaughter SA (2012) Is software "green"? Application development environments and energy efficiency in open source applications. Inform Software Tech 54 (2012):60–71
4. Glinz M (2007) On non-functional requirements. In: International conference on requirements engineering, pp 21–26
5. IEEE 830 (1998) IEEE recommended practice for software requirements specifications
6. ISO/IEC 25000 (2010) Systems and software engineering – software product quality requirements and evaluation SQuaRE
7. ISO/IEC 25010 (2010) Systems and software engineering – software product quality requirements and evaluation (SQuaRE) – software product quality and system quality in use models. ISO
8. Jensen FV (2001) Bayesian networks and decisions graphs. Springer, Berlin
9. Moraga MA et al (2008) Evaluating quality-in-use using Bayesian networks. In: 12th ECOOP workshop: quantitative approaches on object oriented software engineering (QAOOSE 2008)
10. Neil M, Krause P, Fenton NE (2003) Software quality prediction using Bayesian networks. In: Khoshgoftaar TM (ed) Software engineering with computational intelligence, Chap. 6. International series in engineering and computer science. Kluwer Academic, Higham, MA. ISBN: 978-1-4615-0429-0
11. Penzenstadler B et al (2012) Sustainability in software engineering: a systematic literature review for building up a knowledge base. In: 16th international conference on evaluation and assessment in software engineering (EASE 2012)

Chapter 11
Green Software Measurement

Mª Ángeles Moraga and Manuel F. Bertoa

11.1 Introduction

People today have lifestyles that put the resources of future generations at risk. There is, however, a growing consciousness of this problem. In fact, civil societies are increasingly requiring manufacturers to incorporate principles of sustainably sound design into their products and to produce these products in an ecologically and socially responsible manner [7]. One of humanity's current challenges is therefore to conserve the environment and attain a sustainable economic, social and personal development.

Both the industry and consumers are reacting to this need, which has led to the appearance of various Green IT initiatives. However, it would appear that the industry is far more conscious of this need than are the consumers. According to [1], users are often unaware of the concept of sustainable software and do not see unsustainable software as an environmental concern.

It is clear that nowadays people are using more and more devices, such as iPads, mobile phones, smartphones, etc. In fact, software systems are part of our day-to-day life, and this signifies that the energy consumption of information and communications technology (ICT) is increasing. Some authors such as [6] state that information technology (IT) plays a predominant role in reducing energy consumption, both as a tool to monitor and optimise the energy efficiency of any production process and as a target of energy efficiency initiatives. There are, however, other authors who are unsure whether or not the energy savings made through the use of

Mª.Á. Moraga (✉)
Department of Information Technologies and Systems, University of Castilla-La Mancha, Ciudad Real, Spain
e-mail: MariaAngeles.Moraga@uclm.es

M.F. Bertoa
University of Málaga, Málaga, Spain
e-mail: Bertoa@lcc.uma.es

ICT overbalance the energy consumed by ICT [8, 13]. This is owing to the fact that ICT can optimise material flows and thus reduce energy consumption [11], but energy consumption by ICT itself, especially through the Web, has been gradually increasing [9]. ICT uses resources and consumes energy in both the construction and the use of hardware and software products and may therefore have a positive or a negative effect on the environment.

It is therefore clear that concerns exist as regards ICT helping to reduce negative effects on the environment. But the majority of works to date have focused more on analysing the concerns related to hardware than those related to software [2, 7]. This may be owing to the fact that it is evident that hardware consumes energy and that this affects sustainability. One very clear case is that of data centres, which require ever-increasing quantities of energy. There are, therefore, many works in the literature related to the energy efficiency of data centres [8].

However, software development should not remain indifferent to the need to construct software products that contribute towards sustainability, both during their creation and use. Although software does not directly consume energy, it greatly affects the consumption of hardware equipment, as it indirectly guides its functioning [6]. An efficient software will indirectly consume less energy by using up less hardware equipment in order to run [14]. In fact in [5], the authors have carried out an experiment in which they have found that different MIS applications that satisfy the same functional requirements and run on the same hardware and operating systems have significantly different amounts of consumption (up to 145 %).

Software is the core of any IT technology, and the way in which a software is developed may therefore have a great influence on the activities that use this software, such as the functions offered, how the IT infrastructure is used or the amount of energy that is needed. Nevertheless, as noted previously, the energy efficiency of software products and the sustainability aspects related to software products in general have been studied less than hardware. According to [6], the software development life cycle and related development tools and methodologies rarely, if ever, consider energy efficiency as an objective. However, this trend is changing, and new proposals have emerged in the last few years.

Green software engineering practices can help companies to reduce or minimise the environmental impact of their software products. In fact, the main objective of green software engineering is to develop software products that will reduce negative environmental impacts. A definition of green and sustainable software can be found in [7] in which the authors define the term as 'the art of defining and developing software products in a way, so that the negative and positive impacts on sustainable development that result and/or are expected to result from the software product over its whole life cycle are continuously assessed, documented, and used for a further optimization of the software product'.

The authors of [16] carried out a systematic literature review (SLR) with the objective of discovering the proposals related to software engineering for sustainability. This work was an updated version of a previous work which was presented in [15]. As a result of the aforementioned SLR, the authors obtained that 62 of the

83 publications selected as being relevant had been published in the previous 3 years (2011–2013). This shows two things. On the one hand, it indicates that researchers are increasingly more concerned about this topic, which has led to a considerable growth in the number of publications during the last few years. On the other hand, it demonstrates that the topic is still in its initial stages and not yet consolidated.

Since this is such a new subject, it may therefore run the risk of beginning with typical problems such as the lack of widely accepted theoretical bases on which to work or the non-application of methodologies that will ensure that things are done correctly. In order to avoid this, the first step is to define a greenability quality model for software products. Therefore, the green quality characteristics which affect software products should be identified and defined. This study and the definition of the model are dealt with in Chap. 10 of this book. However, although the definition of a greenability quality model is the first step and is essential, it is not sufficient. In order to have a useful model, it is necessary to define measures. Bearing all this in mind, in this chapter we focus on those aspects related to measures.

The chapter is organised as follows. Section 11.2 is focused on measurement and the importance of measurement in general. In order to use a common terminology in the topic of measurement, the software measurement ontology (SMO) is also presented in this section. In Sect. 11.3, some specific green measures are presented and classified according to the product greenability quality model presented in Chap. 10, whereas in Sect. 11.4 some examples of SMO application to the definition of green measures are shown. Finally, Sect. 11.5 provides some conclusions.

11.2   Importance of Measurement

The topic of measurement has not appeared recently and dates back to ancestral times. This practice is very common in any type of engineering and no less so in software engineering.

In the software engineering domain, one of the main motivations for measuring has arisen due to a growing interest in this topic and the need for measures in order to make improvements. These improvements may be oriented towards the project, the process or the product. We shall focus on aspects related to the product.

In order to be able to state that one product has more quality or is better than another, it is first necessary to carry out measurements in order to compare the results. Measurements are a good means to understand, monitor, control, predict and test software development and maintenance projects [4] and can be used by professionals and researchers to make better decisions [17]. Another indication of its importance is the fact that measurement is considered in 11 out of 15 KA of the SWEBOK [12].

In general, the software measurement process attempts to attain three principal objectives [10]:

- To help us understand what happens during development and maintenance
- To allow us to control what occurs in projects
- To allow us to improve processes and products

However, software measurement is a relatively new discipline. As a result, the literature, until recently, contained different concepts and terminology that had not been agreed by consensus. There was, therefore, no agreement between users and researchers as to the precise meaning of some commonly used terms, such as 'measurement', 'measure', 'metric', 'measurable attribute', etc. It is even possible to find inconsistencies among the various research proposals in the measurement area.

This situation, along with the objective of harmonising the various software measurement standards and research proposals, led to the appearance of the software measurement ontology (SMO).

Figure 11.1 describes the SMO concepts and relationships represented in Unified Modeling Language (UML).

As shown in the figure, the SMO is organised into four main sub-ontologies:

1. *Software measurement characterisation and objectives*, which establish the context and goals of the measurement
2. *Software measures*, which define the terminologies used in the definition of measures
3. *Measurement approaches*, which describe the different means used to obtain the measurement results for the measures defined
4. *Measurement*, which contains the concepts related to performing the measurement process

The SMO concepts and their definitions are detailed in Table 11.1. Tables 11.2 and 11.3, respectively, provide an excerpt of relationship and attribute tables for the SMO. A complete description of all the tables can be found in [3]. Moreover, the SMO has been represented by using the Web Ontology Language (OWL), and its representation can be found at http://alarcos.inf-cr.uclm.es/ontologies/smo.

11.3   Green Measures

A quality model offers many possibilities regarding its use the use of definitions, measures and indicators being the most important. We used this idea as an objective and then consulted the literature related to either direct or indirect software sustainability measurement, along with other review works that had the same objective. We have therefore studied approximately 200 proposed measures that we consider can be adapted to our greenability quality model.
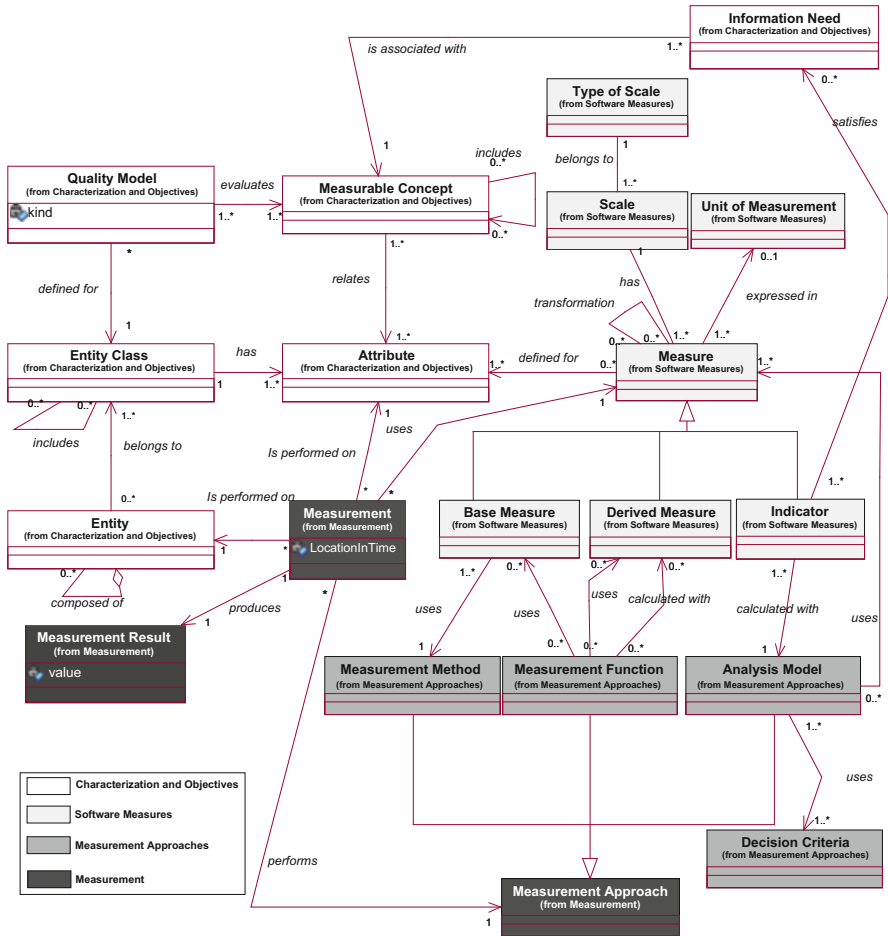
Fig. 11.1 UML representation of SMO concepts, attributes and relationships

We have found 74 measures that are in some way related to software product greenability. We shall discuss them according to how they are calculated and the greenability sub-characteristics with which each one may be related.

It is more common to find indirect or derived measure proposals in the literature, as they usually have greater significance than an isolated value as in direct or base measures. However, we found 34 derived measures that account for 46 % of all the analysed measures. These derived measures are calculated using base measures and/or other derived measures. In our case, we analysed 38 base measures representing 51 % of the total. The others are two indicators which, in International Organization for Standardization (ISO) terminology, are complex measures that need an analysis model based on, among other things, derived and base measures (see Fig. 11.2).

Table 11.1 Concepts of the SMO

Concept	Superconcept	Definition
Information need	Concept	Insight necessary to manage objectives, goals, risks and problems
Measurable concept	Concept	Abstract relationship between attributes of entities and information needs
Entity	Concept	Object that is to be characterised by measuring its attributes
Entity class	Concept	The collection of all entities that satisfy a given predicate
Attribute	Concept	A measurable physical or abstract property of an entity that is shared by all the entities of an entity class
Quality model	Concept	The set of measurable concepts and the relationships between them which provide the basis for specifying quality requirements and evaluating the quality of the entities of a given entity class
Measure	Concept	The defined measurement approach and the measurement scale (a measurement approach is either a measurement method, a measurement function or an analysis model)
Scale	Concept	A set of values with defined properties
Type of scale	Concept	The nature of the relationship between values on the scale
Unit of measurement	Concept	Particular quantity, defined and adopted by convention, with which other quantities of the same kind are compared in order to express their magnitude relative to that quantity
Base measure	Measure	A measure of an attribute that does not depend upon any other measure and whose measurement approach is a measurement method
Derived measure	Measure	A measure that is derived from other base or derived measures, using a measurement function as the measurement approach
Indicator	Measure	A measure that is derived from other measures using an analysis model as the measurement approach
Measurement method	Measurement approach	Logical sequence of operations, described generically, used in quantifying an attribute with respect to a specified scale (a measurement method is the measurement approach that defines a base measure)
Measurement function	Measurement approach	An algorithm or calculation performed to combine two or more base or derived measures (a measurement function is the measurement approach that defines a derived measure)
Analysis model	Measurement approach	Algorithm or calculation combining one or more measures with associated decision criteria (an analysis model is the measurement approach that defines an indicator)
Decision criteria	Concept	Thresholds, targets or patterns used to determine the need for action or further investigation or to describe the level of confidence in a given result
Measurement approach	Concept	Sequence of operations aimed at determining the value of a measurement result (a measurement approach is either a measurement method, a measurement function or an analysis model)

Table 11.1 (continued)

Concept	Superconcept	Definition
Measurement	Concept	A set of operations having the object of determining a value of a measurement result, for a given attribute of an entity, using a measurement approach
Measurement result	Concept	The number or category assigned to an attribute of an entity by making a measurement

**Table 11.2** Relationship table for the measurement approaches sub-ontology

Name	Concepts	Description
Calculated with	Derived measure—measurement function	Every derived measure is calculated with one measurement function. Every measurement function may define one or more derived measures
Calculated with	Indicator—analysis model	Every indicator is calculated with one analysis model. Every analysis model may define one or more indicators
Uses	Base measure—measurement method	Every base measure uses one measurement method. Every measurement method defines one or more base measures
Satisfies	Information need—indicator	An indicator may satisfy several information needs. Every information need is satisfied by one or more indicators
Uses	Measurement function—derived measure	A measurement function may use several derived measures. A derived measure may be used in several measurement functions
Uses	Measurement function—base measure	A measurement function may use several base measures. A base measure may be used in several measurement functions
Uses	Analysis model—measure	An analysis model uses one or more measures. A measure may be used in several analysis models
Uses	Analysis model—decision criteria	An analysis model uses one or more decision criteria. Every decision criterion is used in one or more analysis models

Table 11.3 Attribute table for the measurement sub-ontology

Concept	Attribute	Description	Type	Card
Measurement	Location in time	Time instant where measurement is carried out	Time/date	1
Measurement result	Value	Value which represents the result of the measurement action	Variant	1

According to the greenability sub-characteristics from the product greenability quality model, we have found several measures for all of the proposed sub-characteristics (see Fig. 11.3). However, the distribution among them is uneven. The majority of the measures are related to energy efficiency, and 30 of

Fig. 11.2 Type of
measures
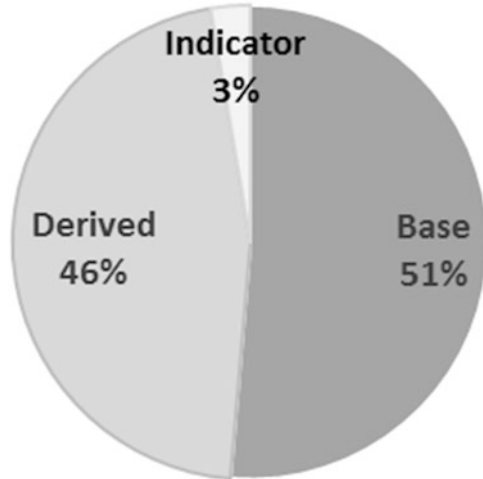
the 81 measures studied can be related to this sub-characteristic. This is not surprising because when we think of sustainability, one of the first and foremost aspects to appear is that of energy consumption.

