



Ervin Varga

Unraveling Software Maintenance and Evolution

Thinking Outside the Box

EXTRAS ONLINE

 Springer

Unraveling Software Maintenance and Evolution

Ervin Varga

Unraveling Software Maintenance and Evolution

Thinking Outside the Box



Springer

Ervin Varga
Expro I.T. Consulting
Kikinda, Serbia

ISBN 978-3-319-71302-1 ISBN 978-3-319-71303-8 (eBook)
<https://doi.org/10.1007/978-3-319-71303-8>

Library of Congress Control Number: 2017963098

© Springer International Publishing AG 2017

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, express or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Printed on acid-free paper

This Springer imprint is published by Springer Nature
The registered company is Springer International Publishing AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

To my family (my wife Zorica and my sons Andrej and Stefan), who withstood another book project.

Preface

During my professional career as a consultant, I have witnessed a frequently reappearing pattern in practice. A project starts seemingly okay, lots of cool technologies are applied, but eventually things simply get stuck. When that happens, all sorts of ad hoc remedies are tried out until the company's new product is finally shipped. Nevertheless, even in this success scenario, evolving the product becomes painful. Why does this happen so often? Do we lack more cool stuff (programming languages, software development methods, tools, etc.) in the industry? Are we bad at teaching people how to perform software development? Maybe with a new language we could reduce quality issues and increase productivity at the same time. Perhaps using a more sophisticated tool would solve our core problems. Possibly by leveraging massive open online courses (MOOC), we could smooth out educational differences worldwide. However, as F. Brooks has pointed out in his seminal book *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition* (2nd Edition) (Addison-Wesley, 1995), we should stop hoping for a magical *silver bullet*. But what should we do then? I've contemplated a lot about this question and concluded that revisiting our roots is probably the best action. The reason is simple; without knowing what drives our mental processes, we will always misapply even the best technologies, tools, and languages. This book lets you embark on a journey toward unraveling some fundamental rules/principles governing maintenance of software-intensive systems. It will equip a software engineer with the necessary knowledge and experience to judge what/when/how to apply to accomplish the goal, that is, to deliver a maintainable and evolvable solution. It will help a teacher craft more effective courses by focusing on vital aspects of software engineering.

Software maintenance and evolution demand a creative design approach that combines analytical and systems thinking in a holistic manner; typical problem-solving exercises are far from reality and cannot be easily mapped to everyday situations. This book is organized to emphasize those creative aspects of the software maintenance effort. Consequently, it follows an unusual structure, contrary to accustomed hierarchical, rigid, and formal structures present in most books. Each set of rules/principles is exemplified inside a self-contained educational unit. This book hops around various such sets showing their interconnectedness.

The resulting graph is not a “straight line.” Nonetheless, a book must necessarily retain a linear exposition model, even though a creative design process is nonlinear—again, something neatly illustrated by F. Brooks in his book *The Design of the Design: Essays from a Computer Scientist* (Addison-Wesley, 2010). Brooks argues that a true design process cannot be templated into an automated routine. Consequently, this book also showcases that blithely following the so-called technical rationality mindset results in a rigid, shortsighted, and unmaintainable solution. The aim is to motivate a reader to think in broader perspectives. For example, a software engineer who is aware of her or his global role will more appreciate the importance of preserving the architectural consistency and integrity of a system during code construction. The dreaded “What isn’t mine...isn’t my problem” syndrome would vanish.

The book will strive to expose intermediate phases (a background trial-and-error process) toward the outcome, even if they raise unwanted details. Many authors like to hide them by revealing a purely idealized final form of a process. Such phenomenon is nicely explained by Parnas and Clements in their article “A Rational Design Process—How and Why to Fake It.” The reason to expose things in an unpolished fashion is compelling: it demonstrates to a reader how mistakes accumulate and influence future activities. Being aware of such a chain of cause–effect pairs is central to understanding and classifying design decisions in each context.

Software maintenance work is often an unencouraging activity, something that doesn’t motivate hotshots. This book will exhibit the opposite; it is exactly maintenance and evolution that demands the most creativity and thinking out of the box. This is on a par with the quote from Albert Einstein: “We cannot solve our problems with the same level of thinking that created them.”