There are, however, quite a large number of measures that are related to resource optimisation (17) and perdurability (21) (23 % and 28 %, respectively). This might result from the fact that both economic necessity and care of the environment have, from the outset, led to the need to measure spending and the possible optimisation of the resources used for a software product. It is interesting to consider the large number of perdurability measures. Many measures can be associated with this property when we define this sub-characteristic in terms of ease of modifiability, adaptability and reuse, and therefore, it is not surprising that a large number of measures are found.

Finally, for the capacity optimisation sub-characteristic, we have only found six measures (8 %). We think that this smaller number of measures is due to the difficulty in proposing measures to evaluate the optimisation of resource allocation.

The following tables, which are divided into greenability sub-characteristics, show the references and classification of the measures with which we have worked.

Table 11.4 shows the measures that we have been able to relate to energy efficiency. We have followed a flexible inclusion criterion and have assumed that although some measures are not clearly oriented towards a software product, they may, with a minor adaptation, serve to evaluate a particular aspect of this sub-characteristic.

Table 11.5 shows the measures studied that are related to the resource optimisation sub-characteristic. The measures classified into this sub-characteristic should evaluate the consumption and optimisation of the different resources that a software product uses. These resources may be other software products, the system's hardware and software configuration or materials.

Fig. 11.3 Greenability
sub-characteristic measures
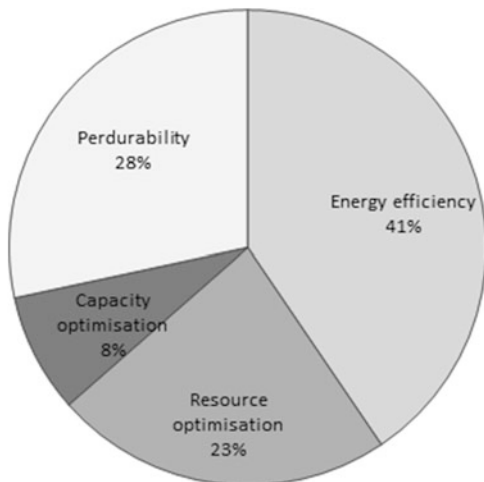
Table 11.6 contains the measures that have been classified as being related to the capacity optimisation sub-characteristic. These measures are all of the derived type, since the sub-characteristic proposed attempts to evaluate the capacity (as regards resources) used in an optimum manner and will normally have to compare the amount of resources that have been assigned or used with the maximum capacity of these resources.

Finally, Table 11.7 presents the measures related to perdurability. Please note that the measures related to ease of use, adaptability and modifiability are classified into this sub-characteristic. Some of the measures have been proposed for the evaluation of these product quality sub-characteristics and, therefore, may also evaluate this greenability sub-characteristic.

In this work, we have only focused on measures assessing software products. We have in fact found far more proposed measures (about 200) related to the development process and the project, at the company level, for the complete system, among others. Some of the measures that we have classified as being for the software product could even be understood to be related not only to the software product but also to the complete system. Nevertheless, our goal was to verify that measures already existed, so we have accepted with a broad and flexible criterion that a measure can be used to evaluate a greenability sub-characteristic attribute.

Some examples will be provided in the following section, in which we shall also attempt to formalise some of these measures.

Another limit that is complex to define is whether a measure is of the product or of the product in use. A software product obviously 'only' consumes energy when it is being executed, that is it is being used, but we consider that greenability in use is closely related to the user, and we have therefore defined the quality in use model, which includes greenability in use (see Chap. 10). Few of the measures studied are

Table 11.4 Energy efficiency measures

Reference	Name	Description/definition	Type
Arnoldus 2013	AE annual consumption	The total energy consumption of an e-service on an annual basis $AE = AE_{hardware} + AE_{communication}$	Derived
	ET consumption per transaction	The average energy consumption per executed end user or business transaction $ET = AE/AT$ where AT is the annual number of transactions	Derived
Das 2008	Power and performance	Power and performance measurements	Base
Goiri 2013	Load t	The average IT power the data centre will consume in epoch t	Derived
Grosskop 2013	Power usage effectiveness (PUE)	Indicator for efficiency of data centre infrastructure	Derived
	Consumption near sweet spot (CNS)	The ratio between the system's average consumption and its optimal consumption per unit of work	Derived
Hindle 2012 a,b	Power per second	Power measures per second	Base
	Power consumption	The relationship between software changes and power consumption	Base
Jiang 2008	Energy usage	Real-time energy usage. It is a comprehensive set of measurements such as real, active and reactive power	Base
Johann 2011	Energy efficiency	Useful work done/energy used	Derived
Kim 2012	Power	Power is the rate of energy consumption, measured in watts (W)	Base
	Energy	Energy = power × time, measured in watt-hour (Wh) rather than joule (J) for energy	Base
	Power consumption rating R(P(i, j))	Power P(i, j) consumed by content j of website i $R(P(i, j)) =$ 1, for 5 * max∀l,m{P(l,m)}/P(i,j) 2, for 4 * max∀l,m{P(l,m)}/P(i,j) < 5 3, for 3 * max∀l,m{P(l,m)}/P(i,j) < 4 4, for 2 * max∀l,m{P(l,m)}/P(i,j) < 3 5, for max∀l,m{P(l,m)}/P(i,j) < 2	Indicator
	Energy consumption E(i)	The amount of energy consumption E(i) of a website i by visitors	Derived
	Energy consumption rating model for websites	Power consumption for websites	Base

Table 11.4 (continued)

Reference	Name	Description/definition	Type
Noureddine 2012	Power software	$P_{software} = P_{comp} + P_{com}$	Derived
	Pcomp	CPU power consumed by software $Pcomp = Pcpu(d) \cdot Ucpu(d)$	Derived
	Pcpu	Global CPU power during d $P_{CPU}(d) = \frac{0.7 \cdot TPD}{f_{TPD} \cdot V_{TPD}^2} \cdot F \cdot V^2$	Derived
	Pcom	Power consumed by the network card to transmit software's data $Pcom = \frac{\sum_{i \in states} tixPixd}{t_{total}}$	Derived
Seo 2008, 2009	Communication energy cost	Energy cost owing to the data exchanged over the network	Base
	Component energy cost	Energy cost of a component due to exchanging subscriptions, unsubscriptions and events with pub-sub connectors	Base
	Facilitation energy cost	Energy cost of a pub-sub connector incurred by managing subscriptions and publications, finding the set of subscriptions that match each published event and creating connection objects that implement remote communication	Base
	Service energy cost	Energy cost of services that a connector may provide	Base
	Client connector energy cost	Energy cost of a client connector incurred by receiving requests from and forwarding responses to clients	Base
	Client energy cost	Energy cost of a client owing to sending requests to and receiving responses from a connector	Base
	Client-server facilitation energy cost	Facilitation energy cost of client and server connectors	Base
	Communication energy consumption	Energy consumption of communication, which includes the cost of exchanging data both locally and remotely	Base
	Conversion pub-sub energy cost	Energy cost of a pub-sub connector incurred by marshalling and unmarshalling events that are transmitted remotely	Base
	Coordination pub-sub energy cost	Energy cost of a pub-sub connector incurred by performing coordination	Base
Sinha 2001	Second-order software energy ESTI model	The amount of current consumed by a program during its execution with different instruction classes	Base

Table 11.5 Resource optimisation measures

Reference	Name	Description/definition	Type
Albertao 2010	Relative response time	The number of tasks with an unacceptable response time divided by the number of tasks tested	Derived
	Learnability	Ratio of how fast a user can learn to use the application w.r.t. the time the user used it	Derived
	Effectiveness	Ratio of tasks accomplished without help w.r.t. the total number of tasks	Derived
	Error rate	Ratio of errors w.r.t. the number of tasks	Derived
Amsel 2011	Green Tracker	Percentage CPU usage	Base
	Executed instruction count measure (EIC)	The number of assembly instructions executed considering a typical embedded integer processor core	Base
Heisig 2004	Model tree	Resource utilisation	Base
Hindle 2012 a,b	System activity information	CPU, memory, disk, network, etc., measures	Base
	Combination of previous	Combine both and synchronise them with timestamps	Derived
Kip 2011, 2012	Consumable	Volume of consumables generated by the application during its workflow execution	Base
	Availability	Probability that a request is correctly fulfilled within a maximum expected time frame	Derived
	Human resources	Costs of human factors affecting the software life cycle	Base
	Response time	The time taken by a service to handle user requests	Base
Lami 2013	Percentage of used	Percentage of virtual servers used versus percentage of physical servers used	Derived
Marzolla 2012	Response time	Collecting performance measures at runtime, response time R, throughput X and individual device utilisations, $U_k, k = 1\ldots K$	Base
Medland 2010	Print consumption	Metrics summarising print consumption data for dynamic time and personnel ranges	Base
Noureddine 2012	Process CPU usage during d	$U_{CPU}^{PID}(d) = \frac{t_{CPU}^{PID}}{t_{CPU}}(d)$	Derived

clearly related to this aspect of greenability. We believe that, as has occurred with quality in use, greenability in use will become more important in the future.

This aspect is normal in any software product quality assessment, in which the tendency is to maximise the product quality as a means to achieve the best quality in use. We do not believe that this is the best way to work with quality, and we advocate another way to plan the measurement: first, fix the quality in use preferences, and then ensure the product quality characteristics that are necessary to achieve them.

Table 11.6 Capacity optimisation measures

Reference	Name	Description/definition	Type
Arnoldus 2013	RE, relative efficiency	RE = Etoptimal/ET The efficiency of the e-service with regard to its optimal efficiency (typically at maximum load)	Derived
Goiri 2013	Workload t	The amount of computational load offered (measured in terms of average power per epoch, including the covering subset) in epoch t	Derived
Grosskop 2013	Fixed to variable energy ratio	FVER = EUfixed/Euvariable EUfixed is the consumption when idle. Euvariable is the difference between this idle baseline and the maximum energy consumption per unit of work	Derived
Kip 2011, 2012	Application performance	Performance MEAS with regard to energy consumption	Derived
	Asset efficiency	IT resource efficiency in terms of energy and utilisation	Derived
Lami 2013	Percentage of used	Percentage of used functionalities/features of the tools supporting development actually used with regard to the full set of functionalities/features provided	Derived

11.4   Definition of Green Measures Using SMO

In this section, we present a set of measures that have been defined according to the SMO described in Sect. 11.3. In general, the entities that we wish to evaluate will be software products (or a specific category of them), and the information need that we have is to evaluate the greenability of this software or a particular property that is related to greenability. Usually, the measures proposed by the authors are presented in a very general manner. Upon formalising them, we have assumed and stated certain aspects that were not proposed by the original authors. For example, the IoR (index of regeneration) indicator and its analysis model have not been proposed by Soto and Ciolkowski in [18], but formalising the measures and having to think about all their details by following the SMO have been an interesting exercise.

11.4.1   Measure Related to Energy Efficiency

Level of power consumption
– **Description**: Used to classify the content of websites at various levels of power consumption
– **Entity**: Websites
– **Information need**: Evaluation of websites for energy saving
– **Measurable concept**: Energy efficiency
– **(Sub-)characteristic**: Energy efficiency

Table 11.7 Perdurability measures

Reference	Name	Description/definition	Type
Albertao 2010	Modifiability (D)	$D = (A + I) - 1$	Derived
	Abstractness (A)	$A = Na/Nc$. How much a package can withstand change	Derived
	Instability (I)	$I = Ce/(Ce + Ca)$. Potential impact of changes in a given package	Derived
	Estimated system lifetime	Estimated number of years in which the minimum hardware required by the system will reach the market	Base
	Support rate	The number of user questions that required assistance divided by the number of minutes the system was used in a given session	Derived
	Estimated installation time	The amount of time the user takes to install the product without assistance	Base
	Defect density	Known defects (found but not removed)/LOC	Derived
	Testing efficiency	Defects found/days of testing	Derived
	Testing effectiveness	Defects found and removed/defects found	Derived
	Support for motor-impaired users	Based on Web accessibility initiative (WAI)	Base
	Visually impaired users	Based on Web accessibility initiative (WAI)	Base
	Blind users	Based on Web accessibility initiative (WAI)	Base
	Users with language and cognitive disabilities	Based on Web accessibility initiative (WAI)	Base
	Illiterate users	Based on Web accessibility initiative (WAI)	Base
	Internationalisation and localisation	Based on Web accessibility initiative (WAI)	Base
	Distance from the main sequence	The ability to make changes quickly and cost effectively	Base

Kip 2011, 2012	Compliance	Cost of guaranteeing the conformity degree as regards regulations and policies established by third parties	Base
	Recoverability	The capability of a service to restore the normal execution after a failure within a given period of time	Base
	Reliability	Probability that a service remains operational to deliver the desired function over a specified period of time	Derived
Seacord 2003	WMRD (weighted modification requests days)	$WMRD = \sum_{i}^{\# \, of \, OpenModReq} EstimatedChangeEffort \cdot (SnapshotDate - submissionDate)$ With i from 1 to the number of open modification requests	Derived
Soto 2009	Sustainability	The likelihood that an F/OSS community will continue to be able to maintain the product or products it develops over an extended period of time	Indicator

– **Attribute**: Energy usage
– **Indicator**: Ratio of power consumption—*R(P(i,j))*
– **Derived measure**: $\frac{\max(\forall l, m\{P(l,m)\})}{P(i,j)}$
– **Base measure**: $P(i,j)$, power consumed by content j of website i
– **Measurement method**: Using the Green Tracker tool
– **Measurement function**:

$$R(P(i,j)) = \begin{cases} 1, \text{ for } 5 * \dfrac{\max(\forall l, m\{P(l,m)\})}{P(i,j)} \\[2mm] 2, \text{ for } 4 * \dfrac{\max(\forall l, m\{P(l,m)\})}{P(i,j)} < 5 \\[2mm] 3, \text{ for } 3 * \dfrac{\max(\forall l, m\{P(l,m)\})}{P(i,j)} < 4 \\[2mm] 4, \text{ for } 2 * \dfrac{\max(\forall l, m\{P(l,m)\})}{P(i,j)} < 3 \\[2mm] 5, \text{ for } \dfrac{\max(\forall l, m\{P(l,m)\})}{P(i,j)} < 2 \end{cases}$$

– **Scale**: Integer from 1 to 5
– **Units**: Absolute values
– **Comments**: The authors set the threshold such that the worst grade (=5) begins at two times the average power consumption value. The authors work with R(P (i,1)), that is j = 1, which is the content of the main page that generally has flash content which consumes a considerable amount of electrical energy.
– **Source**: *Taeseong Kim, Yeonhee Lee and Youngseok Lee. 2012. Energy measurement of web service. In Proceedings of the 3rd International Conference on Future Energy Systems: Where Energy, Computing and Communication Meet (e-Energy '12)*

11.4.2 Measure Related to Resource Optimisation

Percentage of CPU usage
– **Description**: Estimate energy consumption according to the use of the CPU for Internet browsers
– **Entity**: Internet browsers
– **Information need**: To assess energy consumption for environmental purposes
– **Measurable concept**: Software greenability
– **(Sub-)characteristic**: Resource optimisation
– **Attribute**: CPU usage
– **Derived measure**: % CPU usage
– **Base measure**: Time CPU usage
– **Measurement method**: Using the Green Tracker tool
– **Measurement function**: Time CPU usage/1 min × 100
– **Scale**: Real number from 0 to 100

– **Units**: Percentage
– **Comments**: The time that the CPU has been used is measured each minute. Although the authors state that they measure energy consumption, they are really measuring the use of the CPU. They affirm that the higher use of the CPU, the higher the energy consumption, but they do not measure the energy consumed.
– **Source**: *Nadine Amsel, Zaid Ibrahim, Amir Malik and Bill Tomlinson. 2011. Toward sustainable software engineering (NIER track). In Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*

11.4.3   Measure Related to Durability

Index of regeneration
– **Description**: The index of regeneration is the average for all the developers as regards the time that they spend making contributions with respect to the total time that the project lasts.
– **Entity**: Free/open source software
– **Information need**: Perdurability of F/OSS
– **Measurable concept**: Community sustainability
– **(Sub-)characteristic**: Perdurability
– **Attribute**: Community regeneration
– **Indicator**: Index of regeneration (IoR). We can define an indicator that takes into account the total time taken to complete the project and the amount of time that each developer has been involved with the project. The index of regeneration is calculated for each developer. As an indicator, it is possible to calculate the average of individual indices. If the average is high ($>=0.75$), this will indicate a low regeneration; if it is low ($<=0.25$), this will indicate a high regeneration.
– **Derived measure**: The amount of time a developer has worked on project i, TT(i)
– **Base measures**: Time of the first contribution of developer I (TFC(i)). Time of the last contribution of developer I (TLC(i)). Time of the first contribution (TFC). Time of the last contribution (TLC)
– **Measurement method**: Looking at the first and last contributions of each developer throughout the project's history
– **Measurement function**:

  1. $TT = TLC - TFC$
  2. $TT(i) = TLC(i) - TFC(i)$
  3. $IoR(i) = TT(i) - TT$
  4. $IoR = Average(IoR(i))$

– **Scale**: Real number from 0 to 1
– **Units**: Index $-$ (day/day)
– **Comments**: Community sustainability is affected by factors such as the composition of a community and its ability to grow or regenerate. With regard to

growth and regeneration, a community's history in this respect can be a good predictor of its future behaviour. If the same core group of developers has been active throughout the project history, this does not reflect significant regeneration.

– **Source**: *Martin Soto and Marcus Ciolkowski, 2009. The QualOSS open source assessment model measuring the performance of open source communities. In Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement (ESEM '09)*

11.5 Conclusions

Nowadays, society is increasingly concerned about looking after the environment. Each day, we are bombarded with numerous advertisements that tell us to recycle, switch off lights, not waste water, etc. The ICT area cannot remain indifferent to this effort. Numerous proposals from the ICT community have therefore appeared which can be classified from two perspectives. On the one hand, proposals have appeared whose objective is to attempt to reduce the negative impacts on the environment as a consequence of the use of ICTs themselves (Green ICT). Moreover, on the other hand are those proposals that use ICT to reduce the negative impacts on the environment in other areas (Green by IT).

The majority of these proposals are more focused on the hardware aspect and particularly deal with improving energy efficiency. However, the development and use of a software product also have an impact on the environment. Software developers, industry and even users are therefore increasingly concerned about this topic. Green software engineering, whose importance has increased over the last few years, has consequently arisen, and in fact in the last 3 years, the majority of the works dealing with this topic have appeared [16].

Bearing in mind both this concern and the few works that exist in relation to measurement, in this chapter we have focused on studying measurement within the context of green software.

We have first set out the measurement bases, stressing the importance of measuring and presenting a software measurement ontology (SMO) as a basis for the use of a common terminology. We have then focused on the green aspect and have identified a set of measures that have been proposed in the literature for this context. We have studied a set of 192 measures proposed by several authors and have selected 74 measures that we understand to be related to software product greenability. We have seen how these 74 measures can be classified as regards the greenability sub-characteristics proposed in the greenability quality model (presented in Chap. 10). We have seen that all the sub-characteristics can take advantage of the measures that have been already proposed in the literature, although we have verified that there are considerably fewer measures for capacity optimisation than for the other three sub-characteristics, which are well represented.

However, the vast majority of the measures proposed appear in a very informal and general form. We have therefore used the concepts and terms proposed in the SMO in order to create three examples of how these measures can be presented in a more formal manner. It is important to stress that we have created the majority of the information appearing in this section based on the information found in the works used as a source for each measure and that we have made quite a few suppositions of our own. In this respect, it is important to highlight that upon attempting to formalise any measure and concentrate on its details by following the SMO, various aspects come to the fore that are necessary to think about and discuss and that can be avoided when an ambiguous and general proposition is formed.

Appendix: References of Tables 11.4, 11.5, 11.6 and 11.7

Reference	Authors	Title	Journal/Proceedings
Albertao 2010	Albertao, F., Xiao, J., Tian, C., et al.	Measuring the sustainability performance of software projects	e-Business Engineering (ICEBE), 2010 IEEE 7th International Conference on, pp. 369–373. 2010
Amsel 2011	Amsel, N., Ibrahim, Z., Malik, A. and Tomlinson, B.	Toward sustainable software engineering (NIER track)	Proceedings of the 33rd International Conference on Software Engineering, pp. 976–979. 2011
Arnoldus 2013	Arnoldus, J., Gresnigt, J., Grosskop, K., and Visser, J.	Energy-efficiency indicators for e-services.	Proceedings of the 2nd International Workshop on Green and Sustainable Software (GREENS '13), pp. 24–29. 2013
Das 2008	Das, R., Kephart, J.O., Lefurgy, C., et al.	Autonomic multi-agent management of power and performance in data centers	Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems: industrial track, pp. 107–114. 2008
Goiri 2013	Goiri, I., Katsak, W., Le, K., et al.	Parasol and GreenSwitch: managing datacenters powered by renewable energy.	SIGARCH Comput. Archit. News 41, 1 (March 2013), 51–64.
Grosskop 2013	Grosskop, K.	PUE for end users - Are you interested in more than bread toasting?	Proceedings of 2nd Workshop Energy Aware Software-Engineering and Development. EASED@BUIS 2013
Heisig 2004	Heisig, S. and Moyle, S.	Using model trees to characterize computer resource usage	Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems, pp. 80–84. 2004.

Reference	Authors	Title	Journal/Proceedings
Hindle 2012a	Hindle, A.	Green mining: a methodology of relating software change to power consumption	Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on, pp. 78–87. 2012
Hindle 2012b	Hindle, A.	Green mining: investigating power consumption across versions	Proceedings of the 2012 International Conference on Software Engineering, pp. 1301–1304. 2012
Jiang 2008	Jiang, X., Dawson-Haggerty, S., Taneja, J., et al.	Creating greener homes with IP-based wireless AC energy monitors	Proceedings of the 6th ACM conference on Embedded network sensor systems, pp. 355–356. 2008
Johann et al. 2011	Johann, T., Dick, M., Naumann, S., and Kern, E.	How to measure energy-efficiency of software: Metrics and measurement results	Green and Sustainable Software (GREENS), 2012 First International Workshop on, pp. 51–54. 2012
Kim 2012	Kim, T., Lee, Y., and Lee, Y.	Energy measurement of web service	Proceedings of the 3rd International Conference on Future Energy Systems: Where Energy, Computing and Communication Meet, pp. 27:1–27:8. 2012
Kip 2011	A. Kipp, J. Liu, T. Jiang, et al.	Approach towards an energy-aware and energy-efficient high performance computing environment	IEEE International Conference on In Intelligent Computer Communication and Processing (ICCP), pp. 493–499, Aug. 2011.
Kip 2012	A. Kipp, T. Jiang, M. Fugini, and I. Salomie	Layered green performance indicators	Future Generation Computer Systems 28(2): pp. 478–489, Feb. 2012
Lami 2013	G. Lami, F. Fabbrini, and M. Fusani	A methodology to derive sustainability indicators for software development projects	Proceedings of the 2013 International Conference on Software and System Process (ICSSP 2013), pp. 70–77. 2013
Marzolla 2012	Marzolla, M.	Optimizing the energy consumption of large-scale applications	Proceedings of the 8th international ACM SIGSOFT conference on Quality of Software Architectures, pp. 123–132, 2012
Medland 2010	Medland, R.	Curbing paper wastage using flavoured feedback	Proceedings of the 22nd Conference of the Computer-Human Interaction Special Interest Group of Australia on Computer-Human Interaction, pp. 224–227, 2010
Noureddine 2012	Noureddine, A., Bourdon, A., Rouvoy, R., and Seinturier, L.	A preliminary study of the impact of software engineering on GreenIT	First International Workshop on Green and Sustainable Software (GREENS), pp. 21–27, 2012

Reference	Authors	Title	Journal/Proceedings
Seacord 2003	Seacord, R., Elm, J., Goethert, W., et al.	Measuring software sustainability	Proceedings of International Conference on Software Maintenance, ICSM 2003, pp. 450–459, 2003.
Seo 2008	C. Seo, S. Malek, & N. Medvidovic	Component-Level Energy Consumption Estimation for Distributed Java-Based Software Systems	Proceedings of the 11th International Symposium on Component-Based Software Engineering (CBSE '08), pp. 97–113, 2008.
Seo 2009	C. Seo, G. Edwards, D. Popescu, et al.	A framework for estimating the energy consumption induced by a distributed system's architectural style	Proceedings of the 8th international workshop on Specification and verification of component-based systems (SAVCBS '09). 2009
Sinha 2001	A. Sinha	Energy efficient operating systems and software	Doctoral Thesis in Electrical Engineering and Computer Science at the Massachusetts Institute of Technology, August 2001
Soto 2009	Soto, M. and Ciolkowski, M.	The QualOSS open source assessment model measuring the performance of open source communities	3rd International Symposium on Empirical Software Engineering and Measurement, ESEM 2009., pp. 498–501, 2009

References

1. Amsel N, Ibrahim Z, Malik B (2011) Tomlinson toward sustainable software engineering: NIER track. In: 33rd international conference on software engineering (ICSE), pp 976-979
2. Benini L, de Micheli G (2000) System-level power optimization: techniques and tools. ACM Trans Des Autom Electron Syst 5(2):115–192
3. Bertoa MF, García F, Vallecillo A (2006) An ontology for software measurement. In: Calero FRC, Piattini M (eds) Ontologies for software engineering and technology. Springer, Berlin, pp 175–196
4. Briand LC, Morasca S, Basili VR (1996) Property-based software engineering measurement. In: IEEE transactions on software engineering, pp 68–85
5. Capra E, Formenti G, Francalanci C, Gallazzi S (2010) The impact of MIS software on IT energy consumption. In: European conference of information systems
6. Capra E, Francalanci C, Slaughter SA (2012) Is software "green"? Application development environments and energy efficiency in open source applications. Inform Software Tech 54 (1):60–71
7. Dick M, Drangmeister J, Kern E (2013) Naumann green software engineering with agile methods. In: 2013 2nd international workshop on green and sustainable software (GREENS), pp 78–85
8. Dick M, Naumann S (2012) Enhancing software engineering processes towards sustainable software product design. In: Greve K, Cremers AB (eds) EnviroInfo 2010: integration of

environmental information in Europe. Proceedings of the 24th international conference EnviroInfo. Cologne/Bonn, Germany, pp 706–715

9. Erdmann L, Hilty M, Goodman J, Arnfalk P (2004) The future impact of ICTs on environmental sustainability. Available from: http://ftpjrc.es/EURdoc/eur21384en.pdf

10. Fenton N, Pfleeger SL (1997) Software metrics: a rigorous approach. Chapman & Hall, London

11. Fichter K (2001) Sustainable business strategies in the internet economy. In: Sustainability in the information society 2001, Metropolis-Veri, Marburg

12. IEEE (2004) Guide to the software engineering body of knowledge (SWEBOK)

13. Johann T, Dick M, Kern E, Naumann E (2011) Sustainable development, sustainable software, and sustainable software engineering: an integrated approach. In: International symposium on humanities, science & engineering research (SHUSER), pp 34–39

14. Mahmoud S, Ahmad I (2013) A green model for sustainable software engineering. Int J Software Eng Its Appl 7(4):55–74

15. Penzenstadler B, Bauer V, Calero C, Franch X (2012) Sustainability in software engineering: a systematic literature review. In: 16th international conference on evaluation & assessment in software engineering (EASE 2012), pp 32–41

16. Penzenstadler B, Raturi A, Richardson D, Calero C, Femmer H, Franch X (2014) Systematic mapping study on software engineering for sustainability (SE4S). In: 18th international conference on evaluation and assessment in software engineering

17. Pfleeger SL (1997) Assessing software measurement. IEEE Software March/April:25–26

18. Soto M, Ciolkowski M (2009) The QualOSS open source assessment model measuring the performance of open source communities. In 3rd international symposium on empirical software engineering and measurement. ESEM 2009, pp 498–501

Part V
Practical Issues

Chapter 12
A Decision-Making Model for Adopting Green ICT Strategies

Qing Gu, Patricia Lago, and Paolo Bozzelli

12.1 Introduction

The interest of organisations in becoming more environmentally sustainable by adopting *Green ICT solutions* is constantly growing. More and more initiatives have been proposed on energy efficiency computing, ranging from hardware to software solutions [9]. Accordingly, energy efficiency has become an important issue for the industry due to the fact that the energy consumption of ICT systems is rapidly growing and the reduction of the related energy footprint is highly demanded to realise financial savings while decreasing the environmental impact of their systems—in terms of, for example, greenhouse gas emissions, e-waste and heat generation [8].

An impediment for organisations to become more environmentally sustainable is that decision makers lack sufficient or necessary information, and hence knowledge, about which Green ICT solutions can or should be adopted. They need suitable tools to guide them in deciding where to invest. This work aims at providing such a tool, that is, addressing the following two issues so that decision makers can easily decide on the most promising Green ICT investment areas (IAs):

1. Improve the knowledge of decision makers about Green ICT investment areas.
2. Provide a tool that helps decision makers decide on Green ICT investment areas.

Q. Gu (✉)
HU University of Applied Sciences, The Netherlands
e-mail: qingbonnet@gmail.com

P. Lago • P. Bozzelli
VU University Amsterdam, Amsterdam, The Netherlands
e-mail: p.lago@vu.nl; Paolo.bozzelli@gmail.com

12.2 The Decision-Making Model

In our previous work, we have defined an ontological model for Green ICT strategies [3]. Through a number of case studies and collaborations with the industry and public administrations, we validated and refined the model [2]. Our main motivation for defining a model for Green ICT strategies was to offer a semantic structure for the essential elements that define a strategy, for the sake of, for example, reuse, comparison and selection. Accordingly, it was natural to use the elements of this model as a starting point for defining our decision-making model, as presented in this chapter.