The intention of this book is to basically bridge the gap between theory and practice. As such, it is perfectly suitable as a course material. One of the biggest challenges in education is how to effectively endow a lecture with persuasive examples. This is especially hard to achieve in the realm of software maintenance and evolution. Albert Einstein once said that “Example isn’t another way to teach; it is the only way to teach.” Hence, the book is abundant with examples for every elaborated concept. This makes it a pragmatic complementary material of existing curricula. As all examples are implemented in Java SE 8 using Maven as a build tool, it is even possible to use this book as an additional source of Java/Maven demos.

The book is comprised of four major parts: Introduction to Key Concepts (Chaps. 1–4), Forward Engineering (Chaps. 5 and 6), Reengineering and Reverse Engineering (Chap. 7), and DevOps (Chaps. 8–10). These parts may be read in any order, although the introductory material is the basis for the other parts. Each part can also be studied in an arbitrary fashion; however, I recommend a sequential order since the presentation of concepts/rules/principles inside them assumes a linear arrangement. The chapters further divide the content of parts. The only exception is the very first chapter that introduces software maintenance and evolution as well as serves as a general guide for the content of this book. This chapter

also presents some known case studies of software failures to demonstrate that “bugs” aren’t usually bare programming mistakes. Such an understanding will help you better appreciate the concepts introduced later.

Every subsequent chapter has the following structure:

- *Inaugural section* explains the topic of the chapter.
- *Problem statement(s)* sets the stage for a discussion.
- *Problem resolution(s)* describes the various solutions for the problem stated in the above section.
- *Summary* wraps up the chapter’s material with key takeaways.
- *Exercises* tests your understanding of the chapter as well as sparks more thinking about the problem.
- *Further reading* provides references to external sources that you should read (these further references are understood to be in addition to the regular bibliographic references).
- *References* includes the material used for the chapter.

The Introduction to Key Concepts part briefly introduces you to the major elements of the software maintenance knowledge area. It highlights some core concepts that are indispensable for you to see the forest for the trees. Each concept is illuminated with a worked example.

The Forward Engineering part debunks the myth that being fast and successful during initial development is all that matters. We will consider two categories of forward engineering: an inept initial project with a multitude of hard evolutionary phases and a proper initial project with multiple straightforward future increments.

The Reengineering and Reverse Engineering part shows the difficulties of dealing with a typical legacy system. We will tackle tasks such as retrofitting tests, documenting a system, restructuring a system to make it amenable for further improvements, etc.

The DevOps part focuses on the importance and benefits of crossing the development vs. operation chasm. It shows you how the DevOps paradigm may turn a loosely coupled design into a loosely deployable solution. This is similar in nature to the way modularity solves the *Ivory Tower Architecture* anti-pattern (see agilemodeling.com/essays/enterpriseModelingAntiPatterns.htm).

To grasp the big picture, a student must become part of the inner creative design processes, rather than passively watch the stage and recollect facts. Examples are devised in a manner to realize this idea. As a positive side effect, a student may assume the role of a software engineer in each story (example) and feel the hassle of being exposed to design critiques. This is a radically different way of teaching, compared to that advocated in most books and university curricula.

This book assumes that the reader is familiar with the Java programming language and has a basic understanding and experience regarding software construction and testing. However, these are not strict prerequisites. The first chapter and part are compulsory for students, as they form the basis for the other parts. The Forward Engineering part may be combined with a classical software development

course, while the Reengineering and Reverse Engineering part with traditional software maintenance material. Chapters from parts II and III may be independently selected as needed for a course. The DevOps part may be taught as a separate course.

Publishing a book is a team effort. I am thankful to the team at Springer, especially Ralf Gerstner for his professionalism and tremendous support during the preparation of this manuscript. Rajendran Mahalakshmi did a great job as a project manager at handling the production process.