12.2.1 The Metamodel

To address the two issues mentioned above, a decision-making model should capture the knowledge necessary to reason about where to invest and should support the reasoning process. To cover these two aspects, we carried out a systematic study of the literature (details are available in [1]) discussing two main artefacts: the *investment areas* that are already claimed in the green and sustainable ICT community and the best practices, or *strategies*, claimed by companies and organisations. Obviously, strategies act in one or multiple investment areas. Hence, the two artefacts are linked. Accordingly, our Green ICT strategy model has been extended to cover this dependency.

Furthermore, strategies naturally need economic investments and bring in economic benefits. In the same vein, they have (positive and/or negative) environmental effects. While economic impacts and environmental effects are elements of our original strategy model, they should also be linked to the investment areas where such impacts/effects may occur. In other words, decision making needs to cluster strategies, and their dependencies, within the scope of investment areas, and investment areas need to be related to the potential economic impacts and environmental effects.

12.2.2 Model Elements

Figure 12.1 shows our decision-making metamodel extending the Green ICT strategy model from our previous work.

A decision-making model would consist of the following key elements:

- **Goals** are objectives that an organisation sets itself to achieve.
- **Green investment areas** are a portion of business assets in which companies spend capitals to achieve green/sustainable goals in terms of green strategies.
- **Green strategies** are clusters of green practices that address common environmental concerns.
- **Dependencies** are defined as relations between strategies.

  For example, in the figure, green strategy a and green strategy b are linked by dependency x (i.e. one strategy may *require* the other).
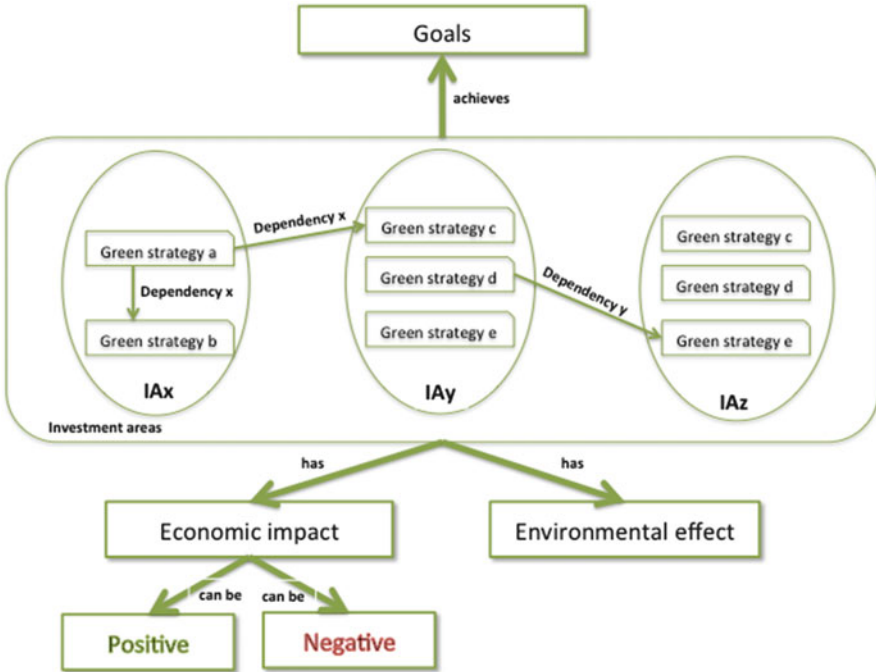
Fig. 12.1   The decision-making metamodel

- **Economic impact** captures the financial implications of adopting strategies in a selected investment area. It can be either positive or negative. A positive economic impact can be revenues, gains or returns on investment; a negative economic impact can be expenses required due to the adoption of certain strategies.
- **Environmental effects** capture the ecologic outcomes of adopting strategies in a selected investment area. Examples are the optimisation of energy consumption or the reduction of the company carbon footprint.

As shown in Fig. 12.1, a core part of the decision model is a set of investment areas (IAs). Each investment area has a number of strategies, which have dependencies with other strategies, either within the same IA or different IAs. Each strategy can achieve a number of goals, and as a result, the IA it belongs to can achieve the same goals. Similarly, each strategy can have a number of environmental and economic impacts, and as a result, the IA it belongs to can have the same environmental and economic impacts.

12.2.3 An Instantiation of the Decision-Making Model

Next to the definition of the decision-making model, the systematic literature study presented in [1] also provided quite a rich collection of specific investment areas and information characterising how to address them in various strategies.

We have used these results to instantiate the decision-making model. This instantiation is illustrated by means of two views: the goals view and the dependencies view. The *goals view* (see Fig. 12.3) illustrates which goals are achieved by the investment areas resulting from our studies. The *dependencies view* (see Fig. 12.4) shows dependencies among strategies and defines dependent and independent investment areas.

The notation used in the views is depicted in Fig. 12.2. The thickness of the arrows depicts the *relation strength* (σ) of a dependency. The thicker the arrow, the stronger is the dependency. Furthermore, the views also show which of these dependencies are *critical* or *non-critical*, respectively, depicted as a *red* dependency or a *green* dependency, following the distinction illustrated in Fig. 12.2.

Figure 12.3 depicts the *goals* view in more detail. In this view, only investment areas and their goals are shown. The goals in our study have been extracted from the Green ICT practices. In other words, investment goals correspond to a number of green practices, which are often clustered as green strategies. The thickness of lines in the view denotes the number of practices that have the same goal. In other words, the thicker the line, the higher is the chance to achieve a goal by investing on the related IA. For instance, the number of green practices that can achieve the goal *energy management and good housekeeping* is more in the investment area *IT equipment* than that of *way of working*. This means it is better to invest in *IT equipment* than *way of working* if *energy management and good housekeeping* is the main goal.
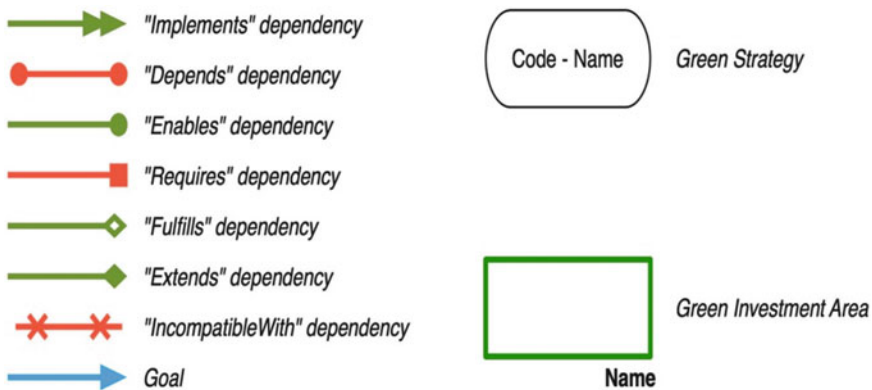
Fig. 12.2 Decision-making model—notation
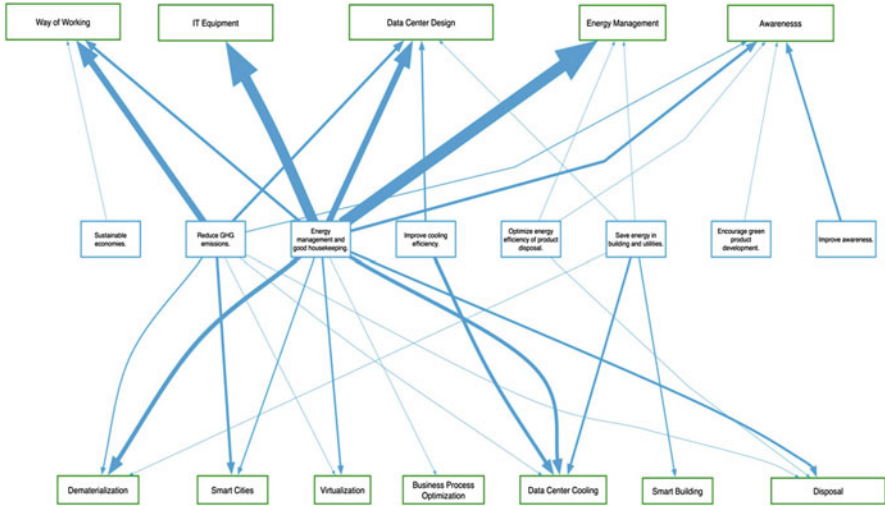
Fig. 12.3 Decision-making model—goals view
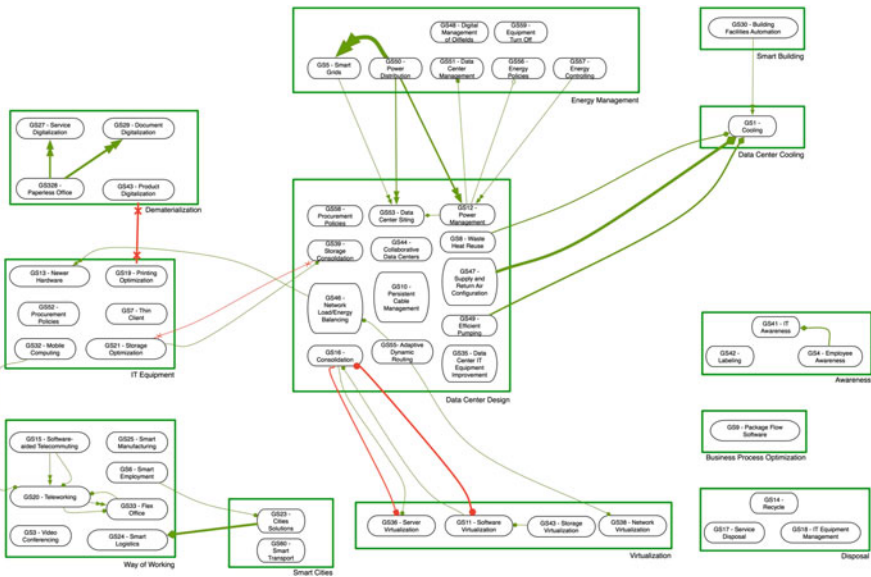
Fig. 12.4 Decision-making model—dependencies view

Figure 12.4 represents the *dependencies* view. In this view, investment areas may be divided into two main categories: the *dependent* IAs, which are investment areas containing strategies that are related to strategies of other investment areas,

and the *independent* IAs, which are made up of strategies not related to strategies of other investment areas.

For example, *data centre cooling* and *data centre design* are dependent IAs because three 'extends' dependencies occur among their strategies, while awareness is an independent IA because no dependency is defined between its strategies and strategies of any other investment area. Consequently, the model shows how dependencies occur among strategies, providing a set of arrows with different ends.

12.3 The Decision-Making Process

In this section, we illustrate how decisions on a green investment area should be taken by using our decision-making model. We were able to identify two alternative approaches that depend on what information is available for decision making.

The first approach starts with defining a set of goals that decision makers would like to achieve. Based on the goals, a list of candidate investment areas can be identified. By analysing the expected environmental effects and economic impact of these IAs, decision makers can decide on which area(s) to invest in depending on their specific requirements. We define this approach as the *goal-driven* process.

The second approach assumes that decision makers already have the knowledge of the investment areas they want to target. Starting from these investment areas, the model allows decision makers to check which goals are achieved by addressing those investment areas and to evaluate the related environmental effects and economic impact. We define this approach as the *strategy-driven* process.

12.3.1 Goal-Driven Process

First, the decision maker should gather high-level (or user-level) information like the goals one wants to achieve, the drivers motivating the decision-making process towards a certain decision and eventually the challenges, that is, the constraints in place at the company premises, and the decision maker should know upfront on how to pose a reduction in the set of possible final solutions (like initial capital availability). In other words, *goals* will be matched with the ones defined by each green practice. *Drivers* will be used as a rationale to track the motivation that led to the final decision. *Challenges* will be used to constrain the set of possible alternative decisions.

Once the above information has been defined (see Fig. 12.5), it is compared with the possible decisions (e.g. investment areas, strategies and eventual dependencies). Therefore, the investment areas and related strategies matching with the decision maker's requirements are selected.
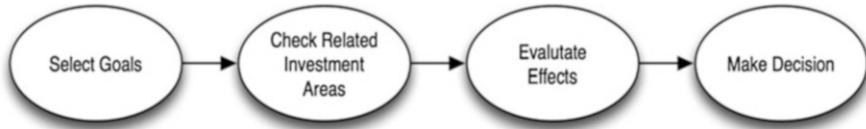
Fig. 12.5 The goal-driven decision-making process

Fig. 12.6 The strategy-driven decision-making process

Therefore, a list of economic and environmental effects for each resulting investment area is provided. As a result, the decision maker can easily decide which investment area provides the most important environmental and economic effects. To aim towards this decision, a set of metrics is provided to quantify the effects, so that decision makers can easily assess the impact of their decision.

After the effects are assessed, decision makers can easily decide on which investment area their company should focus on.

12.3.2   Strategy-Driven Process

Differently from the goal-driven process, the information gathered at the beginning of the *strategy-driven* process (see Fig. 12.6) includes investment areas rather than goals. This approach assumes that decision makers already have the knowledge about some investment areas, and they want to investigate which goals such IAs allow to achieve and which economic impact and environmental effects they bring about. Once one or more investment areas have been identified, decision makers can check which goals are achieved by applying strategies of the selected investment area(s). Furthermore, the model returns a set of environmental and economic effects and provides a set of metrics that allows decision makers to assess the environmental benefits and the economic impact.

As a result, decision makers can check if the resulting goals are compliant with their own goals. Moreover, they make the decision comparing the effect assessment of each selected investment area with their expected effects.

Finally, the investment area that matches with their goals and their expected effects should be decided.

12.4 Usage Scenarios

The following illustrates how each approach works in practice. For each approach, a usage scenario is first defined. Then each step of the approach is applied.

12.4.1 Scenario 1: Goal-Driven Process

Scenario 1: Definition—*CM is a cloud-video surveillance company that is not familiar with Green ICT practices and would like to invest its capital to get economic benefits and to improve the 'greenness' of the company by reducing the environmental impact of its infrastructure. As a starting initiative, its decision makers are investigating a way to reduce the energy consumption generated by the IT equipment of its office.*

12.4.1.1 Step 1: Select Goal

As described in Sect. 12.3.1, the first step is to select the company's goals. CM wants to reduce the energy consumed by its IT equipment, such as workstations, printers, network devices and so on. The goal that CM wants to achieve is:

G1 Energy management and good housekeeping

12.4.1.2 Step 2: Check Related Investment Areas

Starting from goal G1 takes into account different investment areas. From Fig. 12.3, we can see that the top three investment areas are *IT equipment, data centre design and energy management*. After allaying these three investment areas, the decision makers would like to further focus on *IT equipment* because *data centre design* is about the design and development of more energy-efficient and environment-friendly data centres while *energy management* is about the planning and operation of energy-related provision, which are less relevant to CM.

Therefore, decision makers investigate for strategies that allow to reduce energy consumption due to IT equipment utilisation.

IT equipment defines the following strategies:

- Mobile computing: This strategy focuses on the replacement of desktop computers with notebook computers.
- Newer hardware: This strategy includes a set of solutions to improve the energy efficiency of existing IT equipment or to create energy-efficient workplaces.

- Printing optimisation: This strategy includes a set of solutions and policies to reduce paper waste and optimise the utilisation of printers.
- Procurement policies: This strategy suggests to impose requirements on external suppliers with regard to eco-certified hardware.
- Storage optimisation: This strategy suggests solutions such as switching to offline storage or lower storage devices.
- Thin client: This strategy promotes the replacement of desktop computers with thin client computers.

12.4.1.3 Step 3: Evaluate Effects

Assess Environmental Effects
Consequently, decision makers can assess the environmental effects (in Table 12.1) and the positive economic impact (in Table 12.2).

In summary, the environmental effects generated by these strategies are:

- **IT equipment power consumption** allows to measure the power consumption generated by the utilisation of an IT device, such as storage devices, processors and so on. It is calculated as follows:

$$EIT = \sum_{i=0}^{dIT} \varepsilon i$$

where dIT is the total number of IT devices and εi is the energy consumption of the ith IT device. The result of this metrics is expressed in kilowatt (kW). The IT devices to be taken into account in this calculation are printing devices.

Table 12.1 Environmental effects of the strategies

Green strategies	Environmental effects	Metrics
Mobile computing	Reduce power consumption	IT equipment power consumption
Newer hardware	Reduce power consumption	IT equipment power consumption
Printing optimisation	Reduce power consumption, reduce paper waste, reduce cartridge-related pollution	Printouts power consumption, paper waste
Procurement policies	Reduce power consumption	IT equipment power consumption
Storage optimisation	Reduce power consumption	IT equipment power consumption
Thin client	Reduce power consumption	IT equipment power consumption

Table 12.2 Positive economic impact of the strategies

Green strategies	Environmental effects	Metrics
Mobile computing	Reduce energy costs	IT equipment energy cost
Newer hardware	Reduce energy costs	IT equipment energy cost
Printing optimisation	Reduce printing costs	Printouts energy costs, printing costs
Procurement polices	Reduce energy costs	IT equipment energy cost
Storage optimisation	Reduce energy costs	IT equipment energy cost
Thin client	Reduce energy costs	IT equipment energy cost

- Power consumption is also related to the performed printouts. Therefore, decision makers can use the **printouts power consumption**, which measures the power consumption needed to generate printouts. It is calculated as follows:

$$Eprintout = \sum_{i=0}^{p} \varepsilon i + \sum_{j=0}^{po} \varepsilon j$$

where p and po are respectively the number of printers and the number of printouts and εi and εj are respectively the energy consumed by printers and the energy consumed due to printouts.

- The reduction of cartridge-related pollution can be assessed by the **cartridge waste** metrics, which allows to measure the amount of wasted cartridge in terms of kilograms of wasted toner and/or ink. It is calculated as follows:

$$W_{cart} = \sum_{i=0}^{k} Ki$$

where k is the total number of cartridges and Ki is the toner/ink wasted by the ith cartridge. Its result is expressed in kilograms (kg).

- The reduction of the wasted paper can be easily calculated by the **paper waste** metrics, which assesses the waste of paper with respect to the number of installed printers:

$$W_{cart} = \sum_{i=0}^{p} P_i$$

where p is the number of installed printers and P_i is the amount of paper consumed by the ith printer. Its result is expressed in kilograms (kg).

Assess the Positive Economic Impact

In turn, the positive economic impacts generated by these strategies are:

- **IT equipment energy cost** expresses the cost of energy consumed by the IT equipment of a data centre or an office:

$$C_{eIT} = \sum_{i=0}^{d} c_\varepsilon \varepsilon_i$$

where c_ε is the fixed electricity fare, εi is the energy consumed by the ith IT device and d is the number of IT devices in the facility. The IT devices, in this case, are printers in the office.

- The **printing costs** evaluate the cost incurred by the printing tasks. It is calculated with respect to the used amount of paper and cartridge only, as follows:

$$C_{print} = \left(U_{paper} C_{paper}\right) + \left(U_{cart} C_{cart}\right)$$

where U_{paper} and U_{cart} are respectively the amount of used paper and cartridge, while C_{paper} and C_{cart} are respectively the prices of paper and cartridge.

- The **printouts energy cost** metrics calculates the cost of the energy consumed due to printouts. It is calculated as follows:

$$C_{printout} = c_\varepsilon \left(\sum_{i=0}^{p} \varepsilon i + \sum_{j=0}^{po} \varepsilon j \right)$$

where c_ε is the fixed electricity fare, p and po are respectively the number of printers and the number of printouts and εi and εj are respectively the energy consumed by printers and the energy consumed due to printouts.

Assess the Negative Economic Impact
Table 12.3 shows the negative economic impact of each strategy, with the metrics for calculating the impact. Obviously, the strategies *printing optimisation and procurement policies* require the least expenses and investments.

In particular:

- The IT equipment procurement cost metrics performs the calculation as follows:

$$C_{purchase\,IT} = \sum_{i=0}^{d} C_i$$

where C_i is the purchase cost of the ith IT device and d is the number of IT devices in the facility. The number of IT devices is measured, since it suffices to count the IT devices to be replaced or the IT devices to be newly purchased. The cost of IT devices is, for instance, provided by IT reseller catalogues.

Further, the costs of printing equipment depend on the recycling process, the purchase of eco-labelled cartridge and the purchase of multifunctional printers. Consequently:

Table 12.3 Negative economic impact of the strategies

Green strategies	Negative impact	Metrics
Mobile computing	Purchase of new hardware	IT equipment procurement cost
Newer hardware	Purchase of new hardware	IT equipment procurement cost
Printing optimisation	Purchase of printing accessories or new printers	Paper recycling costs, cartridge cost, IT equipment procurement cost
Procurement policies	No	–
Storage optimisation	Purchase of new hardware	IT equipment procurement cost
Thin client	Purchase of new hardware	IT equipment procurement cost

- **The paper recycling cost** metrics allows to calculate the costs incurred by the recycling process as follows:

$$C_{recyle} = p_{crec}$$

where p is the amount of paper to be recycled and $crec$ is the recycling fare. The amount of paper is expressed in kilograms, and it is estimated before delivering the whole amount of paper to the recycling service provider. The recycling fare is set by the recycling service provider, and it is expressed in dollars per kilogram ($/kg).

- The cost of the eco-labelled cartridge is calculated using the **cartridge cost** metrics, which performs the calculation as follows:

$$C_{cart} = \sum_{i=0}^{cart} C_i$$

where C_i is the cost of the ith cartridge and $cart$ is the total amount of purchased cartridges. The cost of the cartridge is provided by the printing equipment reseller. The results are expressed in dollars ($).

- The cost of the multifunctional printers is calculated with the **IT equipment procurement** cost metrics, introduced above. It performs the calculation as follows:

$$C_{purchase\,IT} = \sum_{i=0}^{d} C_i$$

where C_i is the purchase cost of the ith IT device and d is the number of IT devices in the facility. In this case, the IT device is a multifunctional printer. The number of printers is measured, since it suffices to count the printers to be replaced or the

printers to be newly purchased. The cost of printers may be provided by IT reseller catalogues.

12.4.1.4   Step 4: Make a Decision

Once the effects have been assessed, decision makers can decide on which invest-ment area to focus their attention on and which strategies they should apply. Due to the low negative economic impact and promising savings as well as environmental effects, decision makers might choose:

- Decided investment area: **IT equipment**
- Decided strategies: **printing optimisation and procurement policies**

12.4.2   Scenario 2: Strategy-Based Process

Scenario 2: Definition—*The Environmental Department of the Turkish govern-ment has been proposed to provide funding for a green initiative, organised by a non-profit association that wants to encourage the department staff to behave in an environment-friendly way. Decision makers of the department are fully acquainted with the initiative, but they would like to know more about its environmental effects. Since it is an initiative organised by a non-profit association, revenues of the initiative are less important, but expenses should be estimated in advance.*

12.4.2.1   Step 1: Select Investment Areas

As described in Sect. 12.3.2, the first step is to identify the starting investment areas. Decision makers have the knowledge about sensitisation of employees regarding environment-friendly behaviour. For this reason, they select the following invest-ment area:

IA1 *Awareness*

The awareness investment area defines the following three strategies:

- GS41 IT awareness: This strategy incentivises the promotion of green awareness by means of software solutions, such as smart metering and sensitisation by sending messages to customers, employees or users.
- GS42 labelling: This strategy is designed to show to consumers the total amount of GHG emissions expected to be produced throughout the product life cycle.
- GS4 *employee awareness*: This strategy is about the promotion of awareness campaigns and the development of green company policies.

In summary, decision makers take into account only the employee awareness strategy, since it is the only one designed to encourage the staff of an organisation to be aware of their energy consumption and take environment-friendly actions.

12.4.2.2   Step 2: Check Achieved Goals

From Fig. 12.3, we can see that the investment area *awareness* would achieve the following goals, sorted by the related strength:

- Improve awareness
- Energy management and good housekeeping
- Reduce GHG emissions
- Optimise energy efficiency of product proposal
- Encourage green product development

The top three goals perfectly match with the goals of the Environmental Department of the Turkish government, and therefore, it is confirmed that the investment area selected is promising.

12.4.2.3   Step 3: Evaluate Effects

To assess the effects of the employee awareness strategy, decision makers can check the list of linked environmental and economic effects (for details, we refer the reader to Appendixes in [1]). Decision makers focus on environmental effects and the negative economic impact.

Assess Environmental Effects
The only environmental effect generated by the employee awareness strategy is the following: increase the environmental awareness of the employees within the company or the organisation.

To assess this environmental effect, decision makers can use the following metrics:

- The **employee environmental awareness coverage** (%), which allows to measure the potential amount of employees that are affected or sensitised by awareness-oriented practices. It is measured as follows:

$$EACemployees = \frac{e_c}{e}$$

where e_c is the number of involved employees and e is the total number of employees. This metrics returns a percentual value (%).

- The **message-bounded employee environmental awareness coverage** (%), which is a particularisation of the *employee environmental awareness coverage* metrics, because it is calculated with respect to the number of messages that are

sent to involve employees in the green awareness initiative. It is measured as follows:

$$MEAC_{employees} = \frac{e_c}{mE}$$

where e_c is the number of involved employees, E is the total number of employees and m is the total number of employees. This metrics returns a percentual value (%).

Assess the Negative Economic Impact

The only expense to support the employee awareness strategy will be concerning the rewards that have to be paid to the most active and environment-friendly employees. Therefore, the only negative economic impact will be the following:

- *Extra costs are needed to pay rewards.*
  To quantify and calculate this cost, decision makers can use the following metrics:
- The **reward payment**, which allows to evaluate how much should be spent to reward employees for their awareness about green initiatives. It is calculated as follows:

$$C_{rew} = \sum_{i=0}^{r} C_i$$

where r is the number of rewards and C_i is the amount of money spent for the ith reward.

12.4.2.4 Step 5: Make a Decision

Assuming that the rewards for the best environment-friendly employees should be limited up to some thousands of dollars and that only a limited number of employees (e.g. from 1 to 3) should be rewarded, decision makers can decide to invest in the awareness investment area and, therefore, to support the expenses concerning the green initiative.

In summary, decision makers make the following decisions:

- Decided investment area: a**wareness**
- Decided strategies: **employee awareness**

12.5 Conclusions

In this work, we have presented and instantiated a decision-making model and two alternative decision-making approaches (goal driven and strategy driven) addressing the issues claimed in Sect. 12.1, namely, how to improve the knowledge of decision makers about Green ICT investment areas and how to provide a tool to guide the decision-making process. By means of two usage scenarios, we illustrated the usage of the model as well as the two decision-making approaches.

While promising, major research is required to feed the decision-making model with knowledge about where to invest, how to invest and related implications whenever companies and organisations want to *go green*. This should provide knowledge in rendering energy-aware both the ICT solutions themselves (e.g. [4, 6, 7]) and the exploitation of ICT solutions at the service of energy [5].

References

1. Bozzelli P (2013) A decision-making model for Green IT investment areas. Master's thesis, VU University Amsterdam
2. Gu Q, Lago P, Muccini H, Potenza S (2013) A categorization of green practices used by Dutch data centers. In: 3rd international conference on sustainable energy information technology, 7. Elsevier
3. Gu Q, Lago P, Potenza S (2012) Aligning economic impact with environmental benefits: a green strategy model. In: Workshop on green and sustainable software (GREENS), ICSE Companion. IEEE Computer Society, pp 62–68
4. Gu Q, Lago P, Potenza S (2013) Delegating data management to the cloud: a case study in a telecommunication company. In: International symposium on the maintenance and evolution of service-oriented and cloud-based systems (MESOCA), n 7. IEEE Computer Society, pp 56–63
5. Hilty L, Lohmann W, Huang EM (2011) Sustainability and ICT – an overview of the field. Notizie di Politeia 28(104):13–28
6. Procaccianti G, Bevini S, Lago P (2013) Energy efficiency in cloud software architectures. In: Environmental informatics and industrial ecology, Proceedings of the EnviroInfo
7. Procaccianti G, Lago P, Lewis GA (2014) Green architectural tactics for the cloud. In: Working IEEE/IFIP conference on software architecture. IEEE Computer Society
8. Velte T, Velte A, Elsenpeter R (2008) Green IT: reduce your information system's environmental impact while adding to the bottom line. McGraw-Hill, New York, URL http://books.google.nl/books?id=xPQZqKrJN7oC
9. Wang J, Feng L, Xue W, Song Z (2011) A survey on energy-efficient data management. SIGMOD Rec 40(2):17–23. doi:10.1145/2034863.2034867, URL http://doi.acm.org/10.1145/2034863.2034867

Chapter 13
Participation and Open Innovation for Sustainable Software Engineering

Martin Mahaux and Annick Castiaux

13.1 Introduction

For decades, economists have put innovation at the core of economic growth. In their classical conception, sustainability is the capability to maintain and develop the level of economic performance of the society. In this regard, entrepreneurs who creatively change the rules of the economic game by proposing innovative technologies and businesses are key actors that support growth [45]. The ICT sector has obviously been a key provider of this kind of innovation in the past years, the new information and communication technologies being at the foundations of a transition towards a post-industrial economy.

More recently, the acceptation of 'sustainability' has changed in order to take into account the increasingly important issues of sustainable development. In addition to their economic challenge, firms have to deal with growing environmental and social requirements from their stakeholders, that is, actors that affect or are affected by the actions of the firm. From the firm's side, environmental and social requirements can be seen as additional constraints to the firm's innovation space, limiting its opportunities to develop and grow, that is, to reach its own sustainability. In contrast, it can also be seen as an enabler for differentiation and consequently as a competitive advantage.

In this context, innovation processes should be thought differently, as the complexity of sustainability issues asks for systemic approaches, going beyond the borders of a given organisation and integrating economic, social and environmental objectives that are very often antagonist. The purpose of this chapter is twofold: first, we want to demonstrate the importance of participation and openness in innovation processes integrating sustainable development, illustrating it with a prominent example in the software domain—open source software (OSS); second,

M. Mahaux (✉) • A. Castiaux
University of Namur, Namur, Belgium
e-mail: martin.mahaux@unamur.be; annick.castiaux@unamur.be

we want to propose elements of a methodological framework supporting participation in the context of software requirements engineering.

13.2 Participation, Openness and Sustainable Innovation

13.2.1 Why Is the Innovation Process More and More Open?

13.2.1.1 Openness and Participation to Enlarge Innovation Sources

Traditionally, the innovation process was managed in a closed manner, inside the firm's borders. We see two reasons in this closure. First, firms wanted (and still want) to protect the competitive advantage they build through innovation, as interactions with other actors can lead to knowledge spillovers and intellectual property losses. Second, technological firms based their development on knowledge and experience accumulated through the years, in which they found the main sources of innovation. They did not see opportunities outside their usual field. This behaviour is often called the 'not-invented-here' effect. Even inside the firm, the involvement in the innovation process was very limited, as it was the prerogative of some categories of personnel, mainly researchers developing new technologies in the R&D department or engineers conceiving and improving processes in plants.