Kikinda, Serbia

Ervin Varga

Contents

Part I Introduction to Key Concepts

1	Introduction	3
1.1	Software Maintenance and Evolution Overview	4
1.2	The Learning Path	9
1.3	Context, Rules, and Principles	10
1.4	Maintainability and Development	12
1.5	Architecture and Evolution	13
1.6	Ad hoc Development	13
1.7	Disciplined Development	13
1.8	Reengineering and Reverse Engineering	14
1.9	Multifaceted Loose Coupling	14
1.10	Monitoring and Logging	14
1.11	Scale of Deployment	15
1.12	Exercises	15
	References	16
2	Context, Rules, and Principles	17
2.1	Lack of Context: Problem	19
2.2	Lack of Context: Resolution	23
2.3	Lack of Knowledge: Problem	24
2.4	Lack of Knowledge: Resolution	24
2.5	Summary	25
2.6	Exercises	26
	References	28
3	Maintainability and Development	31
3.1	Safety Argument: Problem	31
3.2	Safety Argument: Resolution	32
3.3	Summary	36
3.4	Exercises	36
	References	38

4	Architecture and Evolution	39
4.1	Extension of the System: Problem	41
4.2	Extension of the System: Resolution	54
4.3	Elimination of a Technical Debt: Problem	60
4.4	Elimination of a Technical Debt: Resolution	61
4.5	Summary	67
4.6	Exercises	68
	References	70

Part II Forward Engineering

5	Ad Hoc Development	73
5.1	Initial Version: Problem	74
5.2	Initial Version: Resolution	75
5.3	Non-square Matrix Multiplication Bug: Problem	84
5.4	Non-square Matrix Multiplication Bug: Resolution	85
5.5	Inadequate Dimensionality: Problem	87
5.6	Inadequate Dimensionality: Resolution	87
5.7	Retrieve Meta Data: Problem	91
5.8	Retrieve Meta Data: Resolution	91
5.9	Concurrency Extension: Problem	94
5.10	Concurrency Extension: Resolution	94
5.11	Concurrency Bug: Problem	97
5.12	Concurrency Bug: Resolution	97
5.13	Heisenbug: Problem	100
5.14	Heisenbug: Resolution	100
5.15	Printout Improvement: Problem	105
5.16	Printout Improvement: Resolution	105
5.17	Comparison Bug: Problem	111
5.18	Comparison Bug: Resolution	111
5.19	Sparse Matrix Extension: Problem	114
5.20	Sparse Matrix Extension: Resolution	114
5.21	Sparsity Bug: Problem	118
5.22	Sparsity Bug: Resolution	118
5.23	Maintainability Issues: Problem	119
5.24	Maintainability Issues: Resolution	119
5.25	Matrix Transposition: Problem	142
5.26	Matrix Transposition: Resolution	142
5.27	Summary	151
5.28	Exercises	152
	References	156
6	Disciplined Development	159
6.1	Initial Version: Problem	161
6.2	Initial Version: Resolution	162

6.3	First Usability Enhancement: Problem	215
6.4	First Usability Enhancement: Resolution	215
6.5	Second Usability Enhancement: Problem	220
6.6	Second Usability Enhancement: Resolution	220
6.7	Multiple Environments: Problem	226
6.8	Multiple Environments: Resolution	227
6.9	Record of Actions: Problem	237
6.10	Record of Actions: Resolution	237
6.11	Summary	241
6.12	Exercises	242
	References	244

Part III Re-engineering and Reverse Engineering

7	Reengineering and Reverse Engineering	247
7.1	Performance Test Suite: Problem	248
7.2	Performance Test Suite: Resolution	251
7.3	Extension of the System: Problem	252
7.4	Extension of the System: Resolution	257
7.5	Legacy Application: Problem	274
7.6	Legacy Application: Resolution	276
7.7	Restructure of the System: Problem	283
7.8	Restructure of the System: Resolution	283
7.9	Summary	286
7.10	Exercises	286
	References	288

Part IV DevOps

8	Multifaceted Loose Coupling	293
8.1	Reuse: Problem	295
8.2	Reuse: Resolution	295
8.3	Summary	296
8.4	Exercises	297
	References	297
9	Monitoring and Logging	299
9.1	Collecting the Data	299
9.2	Interpreting the Collected Data	301
9.3	Visualizing the Gathered Data	304
9.4	Embracing the Holistic Notion of Monitoring	307
9.5	Measurements: Problem	308
9.6	Measurements: Resolution	308
9.7	Centralized Logging: Problem	310

9.8	Centralized Logging: Resolution	311
9.9	Summary	316
9.10	Exercises	316
	References	317
10	Scale of Deployment	319
10.1	Multi-phased Continuous Deployment/Delivery	321
10.2	Infrastructure as a Code: Problem	323
10.3	Infrastructure as a Code: Resolution	323
10.4	Deployment: Problem	329
10.5	Deployment: Resolution	329
10.6	Summary	332
10.7	Exercises	333
	References	333
	Index	335