Progressively, things changed as firms faced highly publicised failures due to their lack of openness in their innovation process. A well-known example is the case of Xerox in the late 1970s. Xerox launched a research centre—PARC, Palo Alto Research Center—in 1970, where excellent and creative computer engineers developed inventions that were never commercialised by Xerox. Instead, these ideas served other companies, such as Apple. Xerox exited the computer market in 1975, refocusing on its core business: printing [17]. One of the reasons of this failure is the lack of integration of the R&D function with other functions of the company, which did not allow Xerox to transform creativity into innovation. More recently, firms realised that every person in the company is able to propose interesting ideas that can lead to innovation. A company like Renault, for instance, challenges its personnel on tricky technological problems, associating human resources in the early stage of the innovation process and using such participatory approach as a motivation factor [9]. Such a participation in the innovation process can even go beyond the borders of the firm, including customers, users or citizens. Several motivations can explain this trend to open the innovation process:

- Thinking out of the box: Firms—especially if they are established—are influenced by their culture, knowledge and competences. Opening innovation to external sources of ideas helps them to think out of the box. This is the principle of inbound open innovation [10].

- Practising cross-fertilisation: Associating multiple profiles in idea generation allows to explore unexpected areas and to combine complementary knowledge backgrounds [6]. In increasingly complex products, this combination of multiple knowledge sources has become mandatory.
- Combining problem and solution focus: Faced with the same problem, a user and a supplier think differently. The user wants to solve a problem (whatever the solution), while a supplier wants to sell a solution (as close as possible from his expertise). Combining these problem and solution orientations seems to lead to better innovations, satisfying both parties [23].
- Assessing technology: Collaborating with users helps to assess technology on various criteria (quality, user-friendliness, design, etc.) and avoids market failures.
- Benefiting from innovative attitudes: A lot of people are innovative and like to contribute to a creative process. They value this participation in itself, as studies in the open source community have shown [22]. Integrating them in the innovation can be a factor of motivation for human resources as well as a method of loyalty development for customers.
- Propagating standards: When developing breakthrough technologies, there is a strong risk that competitors develop similar technologies concurrently and that one of those competitive technologies become the dominant design (the standard), making all other technological designs irrelevant. Cooperation with other actors (users, partners, suppliers) is a way to reduce such a risk [1].

For various reasons that we develop in this chapter, sustainability will greatly benefit from these aspects of open innovation.

13.2.1.2 Openness and Participation to Involve Stakeholders in the Strategy

Freeman proposes an alternative view of strategic management [14]. Beyond a performance path oriented mainly to the satisfaction of shareholders, he demonstrates how other parties with whom the firm is in relation have to be taken into account to optimise the chances of success on a marketplace. At a moderate level, the firm can consider its stakeholders in an instrumental way in order to enhance its performance. Especially in uncertain periods and environments, taking into account the viewpoints of key stakeholders opens alternative and more informed strategic paths. However, Freeman recommends a more radical change: strategic management should integrate stakeholders intrinsically through a strong partnership. In this view, the interests of stakeholders are taken into account in decision making even before strategic decisions, as ethical foundations of the strategy. This is the key principle of corporate social responsibility (CSR).

Such a point of view questions the classical definition of performance. An intrinsic partnership with stakeholders considers that satisfying stakeholders' interests must be taken into account when evaluating the firm's performance [13]. This

means that performance is not only economic but has to integrate dimensions that are valued by stakeholders, in particular social and environmental dimensions. In this sense, participation is strongly linked with sustainability.

13.2.2 Participation, Openness and Sustainability: Is It Possible to Innovate for Sustainability in a Closed View?

Sustainable development issues can only be considered and tackled using a systemic approach of innovation. As a matter of fact, such issues necessarily involve multiple actors that have an impact on their mutual performances because of their strong interactions, both on environmental and social matters. In this section, we demonstrate the necessity to adopt a systemic and open view, beyond the firm's boundaries. We consider two levels of analysis: the global economy and the firm in its value chain.

13.2.2.1 Sustainability and Innovation at the Global Economy Level

Sonntag shows that environmental and social challenges require a global change in consumption modes and technology developments, as both co-evolve [46]. Traditionally, firms try to continuously increase their economic performance and to grow. To meet growth, firms innovate, developing technologies allowing them to produce more at lower costs and with shorter life cycles, in order to enhance their performance, maintain their leadership and ensure their survival, that is, their sustainability. Once they are engaged in a technological path, the cumulative nature of the innovation process and the importance of their investments in a set of dedicated capabilities will affect their future strategic choices and, by 'ricochet', have an impact on the whole industry (by imitation) and on their markets (which become used to some consumption behaviours). This lock-in effect [4] has led to dominant technological trajectories: since the beginnings of mass production technologies after World War II, this trend has increased, notably thanks to information technologies. Cheaper goods and reduced production cycles have changed our consumption habits, which in turn have changed firms that must meet consumer requirements to survive.

Moreover, competitiveness policies developed by governments are based on advanced manufacturing technologies. Even current sustainable policies are embedded in this paradigm: they focus on limiting environmental damages and integrating sustainability criteria into organisation decision making, trying to demonstrate that clean production and eco-efficiency lead, in the end, to economies for the firm. However they do not question the acceleration of consumption and the subsequent low-cost manufacturing. Of course, this race in production and

consumption cycles is damaging for macro-sustainability, as it increases aggregated use of natural resources [21].

Alternative paradigms are possible. For example, the extension of product durability could lead to a reduction of the global consumption of resources. However, this requires a change in business models, where value is created and captured in a different way. In particular, services with added value and co-creation with customers offer alternative opportunities of revenue. These two possibilities ask for collaborative (open) approaches beyond the traditional manufacturer–consumer unidirectional relationship.

13.2.2.2   Sustainability at the Value Chain Level

In a seminal paper, Hall demonstrates the systemic nature of environmental innovations that involve not only the firm but its whole supply chain, as well as other actors affected by its environmental impacts [19]. He underlines that innovations developed to take into account environmental considerations—that he calls 'eco-innovations'—cannot easily emerge spontaneously. He proposes a systemic view putting at stake different agents, inside and outside a firm's supply chain, who influence each other towards the emergence, the development and the adoption of such eco-innovations. He identifies two types of pressures favouring eco-innovations, as illustrated in Fig. 13.1.

First, the vertical pressure goes backwards through the supply chain, from the end customers up to the producers of raw materials. The importance of this vertical pressure depends on two major factors: the power of the agent on upstream agents in the chain and the competences of this agent concerning the technical knowledge at stake in the choice between alternatives. If an agent has such a power and the asymmetry of information between this agent and its suppliers is sufficiently low, this leads to interfirm innovations and systemic improvements. However, this vertical pressure is not sufficient to foster eco-innovation. As a matter of fact, externalities, should they be positive or negative, are not taken into account by the actors of a given supply chain. To favour the integration of externalities in the eco-innovation processes led by firms, an additional pressure has to play its role: the
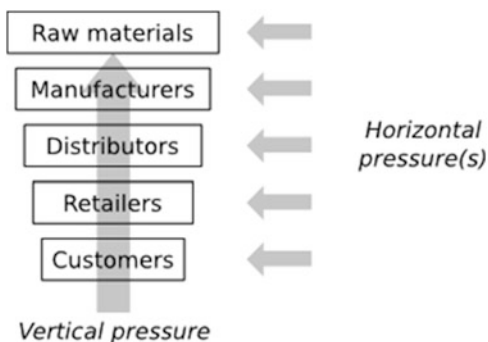
Fig. 13.1 Pressures favouring the emergence of eco-innovations (adapted from [19])

environmental pressure, represented in Fig. 13.1 as the horizontal pressure. Exerted by organisations, which are external to the considered supply chain (e.g. governmental authorities, NGOs), this pressure is motivated by externalities, as the environmental impact of industrial activities. It takes the form of regulations, rules, quality labels or reputation threats. This horizontal pressure is local, as it impacts a given firm or a given group of competitors. Along the whole supply chain, levels of pressure are variable, which has only a limited impact on the whole supply chain. These two pressures, horizontal and vertical, are complementary conditions to the successful emergence of eco-innovations [19].

13.2.3 *Open and Participatory Innovation to Integrate Stakeholders and Reach Sustainability*

The preceding discussion leads to the conclusion that innovations favourable to macro-sustainability should integrate as much as possible the firm's stakeholders. Such participative approaches of innovation should involve internal (employees) and external (users, suppliers) actors in a co-innovation process. This helps the firm to integrate key elements in new technologies and services, favouring the adoption of the innovations. So, not only does participatory innovation meet the stakeholder integration necessary for macro-sustainability, but it is also favourable to micro-sustainability, providing the firm with competitive advantages that it obtains from a better understanding of customers' needs and suppliers' possibilities. Moreover, associating the employees in the innovation process has also been demonstrated as a positive factor for the intrinsic motivation of employees, which benefits both the social sustainability and the efficiency of the firm.

Open innovation [10], which is presented as a shift of paradigm in the way innovation is practised by firms, can also favour macro-sustainability. On the one hand, inbound open innovation, which consists in seeking sources of innovation outside the firm, should strengthen the vertical pressure given by Hall and raise awareness of cultural changes, for instance, in consumption patterns, to begin a virtuous cycle between sustainable consumption and production modes. On the other hand, outbound open innovation should favour the propagation of good innovation practices and sustainable technologies. Once more, however, political intervention is required to support alternative performance values. If open innovation can help to meet the conditions of macro-sustainability, does it favour micro-sustainability? Firms listening to their environment (thus using inbound open innovation) avoid to be surprised by new trends they would not have anticipated. Moreover, outbound open innovation helps the firm to focus on core competencies while leaving space for curiosity and creativity, only keeping the innovative results that can reinforce the firm's competitive advantages. In this perspective also, open innovation is favourable to micro-sustainability.

13.3   Understanding Participation and Co-creative Processes in Socio-technical Systems Design

When building a socio-technical system and in particular software-intensive systems, requirements engineering (RE) refers to the activities through which business problems and solutions are defined, in contrast to implementation activities. While other activities of the software development process may be more of a specialist affair, the RE phase relies heavily on every stakeholder, in particular the user. Indeed, the requirements discovery process can be seen as a process of collaborative creativity [29]. Consequently, participation is always present, in a more or less intense way. The question is thus not whether or not to use participatory processes, but which ones and to which participation degree. This section explores the concept of participation in more detail.

13.3.1   Governance for Participation

Citizen participation has been classified by Arnstein [3] in a number of levels, from pseudo-participation (non-participation) to full citizen control, in raising the level of power given to the citizen. Thirty-five years and several criticising papers later, researchers have augmented this vision with a more subtle way of assessing participation levels. The quality and methods of involvement as well as the type of people involved have notably been lacking in Arnstein's model [16, 50]. These more recent works indicate that there is no simple way to assess the level of participation. They, however, can inspire a simplified multi-ladder framework for describing various participative degrees in the RE process, as illustrated in Fig. 13.2. It focuses on *who* will participate (how many different stake areas and how many representatives from each category will be represented in the various discussions), *how* participation will be facilitated (how frequently, how directly, how qualitatively), *how far* participants will be allowed to influence decisions (from bare consultation to real decision making—this corresponds to Arnstein's ladder) and *on what* matter(s) participation will be allowed (low-level details, high-level objectives or everything that is in between). This framework allows one to assess a

Fig. 13.2 Ladders of participation in RE

level of participation for every RE process, so we now understand better what a participative RE process may mean.

We may thus define 'participative RE processes' as those where stakeholders from various areas of stakes are expected to actively work together to discover, validate, document or analyse requirements elements. A process is considered more participative if more people from more stake areas are involved; if they interact more frequently, in a richer and more direct way with each other; if their potential influence on decisions is bigger; and if they are allowed to influence on more topics.

We refer to *governance* as the way the participative process is managed to reach the desired participation level. The first condition for a successful participative process is an adequate stakeholder selection. Fung describes the five main stakeholder selection techniques in citizen participation [16]:

- Fully open self-selection (anyone may register)
- Self-selection augmented by selective recruitment (open but some less represented areas are recruited)
- Random selection
- Lay stakeholders (unpaid representatives, e.g. neighbourhood association representatives)
- Professional stakeholders

The fully open selection method is known to over-represent some stakeholders; augmentation by selective recruitment is supposed to lower the bias. Using unpaid representatives enables to reach people that have an interesting mix of strong visions, expertise and field experience, while professional stakeholders bring even more expertise and visions—at the risk of being more partial. The parallel with RE is only limited: it applies only to mass-market-driven products, that is, those that target an important number of buyers. The way stakeholders are involved in such projects has evolved strongly. Many developers now tap in the wisdom of the crowd, using information networks to involve a large amount of users and gather new requirements, in the form of feedback on existing products. Some more sophisticated mechanisms enable users to request and vote for specific features to be implemented. Open source communities are the most radical in this direction, offering full-fledged forums and voting schemes to discuss feature requests. Studies have shown that communities at large drive the future of products [37]. The structures implementing the 'fully open' selection pattern have to live with its main drawback: a highly biased population participates, mostly made of tech-savvy people. The perceived lack of usability of open source solutions for non-tech-savvy people might come from this bias. Commercial development may enjoy more professional market survey practices, including a mix of the four selection techniques, which is probably a more profitable cocktail if managed adequately.

Whether or not the product is a mass-market one, there is a need for the participative process to tap on a variety of stake areas. Collaborative creative processes gain richness when the creating team is sufficiently diverse. More participative processes will include and involve as early as possible representatives from the users, customers, sponsors, developers, regulators, experts, etc.

Once the right people are selected, the frequency and richness of the interaction between them will be crucial in shaping the participative process. Do we just ask people to send a comment once via a software distribution platform (like Google Play or Apple's App Store)? Do we allow discussion on a forum in the way open source does? Or do we invite them to take part in creative requirements workshops? These are very different ways to interact. From expressing preferences, through developing them, to building together a consensual solution, there is a wide range of possible participation techniques. More participative processes will tend to support the latter. In this case, political literature will usually talk of negotiation or consensus seeking [16]. In our opinion, however, there is a large spectrum between pure negotiation and creative consensus making. In the first end of this spectrum, the conflicting views are competing, while in the second end, they are enriching each other. Instead of reaching a deceiving middle-point compromise after negotiation, co-creation encourages to use conflicting views as the source for creativity, exploring new dimensions to reach a consensus that is seen by all as better than what each had originally in mind. While crucial in order to yield the best fruits out of participation, switching from negotiation to co-creation is however a step that is not easy to take, and we will suggest a methodological framework for this in Sect. 13.5.

Finally, we can assess the influence of the participative process on the final decisions. More participative processes allow more power to the group, on a broader scope of decisions.

13.3.2 Promises of and Obstacles to Co-creative Processes in Requirements Engineering

Beyond the impact of participatory processes at macro- and microeconomical scales described in Sect. 13.2, the importance of collaborative creativity in RE [29] leads us to think that more participatory approaches to RE will indeed reach better results, more efficiently, as they suit better the natural participative nature of the problem at hand. But further, what is 'better results'? In our context, we are probably interested in 'more sustainable systems', that is, systems that perform better in terms of economical, ecological and social values. First things first: as far as a system that does not bring value to its users is worth trashing, we may confidently conclude that 'sustainable systems' are at least systems that suit the needs of users, which is the core objective of RE as a discipline. Beyond that, sustainable systems will have to take ecological and social requirements into account.

Studies in several domains lead us to think that, indeed, participation has a strong potential to provide more sustainable systems. However, this will not be automatic, and many obstacles have to be mitigated. We develop below an initial argumentation, building on studies in the social development and journalism domains.

13.3.2.1 Promises

Firstly, it is worth noting that in social development projects, participation is seen as being a component of sustainable development [35]. In this discipline, participation has long been seen as a must, not only to ensure acceptance but also as a way to empower people, consequently leading to a better control of their own lives in the long term [28]. A similar pattern exists when, in a company, participative innovation is fostered, leading to empowerment and more satisfaction [53]. In short, we can say that successful participation will lead to long-term empowerment, which in turn supports social sustainability, as having control of one's life is a crucial human need and right. Empowerment also means power to act and think by oneself, thus retrieving an active role in the society, potentially leading to more responsible behaviours. The power to create is seen by many as a core human asset. Creating in groups rebuilds links in a society that is missing them more and more. It has the potential to revive the feeling that we are interconnected, and all depend on each other, on this single planet, which is likely to reinforce responsible behaviours too.

Empowering people also relates to democracy. Many experts estimate that democracy urgently needs to be revived in order to offer the world a chance to adequately tackle the century's challenges, and many lean towards participative processes as the best chance in this domain: participation efforts in this sector have shown that sustainable development indeed needs participation [44]. Both in Africa and in Belgium, participative governance is shown to work well and to relate to sustainable decision making from involved citizens [28, 35]. Those works also indicate that projects using successful participation have a better acceptance rate, last longer and consequently have a stronger impact. This relates to sustainability in that projects that are not accepted or short-lived represent an important waste of resources and energy.

Further, cases illustrate how participation allows deconstruction and creative reconstruction of problem frames, by allowing circulation of the problem in various dimensions and spaces. Participation helps to open up the solution space and let us come to richer solutions that cope with more objectives and constraints, including sustainability constraints that were sometimes out of the initial scope, before the participation [35]. Doelle and Sinclair also advocate that a consensus-making form of participation will be more efficient and lead to more sustainable outcomes than an a posteriori assessment one [11]. Journalists also point to promises from participation, imagining the discipline as a conversation rather than broadcasting [2], nurturing better democracy. People are ready to participatively fund independent journals and are shown to do so for contributing to common good and social change [18].

13.3.2.2 Obstacles

It is clear that participation is not automatically a success. Real participative processes may be rich but always cost time and are not always possible or even desirable. The first problem with participative processes concerns the possibility for the participation to be controlled by a certain type of people, more skilled or culturally stronger. The idea that participation is a discussion forum without rules must thus be rejected, at the risk of seeing the stronger impose its opinions by influencing others in a way or another [35]. Lyons et al. indicate experiences where participation failed for such reasons [28].

But even if there is no such strong person or group, the risk of seeing participants fighting for their own personal, local, short-term interests exists. In these circumstances, the process will be, at best, inefficient. Lyons et al. also show such examples in African development projects: where no democratic and transparent cultures and infrastructures were in place around the project, the participation failed to bear its promises [28]. Participative processes are indeed vulnerable to malfunctioning environments. There is sometimes a huge work to accomplish before the environment is ready for participation. The failure mentioned above draws this conclusion, along with failures in participative journalism: Goode indicates that *people and systems, including software running at major crowd-media platforms, have to make their way* (in order to yield results from participative processes) [18]. In other words, we are not there yet, and the road is not as easy as some would like to let it think. In many places, we live and work in a culture that is focused on the individual, on fighting for one's own ideas, on competition. In particular, in the innovation sphere, competitiveness is still seen as the main objective of most innovation initiatives. Education too is still mostly centred on the individual and on pragmatic skills, rather than on relational skills. Software education, in particular, is still focusing on individual and technical skills, underexploiting the softer and more relational aspects of the discipline, despite some attempts to tackle these problems [40]. Providing training and education in requirements engineering and focusing on human and participative aspects, it is the author's experience that practitioners and students alike have an important gap to bridge in order to be able to exploit the full power of collaboration.

Finally, we underline that if participation is to lead to more sustainability, beyond the positive social aspect of empowering people, we need participants who care about sustainability. The various cases of urban development in Belgium showed that involving a greater public did bring environmental concerns to the front, while experts had neglected some of these aspects. In general, participation relies on strong stakeholder analysis, which we have been used to in requirements engineering. In the case of sustainable systems design, we need to ensure that some stakeholders will stand up for sustainability concerns [41].

In short, we can say that real collaboration is not the norm and that it represents an important paradigm shift at various levels. Participation is a fragile process that

needs to be protected and supported, requiring new infrastructures, mindsets and skills.

13.3.3 A New Role for Experts

A common concern is about the quality of work that can be achieved by amateurs participating and the place that professionals, or experts, should take in the process. The example of online medical forums (a form of participative medicine) is probably making this problem clear. Bypassing doctors and mutually diagnosing and medicating each other online can potentially be extremely dangerous. Similarly, information relayed by microblogging platforms (a form of participative journalism), escaping the journalistic validation, has the potential to convey wrong information at a dangerously rapid rate [18]. Crowdsourcing cars (participative engineering) is nice but should not mean forgetting centuries of engineering to reinvent the wheel.

Consequently, participation must not be seen as excluding professionals and experts from the process, as it tends to be done in the examples above. Instead, we have to reinvent the relation between experts and the public/users/audience. This relation cannot be unidirectional, from top to bottom anymore, but places experts and professionals at the centre of a discussion: they have to act as facilitators and consultants. For example, a participative policy-making effort led in Belgium had invited experts from the academy and industry to present the state of the art and answer questions in the various areas of expertise that the 1,000 selected citizens would discuss. The process was managed professionally and employed trained facilitators [54]. So neither do we reject experts nor do we give them the power to decide on their own: we use them as facilitators and consultants. Journalists cross-checking and validating Twitter feeds to provide accurate uncensored information are another example of a new relation between experts and the public.

13.4 Case Illustration: Open Source Software

13.4.1 Open Source Software: Open and Participative

Open source software (OSS) is a paramount example of open and participative efforts for developing software-based systems. After exploring the notion of open source in terms of participation and openness, we will discuss the effect that this movement has had on innovation and sustainability.

13.4.1.1   Openness

OSS is, of course, by definition open: it allows software to be freely used, modified and shared. There are, however, various levels of openness, as indicated by the many flavours of open source licences. The open source initiative publishes the definition of what constitutes open source and validates open source licences as compliant to their definition [56]. This definition is made of ten items that are directed at ensuring that OSS plays its role in the collaborative evolution of software, involving as many participants as possible, including commercial ones.

13.4.1.2   Participation

As we have mentioned above, participation in OSS mostly follows the fully open self-registration paradigm. Various kinds of stake areas are represented. Coders are the most prominent group, but non-coders also participate, mostly through writing the requirements. In an attempt to understand user participation in writing the requirements for OSS, Noll traced features from first mention to release. The results confirm the importance of user participation in open source projects [37]. There is a public, open role in setting the agenda for OSS, whereas in closed software this was not the case: profit-related objectives would always be the main driver.

The fact that developers with all sorts of skills, origin, background and motivation co-construct software is core to open source. OSS licences ensure that the openness in that regard is total. The only restriction is then sociocultural: only a fragment of the population is skilled and equipped for participation. However, the barrier is lowering quickly, as equipment gets cheaper (thanks to open source hardware and software initiatives) and software education and resources get available on the Internet (thanks to open and/or participative education initiatives).

The richness of interactions of collaborators in OSS projects is diverse. It is rarely direct; most interactions happen via online interfaces that have various levels of richness in the discussions they can support. It goes from classical forums through ticketing systems (e.g. [55]) until clustering and voting mechanisms or specific distributed requirements gathering platforms (e.g. [34]). Online discussion can never support a consensus process as well as a well-facilitated workshop can. But in the context of massively distributed RE, this is probably as good as it can be.

Concerning the power that is given to participants, the term *forking* denotes the possibility for anyone to make a copy of a project and continue to develop it in parallel with the original project. Studies have shown that this pattern obliges project leaders to listen to their community, giving them a formal power that avoids dictatorial situations to the benefit of more participative situations [38]. This also allows for a situation where the scope of participation is total: participants may decide from high-level strategic options until code line level details, just by forking if they do not agree with the current direction.

13.4.2   Open Source and Innovation

There is a common critique of open source projects: they are thought to merely copy other software, making it free but of lesser quality. A typical example of this is the famous OpenOffice project, based on the even more famous Office suite from Microsoft. However, in 2007 already, Ebert [12] summarised 3 years of 'open source' column in *IEEE Software* with an article 'Open Source Drives Innovation'. He underlined that open source components, such as operating systems, databases, application servers and Web servers, are at the heart of an immense amount of innovative systems. But more, open source has brought innovation in the software world by changing the way we develop systems, augmenting the quality, revolutionising software architecture, supporting standards, re-establishing fair competition and reinventing business models.

Another critique to open source is that it facilitates imitation and thus results in lower value for the inventors. In 2008, Pollock [42] examined the relative performance of an 'open' versus a 'closed' regime and explicitly characterised the circumstances in which an open approach, despite its effect on facilitating imitation, results in a higher level of innovation. The outcome is strikingly simple: when open source reduces the cost of innovation at least as much as the cost of imitation, open regime is supporting innovation. This is frequently the case in open source thanks to user involvement, crowd development, code reuse, etc. And this is not even taking into account business model innovations that have brought many additional advantages to the first mover, supporting innovation even more by augmenting the value of it for the inventor.

More recently, Rayna and Striukova [43] compared the performance of open source and patent pools in the open innovation context. Patent pools are a way to pool patents from various inventors such that they are made available as a package, simplifying their use for innovation. They follow the intellectual property paradigm, adapted to the open innovation context, in a 'coopetition' [36] setting. The issues of financial and nonfinancial benefits, appropriability, standards, cooperation, risks and feasibility are, in turn, discussed for each of the structures. No structure is declared better than the other per se, but the authors underline pros and cons of each. The lesson for us is that open source is at least as valid as traditional patent systems for innovation. Sometimes it will work better; at other times it should be avoided.

Some obstacles remain indeed present for open source innovation. A major problem is that open source brings with it an inherent risk of licence conflicts that may become an issue when aiming to develop an innovative demo into an actual product [25]. A lot of work is ongoing to reduce this risk. Another risk is that adopting open source increases the business risk coming from the integration of differentiating contributions within the core release stream. It is also not very clear how the requirements management should adapt to the use of OSS to fully exploit its innovation potential [52].

13.4.2.1   Open Source and Sustainability

We relate hereunder a number of ways in which OSS can be considered more sustainable than traditional software.

Being free (as in 'no money required'), OSS is potentially contributing in larger diffusion of modern living tools, thus hoping to reduce inequalities. In particular, the potential of OSS in developing countries is therefore important. But, more importantly, beyond reducing licensing costs, OSS is hoped to promote indigenous technological development by having access to the source code, avoiding being hostage to proprietary software, advancing knowledge more quickly and helping to set up an information economy, in a way that respects the local culture and techniques [7]. In this way, it is not only the free and open character of the software but also the participative development process that is a factor of sustainability. While the process of building that new economy may not be cheaper than buying proprietary software, its benefits are much higher for the developing country. What is true for developing countries is true in general: OSS has an important role in open learning, potentially reducing inequalities more extensively than reducing licensing costs. Programming is not limited to chartered engineers any more: the barrier to join has been significantly reduced. The educational world is more and more grasping the opportunities offered by OSS, sometimes in advanced ways [24].

The possibility to adapt to OSS easily is also a vector for reducing inequalities. In developing countries, it is an essential property that enables to use software that is really adapted to the huge variety of contexts and specific needs [7]. In the same way, OSS is also an important provider of software for people with disabilities. There are many examples, such as text-to-speech libraries for visually impaired people [48], text-to-Braille [15], improvement of open source Web browsers [20], etc. Again, it is not only the free character of OSS that is important but also the fact that communities are available to develop in a participative way that makes it a sustainability driver.

Open source has also demonstrated an important potential in managing natural disaster crises and humanitarian situations, and more work is ongoing in that direction [27]. At a more preventive stage, it is known that OSS plays an important role in supporting, among others, climate science [47]. Other humanities benefit from this characteristic of OSS, such as health [5].

One of the main interests of open source is its ability to support standardisation and reuse. An important challenge at this level is continuously the attention of an important researcher community [47]. This allows allocating fewer resources to build better products, offering a huge advantage for the sustainability of the sector.

Opening the source code has also an obvious impact on its auditability by the public, that is, its transparency. Consequently, it can offer important guarantees to users, notably in terms of security, privacy and sustainability in the sense of the absence of planned obsolescence. These nonfunctional concerns are gaining more and more interest from the public, and experts have been studying them since long. Privacy is considered by experts as a key stone for the future of the Internet (the

term *privacy* counts >11.900 hits on DBLP [26], versus 314 for *maintainability* for example). Obsolescence has since long been pointed as a key problem of unsustainable innovation [51]. While OSS solves this problem on the one hand, it is however important to note that the obsolescence of an OSS product is mostly linked to the sustainability of its community.

Finally, if strengthening social links between people on the planet is indeed part of social sustainability, then the OSS movement can certainly be seen as a driver for it. Indeed, the main reasons why people engage in OSS projects are peer recognition and the feeling of belonging to a community, as well as the feeling of enjoyment procured by one's creative contribution to something, as Camara and Fonseca summarise from many studies [7].

As a conclusion to this subsection, we think there is ample evidence to show that OSS, a paramount example of openness and participation in IT, extensively supports sustainable innovation.

13.5 Methodological Proposition for Supporting Sustainable Innovation in Software Engineering

In the 3 last years, our lab has been pioneering research on how to support participative and sustainable innovation in the software domain. We are gradually building elements of a methodology, taking specific approaches on creativity, collaboration and sustainability in software and, more particularly, requirements engineering. Its basic constituents are depicted below (Fig. 13.3) and explained further down.

13.5.1 A Conceptual Framework for Creativity in the Design of ICT Systems

This framework, shown in Fig. 13.4, explains what different concepts may lie behind this simple word: creativity. It allows to better understand creativity in a particular context and the methods and techniques to be adopted accordingly. The framework describes five dimensions and three contextual factors that give the specific creativity *identity card* to a project.