List of Abbreviations/Symbols

API	Application programming interface
BDD	Behavior-driven development
COTS	Commercial off-the-shelf
DSL	Domain-specific language
JMS	Java message service
OO	Object-oriented
REST	Representational state transfer
TDD	Test-driven development

Part I

Introduction to Key Concepts

Software maintenance and evolution is usually the longest phase in a large software-intensive system's life cycle. This is the period after the first version had been deployed until the retirement of the product. For some safety-critical software systems, it may span a couple of decades. Open-source software is also a good example for the longevity of the software maintenance phase. Obviously, over such a long period, a product must undergo many changes (defects are reported, new features are requested, new technologies must be supported, etc.). Depending on how those changes are managed and delivered, a product may be perceived as successful or not, where the success factor could be measured in various ways (one common criteria is the positive return on investment of software maintenance activities). The major goal of this book is to unearth hidden artifacts in software maintenance and evolution (a technical process) to put you in a better position to seize control over your product. This chapter sets the stage in our quest to unravel the underlying secrets of this process. You may treat this chapter as a user's guide for this book that explains the importance and roles of the subsequent chapters as well as their interrelationships.

Software engineering is nowadays facing an extremely difficult task in controlling the zealous specialization of its domain. According to [2], we currently have 15 main knowledge areas (the software maintenance is one such area) and 7 related disciplines. The creation of a myriad of further subdisciplines is an inevitable consequence of rapid expansions in all segments.¹ However, such fragmentation may produce isolated and sterile disciplines that will duplicate the educational effort to teach common foundations of computer science and software engineering [3]. Without an overarching curriculum, software engineers working in one area will become ignorant of the value in the other areas (under an assumption that they are even aware of their existence). For example, a typical programmer,

¹Each subdiscipline may be further divided by different technologies. All in all, the branching factor is enormous.

only doing construction, cannot understand his role nor appreciate how his work may impact later evolution. Software engineering requires a holistic teaching approach [4].

We can easily witness a proliferation of programming languages, frameworks, tools, and technologies.² For an average software developer, this is an inescapable labyrinth of options with lots of traps. A less apparent fact is the existence of a vast number of software methods that are supposed to be used in an all-or-nothing fashion. In some way, this is an even bigger problem. Many persons/companies/institutes eagerly try to concoct and promote their own variant method (as their major differentiator on the market) to gain some advantage. Very few try to analyze the long-term consequences of such method fragmentation. As a potential remedy for this *software methods* conundrum, there is a promising community initiative called SEMAT (Software Engineering Method and Theory) with its core output named Essence (for more details visit semat.org or read [12–13] for a superb explanation of why methods need theory). It attempts to provide a unified kernel for software engineering comprised of a solid underpinning theory, proven principles, and best practices. Thanks to SEMAT, software teams would be able to craft customized software processes and efficiently share practices with other teams (see [5] for an example of how to offer an industry verified test-driven development practice as a reusable asset). It is exactly this core idea of SEMAT that has motivated me to write this book. It has a similar aim, i.e., to present a set of foundational elements that drives software maintenance and evolution processes.

1.1 Software Maintenance and Evolution Overview

This section is a short recap about the software maintenance and evolution knowledge area as specified in the international standard for software maintenance IEEE 14764 [2]. It is provided here for the sake of completeness and to specify some common terms and presuppositions that are used throughout this book.

As is the case with all products (including software), they need maintenance. That need is tightly related to usefulness since a product must always satisfy user requirements. Software maintenance is specific due to the peculiar nature of software itself. We rarely worry about wearing and tearing of parts, as is the case with mechanical products. On the other hand, software is always extended, many times beyond the initial product vision. There is even a view that software maintenance is a continued development, and in this respect, we talk about never-ending evolution (successful improvements and extensions of a product). To make this evolution smooth, we strive to achieve a high level of maintainability in our

²Part of this chaos are the accompanying hypes and buzzwords, many of them exploited by persons/companies seeking an opportunity to earn profit by selling fog. Grounding claims is obviously a crucial skill to develop to battle against this issue (see [9] for a nice summary about the techniques to pursue the ground truth).

product. Maintainability as a quality attribute represents the capability to efficiently incorporate changes into the code base. This assumes that changes are done in a nonobtrusive fashion, i.e., we don't break existing stuff. Here comes the importance of having a battery of automated tests to prevent regression, which is proof that things in software engineering are truly interrelated. Maintainability and testability (a quality characteristic that shows how easy it is to test a code) mutually reinforce each other. It isn't surprising then to see why testability is even put under maintainability in the ISO/IEC 25010 standard.

There is a huge misconception between the terms *maintenance phase* and *maintenance activity*. The former demarcates a software life cycle period that starts after the warranty period, or post-implementation support delivery. The latter designates an activity that occurs in the software maintenance process. The set of such activities (requirements engineering, design, construction, testing, documentation, deployment, etc.) is a lot like that from development, as they share the same goal. In both cases, the focus is on altering a software without disturbing its conceptual integrity and consistency. For example, refactoring (as a small-scale reengineering endeavor) may happen during both development and maintenance. Nevertheless, maintenance does possess some unique activities, such as product handover from developers to maintainers, impact analysis to judge the scope of a change request, program comprehension³ to understand the code base (this may trigger a reverse engineering effort), etc. You may find a detailed treatment of these activities in [1].

According to IEEE 14764, we can differentiate between four categories of maintenance that may be grouped into two major types (*correction* and *enhancement*)⁴:

- *Corrective (correction)* is related to fixing failures (observable problems causing some harm in production) that are usually reported by a customer.
- *Preventive (correction)* is related to fixing faults before they transform into a failure. This requires a thorough understanding of how a customer will use a product, as some faults may never turn into failures under some circumstances.⁵
- *Adaptive (enhancement)* pertains to modifications due to changes in the environment (e.g., business process, external regulatory law, platform, technology, etc.). The goal is to proactively keep the software usable.
- *Perfective (enhancement)* deals with all sorts of proactive changes to boost the quality of the software (including maintainability). This activity is essential to combat the detrimental effects of continuous changes to a code base during

³According to [2], approximately half of the total maintenance effort goes to this activity. Therefore, writing readable code is paramount to increase maintainability.

⁴Enhancement comprises about 80% of software maintenance according to some sources mentioned in [2].

⁵Suppose that your code has a function accepting a numeric argument that is used as a divisor. Without checking against zero, it may generate an exception. However, if that argument will never receive a value of zero, then this potential fault cannot become a failure (operational fault).

maintenance and evolution. As new features are added, complexity increases and the existing code structure deteriorates. This is nicely documented in the *Laws of Program Evolution* [6].

All the previously mentioned maintenance activities and categories must be evaluated against the type of the target software system. According to [6], there are three such types:

- *S type system* has a finite enumerable problem space with a fixed environment. Once the initial requirements are specified, nothing will change in the future (here, we don't count potential user interface enhancement as impactful changes). For example, a popular Tic-Tac-Toe game is an S type system.
- *P type system* has a complex problem space with a fixed environment. Various heuristics are applied to search the problem space. An improvement to the abstract model does induce a considerable change in the system. For example, popular board games like chess and go are P type systems.
- *E type system* is the most complex one due to a highly dynamic environment. It is impossible to predict with 100% accuracy how the environment will fluctuate in the future. For example, an accounting software must follow the latest financial laws issued by a government. Unfortunately, nobody can assure that a new law will not disturb the original architecture of the system.

Software maintenance is important from the economics perspective as well. Usually, maintenance efforts in the industry allocate more resources than used by development projects. As elaborated in [10], the total cost of ownership of a software is comprised of the following elements:

- The initial development of a new product
- *Enhancement* maintenance-type activities
- *Correction* maintenance-type activities
- Customer support

Evidently the ratio is 3:1 between maintenance and new development. To earn profit, it is a wise strategy to apply as many best practices as possible during software development and maintenance. As [10] states, the most efficient maintenance practice is to invest in software engineers, as expertise is the key driver to retain quality and attain profitability.