13.5.2 Factors for Collaborative Creativity

In a recent work [29], which is under empirical validation, we have studied factors that influence the effectiveness of groups in collaboratively creative efforts
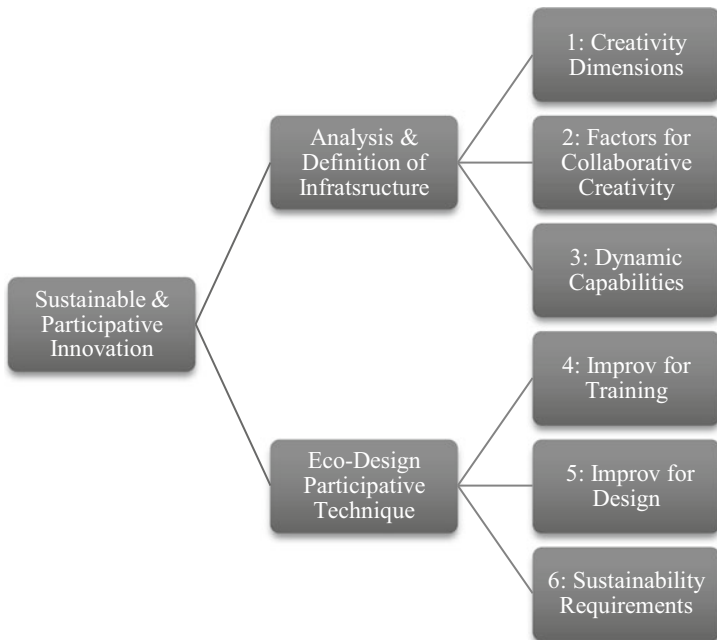
Fig. 13.3 Methodological framework for sustainable software innovation
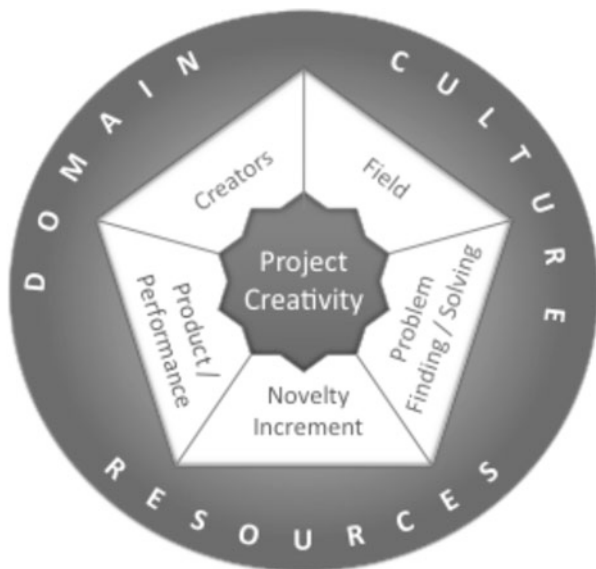
Fig. 13.4 Creativity dimensions framework
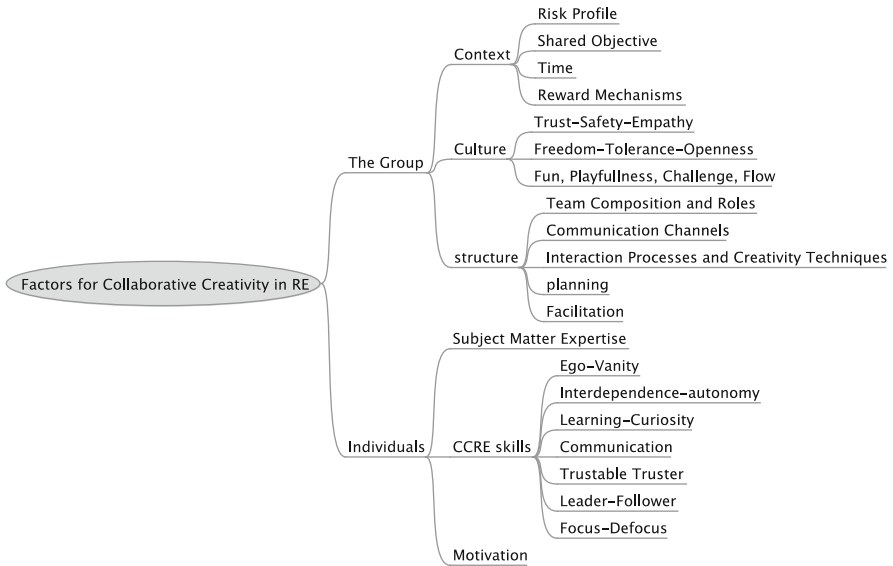
Fig. 13.5 A conceptual framework for collaborative creativity in RE

(Fig. 13.5). We distinguished between factors relating to the team and to the individual.

The team factors are further split into team context, the team values that are shared among the team members and the team structure that describes how the team is organised internally. This study enables to take a holistic approach when trying to support collaborative efforts, by ensuring a full covering of the attention points. The framework however does not prescribe specific methods or techniques: the framework only helps the facilitator to build his process in a systematic way.

13.5.3 *Dynamic Capabilities for Sustainable Development*

In this research [8], we explore the impact of sustainability requirements on dynamic capabilities that a company must develop and maintain to remain competitive in a turbulent environment [49]. In particular, we analyse the new innovation capacity to integrate the three pillars of sustainable development. To do this, we consider the three basic functions of dynamic capabilities (detect, assess and transform) and identify new requirements to fulfil these three functions. A field study of the process of innovation in the ICT industry, in collaboration with IBM, supports this analysis. It shows the new dynamics introduced in this sector to strategically integrate the dimensions of sustainability—particularly energy efficiency—in innovation and the different phases of the innovation process. We finally derive a conceptual framework highlighting the dynamic in presence when

a socio-technical system transits towards sustainable development. The model allows evaluating which endogenous and exogenous organisational pressures shape the development and dissemination of these sustainable ICT technologies.

13.5.4   Improv-Based Training for Participatory Creativity

Theatrical improvisation (*improv*) is a form of stagecraft in which a group writes, directs and plays a piece in the instant. The challenge is to maximise exchanges between protagonists who do not have, by definition, the same vision of history at the beginning. Through a communication endangered by the immediacy of the moment, they will have to make their distinct imaginary worlds as one. The same problem arises in the participative design context: we need to be able to build together a unique solution tailored to a problem, taking into account a variety of visions and constraints. Similarly, participants often have a diverging idea of the problem and the solution; it will now have to converge, maximising the satisfaction of all stakeholders. The issues are the same: How to find good ideas? How to use the conflict as a source of creativity? How to find its place? How to effectively communicate its point of view?

Practitioners of improvisation have developed rules and techniques to perfect their art: our contribution was to recover this work to help design teams to understand the forgotten mechanisms underlying collective creativity—listening, openness, trust, acceptance, co-construction, shared responsibility, etc. [33]. This technique has the potential to help us make the switch from a deeply rooted habit for competition towards a collaborative spirit. Its strong points in this respect are its use of gaming to talk directly to our deeper instincts and provide safe but close-to-reality exercises that everyone can play as the capacities are built progressively. The feedback is also easier to receive in gaming than on the real job.

13.5.5   Creative, Agile and User-Experience-Centred Design Technique

Also based on improvisational theatre, this technique uses improv as a 'machine to build stories together'. Under construction, this tool is hoped to help system designers who have realised the importance of scenario-based participatory work [31, 32]. Its strong points are to tap into people's ability to tell stories and to embody them. Contextualisation and action are supposed to facilitate communication, while the framework of improvisation rules ensures true participation and facilitates the story-telling abilities of the group.

13.5.6 Sustainable Requirements Techniques

Recognising that sustainability could be seen as a particular nonfunctional requirement, as well as safety or performance for example, the centre has been working to initiate research on the tools needed taking those new requirements into account. In the area of security, for example, numerous studies have been conducted to design secure systems 'by design'. Our approach was to initiate a similar path for sustainability. We proposed a series of tools that can be added to the panoply of systems designers eager to control the environmental impact of the product [30]. These tools include add-ons to goal models, context diagrams, stakeholder analysis diagrams, misuse cases, etc. We also co-initiated a series of international workshops on the subject [39].

13.6 Conclusion

Open and participative innovation is gaining interest worldwide, as it shows its ability to perform better in a world that is not focused solely on economic growth. Greater participation is pushing a more subtle view on value, one where people and nature have their place. People at various levels are empowered, and if they feel part of a single common planet, they become more responsible and build more sustainable systems. Collaborative creativity has the power to give this feeling of unity back, and the few places where collaboration is successfully replacing competition are giving us reasons to hope.

At the core of the post-industrial economy, the software industry is a key enabler in this context. Software is an ideal place to play open and participative, as OSS demonstrates. It also has the power to facilitate participation in all other domains, by helping in barrierless knowledge transfer and facilitating distributed discussions.

As experts in the field, we must act as facilitators and consultants to help the world build sustainable systems. The building process will be key: it has to be participative and open to the best extent. It has to be smart about this, because neither participation nor openness is obvious in today's still dominant economical settings, company structures or people's minds. We have to keep reinventing business models and design techniques that will make it work. We have to use tools that will help us think about the impact of the system on society and the planet and to take informed decisions based on this.

We have never been so close to a massive transition of the economic and governance systems for a better world. We need to grab this chance: as ICT system builders, we have a key role to play.

References

1. Abernathy WJ, Utterback JM (1978) Patterns of industrial innovation. Technol Rev 64:254–282
2. Aitamurto T (2011) The impact of crowdfunding on journalism. Journalism Pract 5(4):429–445
3. Arnstein SR (1969) A ladder of citizen participation. J Am Inst Plann 35(4):216–224
4. Arthur WB (1989) Competing technologies, increasing returns, and lock-in by historical events. Econ J 99:116–131
5. Asare P et al (2012) The medical device dongle: an open-source standards-based platform for interoperable medical device connectivity. In: 2nd ACM SIGHIT international health informatics symposium. ACM, New York, pp 667–672
6. Björkdahl J (2009) Technology cross-fertilization and the business model: the case of integrating ICTs in mechanical engineering products. Res Policy 38(9):1468–1477
7. Camara G, Fonseca F (2007) Information policies and open source software in developing countries. J Am Soc Inform Sci Technol 58(1):121–132
8. Castiaux A (2012) Developing dynamic capabilities to meet sustainable development challenges. Int J Innovat Manag 16(6):16
9. Castiaux A, Paque S (2009) Participative innovation: when innovation becomes everyone's business. Int J Enterpren Innovat Manag 10(2):111–121
10. Chesbrough HW (2003) Open innovation: the new imperative for creating and profiting from technology. Harvard Business Press, Boston, MA
11. Doelle M, Sinclair AJ (2006) Time for a new approach to public participation in EA: promoting cooperation and consensus for sustainability. Environ Impact Assess Rev 26 (2):185–205
12. Ebert C (2007) Open source drives innovation. IEEE Software 24(3):105–109
13. Freeman RE et al (2004) Stakeholder theory and "the corporate objective revisited". Organ Sci 15(3):364–369
14. Freeman RE (2010) Strategic management: a stakeholder approach. Cambridge University Press, Cambridge, MA
15. Frees B et al (2010) Generating Braille from OpenOffice.org. In: Miesenberger K et al (eds) Computers helping people with special needs. Springer, Berlin, pp 81–88
16. Fung A (2006) Varieties of participation in complex governance. Publ Admin Rev 66:66–75
17. Gladwell M (2011) Creation myth. http://www.newyorker.com/reporting/2011/05/16/110516fa_fact_gladwell?currentPage=all
18. Goode L (2009) Social news, citizen journalism and democracy. New Media Soc 11(8):1287–1305
19. Hall J (2000) Environmental supply chain dynamics. J Cleaner Prod 8(6):455–471
20. Hanson VL et al (2005) Improving Web accessibility through an enhanced open-source browser. IBM Syst J 44(3):573–588
21. Hardin G (1968) The tragedy of the commons. Science 162(3859):1243–1248
22. Hertel G et al (2003) Motivation of software developers in Open Source projects: an Internet-based survey of contributors to the Linux kernel. Res Policy 32(7):1159–1177
23. Von Hippel E (2005) Democratizing innovation. MIT Press, Cambridge, MA
24. Pavlik PI Jr et al (2012) Facilitating co-adaptation of technology and education through the creation of an open-source repository of interoperable code. In: Cerri SA et al (eds) Intelligent tutoring systems. Springer, Berlin, pp 677–678
25. Kilamo T et al (2012) Open source, open innovation and intellectual property rights – a lightning talk. In: Hammouda I et al (eds) Open source systems: long-term sustainability. Springer, Berlin, pp 298–303
26. Ley M, Bast H. The DBLP computer science bibliography. http://www.dblp.org
27. Li JP et al (2013) A case study of private–public collaboration for humanitarian free and open source disaster management software deployment. Decis Support Syst 55(1):1–11

28. Lyons M et al (2001) Participation, empowerment and sustainability: (how) do the links work? Urban Stud 38(8):1233–1251
29. Mahaux M et al (2013) Collaborative creativity in requirements engineering: analysis and practical advice. In: Proceedings of the 7th international IEEE conference on research challenges in information science, Paris, France
30. Mahaux M et al (2011) Discovering sustainability requirements: an experience report. In: Requirements engineering: foundation for software quality, pp 19–33
31. Mahaux M et al (2010) Making it all up: getting on the act to improvise creative requirements. In: Proceedings of the 18th IEEE conference on requirements engineering. IEEE, Sydney
32. Mahaux M, Hoffman A (2012) Research preview: using improvisational theatre to invent and represent scenarios for designing innovative systems. In: 1st International workshop on creativity in requirements engineering, Essen, Germany
33. Mahaux M, Maiden N (2008) Theater improvisers know the requirements game. IEEE Software 25(5):68–69
34. Merten T et al (2011) Improved representation of traceability links in requirements engineering knowledge using Sunburst and Netmap visualizations. In: 2011 Fourth international workshop on managing requirements knowledge (MARK), pp 17–21
35. Mormont M et al (2006) La participation composante du développement durable: quatre études de cas. VertigO Rev Électronique En Sci Environ 7:2
36. Nalebuff BJ, Brandenburger A (1996) Co-opetition. Harper Collins, New York
37. Noll J (2007) Innovation in open source software development: a tale of two features. In: Feller J et al (eds) Open source development, adoption and innovation. Springer, New York, pp 109–120
38. Nyman L et al (2012) Perspectives on code forking and sustainability in open source software. In: Hammouda I et al (eds) Open source systems: long-term sustainability. Springer, Berlin, pp 274–279
39. Penzenstadler B et al (eds) (2013) Proceedings of the 2nd International workshop on requirements engineering for sustainable systems. Presented at the RE4SuSy@RE, Rio, Brazil
40. Penzenstadler B et al (2013) University meets industry: calling in real stakeholders. In: 2013 I. E. 26th conference on software engineering education and training (CSEE T), pp 1–10
41. Penzenstadler B, Femmer H, Richardson D (2013) Who is the advocate? Stakeholders for sustainability. In: 2nd International workshop on Green and Sustainable Software (GREENS) 2013. IEEE, pp 70–77
42. Pollock R (2009) Innovation, imitation and open source. Int J Open Source Software Process 1 (2):28–42
43. Rayna T, Striukova L (2010) Large-scale open innovation: open source vs. patent pools. Int J Technol Manag 52(3):477–496
44. Reuchamps M (1978) Le G1000. http://orbi.ulg.ac.be/handle/2268/142718
45. Schumpeter JA (1975) Capitalism, socialism, and democracy. Harper Colophon, New York
46. Sonntag V (2000) Sustainability – in light of competitiveness. Ecol Econ 34(1):101–113
47. Stephens A et al (2012) The challenges of developing an open source, standards-based technology stack to deliver the latest UK climate projections. Int J Digit Earth 5(1):43–62
48. Strobbe C et al (2010) Generating DAISY books from OpenOffice.org. In: Miesenberger K et al (eds) Computers helping people with special needs. Springer, Berlin, pp 5–11
49. Teece DJ (2007) Explicating dynamic capabilities: the nature and microfoundations of (sustainable) enterprise performance. Strat Manag J 28(13):1319–1350
50. Tritter JQ, McCallum A (2006) The snakes and ladders of user involvement: moving beyond Arnstein. Health Policy 76(2):156–168
51. Waldman M (1993) A new perspective on planned obsolescence. Q J Econ 108:273–283
52. Wnuk K et al (2012) How can open source software development help requirements management gain the potential of open innovation: an exploratory study. In: Proceedings of the

ACM-IEEE international symposium on empirical software engineering and measurement. ACM, New York, pp 271–280
53. Etude sur l'innovation participative 2011. http://www.innovacteurs.asso.fr/?page_id=29
54. G1000: Platform for democratic innovation. http://www.g1000.org/en/
55. Overview – Redmine. http://www.redmine.org/
56. The open source definition (annotated) | Open Source Initiative. http://opensource.org/osd-annotated

Index