1.1.1 Technical Debt and Maintainability

Most people understand well the notion of a financial debt. For this reason, Ward Cunningham had come up with a metaphor called *technical debt* to associate an interesting phenomenon with something people are already familiar with (you may read [7] for more details). You accrue debt by always focusing on quick and dirty

solutions while neglecting maintainability. Without paying attention to it, a seemingly superb start of a project may grind to a halt. Most of the time, arguments in favor of speed vs. a systematic style are futile. They are vehemently pushed forward by shortsighted and selfish project managers as well as lazy or incapable developers. Of course, there are rare occasions when you must intentionally incur debt to meet a deadline and prevent a major loss, yet that attitude only “works” in cases of noncritical software systems. On the other hand, by taking maintainability into account, you may effectively combat and pay down a technical debt. Maintainability forces you think about future, and this is exactly the mortal enemy to a hectic and inconsiderate tactic resulting in a technical debt.

Another factor that indirectly causes technical debt and hinders maintainability is wrong estimation, more precisely, when estimation is confused with commitment. This is something that I observe in practice all the time. Underestimation turns into over commitment,⁶ which generates stress. Once that happens, it usually establishes a positive feedback loop producing more stress and chaos (this begs for technical debt, as dirty solutions are heralded as life savers). Again, projects like to stretch the limits, afraid of Parkinson’s law (that an individual will always fill in any available time slot with work). However, the penalty “induced” by this law, as the function of estimated time, is linear rather than exponential, as is the case with an overcommitment.

1.1.2 Dichotomy Between Academia and Industry

As a consultant and an associate professor, I may clearly claim that quite often, what academia favors is simply impractical in an industrial setting. I’m not thinking here about some advanced model-driven software engineering methods nor convoluted formal proofs of correctness of software (these are even indispensable in safety-critical software systems). The issue is much more mundane. Stated simply, academia in general favors one-shot greenfield developments [14]. This is reflected through the following traits of many software engineering courses:

- Project assignments are well specified and fixed throughout the duration of the project.
- Students can carefully choose their teammates (if they happen to work as a team), and teams are not larger than three individuals.
- Projects are, by default, about developing something from scratch; the aim is to emphasize pertinent topics from lectures.
- Projects are based upon a pool of similar assignments.
- Beauty is preferable to a pragmatic and maintainable solution.

⁶This usually happens at the project kickoff meeting, when the level of uncertainty is high. There is an associated term called *Cone of Uncertainty* (introduced by Steve McConnell) that describes the propagation of uncertainty during a project. Any commitment should be postponed until uncertainty drops below some threshold.

Let me illustrate the last item with a concrete example. Inevitably, code readability is one of the most important goals regarding maintenance and evolution. Now, the riddle is how to achieve that. If we neglect maintainability, then the coding style with utmost terseness of constructs seems an appealing proposition. Look at the following decision structure using the K&R style:

```
if (<term 1> && <term 2>) {
    <statement 1>
} else if (<term 1>) {
    <statement 2>
} else if (<term 1> || (<term 2> && <term 3>)) {
    <statement 3>
} else {
    <statement 4>
}
```

Below is the same code reformatted to follow the terse style:

```
if      (<term 1> && <term 2>)           <statement 1>
else if (<term 3>)                      <statement 2>
else if (<term 4> || (<term 5> && <term 6>)) <statement 3>
else                           <statement 4>
```

The second version has no braces and everything is perfectly aligned. Nobody would question that it is more beautiful than the first version. The only glitch is that in practice, it is hopeless to attain. Imagine that the whole code base is manually crafted to follow this style. Now, a change comes in that ruins the table structure, and it must be realigned. The only hope is to leverage a sophisticated tool (preferably integrated into your IDE) to do that for you. A similar problem happens when you try to create nice-looking block comments, like the one presented next:

```
*****
* All looks good and nice      *
* until lines break properly,  *
* so that the layout is intact. *
*****
```

So far, the second solution would score better. There are good indicators that academia has started to realize the necessity to reshape their courses to better fit the harsh reality of an industry. When it comes to coding styles, [11] is a good example. It contains useful advices on how to enhance code readability and clearly acknowledges that the style is more for educational purposes than an industrial environment. However, it does contain many recommendations where

maintainability was considered. The only part that I strongly disagree with is the advice to rely on context to simplify code. For example, using short index variables of *i* and *j* or using *m* instead of *month* may work if the code base is small (a typical student project) so that the contextual baggage is controllable. However, if a programmer needs to quickly scan different parts of a large code base, then always remembering what some short variable name denotes in each context is cumbersome. In this case, using a longer variable name is preferable.

Finally, the biggest issue with current curricula is that we lack sound software engineering methods for large-scale complex IT systems. Currently, education is based on purely technical aspects and thus neglects the fact that modern coalitions of systems are socio-technical systems. The prevailing educational paradigm nowadays revolves around reductionism, whose starting premise is that a complex system is none other than a sum of its parts. Unfortunately, this approach cannot cope with the epistemic complexity⁷ of a system of systems, where emergent properties and behaviors are rather the norm than an exception. Such dynamicity cannot be reliably predicted nor completely specified in advance (for a more thorough insight, you may consult [16]). The next learning path is partially crafted to address the needs of preparing software engineers to efficiently build future complex coalitions of systems.

1.2 The Learning Path

I don't want to give you just another book about software maintenance and evolution. There are lots of generic and more specialized books about software engineering in general and maintenance and evolution (I suggest that you start with [1]). Moreover, you may find lots of books about specific maintenance techniques (refactoring, migration, etc.) as well as books dealing with legacy systems. During my career,⁸ I have observed that several software maintenance-related topics are either totally misunderstood or simply neglected by practitioners. On the other hand, the bulk of the existing literature and courses quickly skim through them or depict them in a rather obscure manner. This book tries to reconcile this dichotomy by providing a bridge between industry and academia pertaining to software maintenance and evolution. The set of subjects discussed in this book is based on my personal experience as a consultant and a university professor; thus, it is highly subjective (it is nearly impossible to come up with an unbiased unified view). However, even if potentially incomplete, I hope that it will stimulate both software engineers in the industry as well as educators to rethink their approach to software engineering, especially maintenance and evolution. Figure 1.1 shows our learning path in this book. Each stage is concisely explained in a subsequent section of this chapter.

⁷This complexity stems from the lack of knowledge about how the system operates, rather than from its essential technical complexity alone.

⁸I've been a professional software engineer since 1994, who had written his first BASIC program in 1982 on a Radio Shack TRS-80 pocket computer.

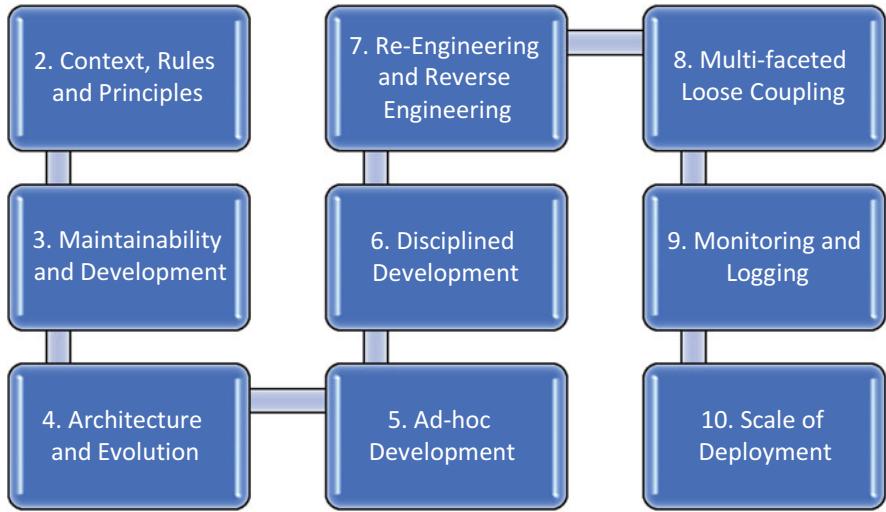


Fig. 1.1 The overall learning path, where boxes denote chapters. As emphasized before, the path may be traversed in a nonlinear fashion; however, it is recommended that you first read Chaps. 2–4 in a sequential order

1.3 Context, Rules, and Principles

A *context* is comprised of all circumstances that influence our interpretation of an event, idea, or problem and has an impact on our action(s). If we ignore the context, then we increase a chance to make an unreasonable move. If we lack knowledge (partially embodied in the form of rules and principles), then we would not be able to use the opportunities encoded in a context nor decipher the context to act in a globally optimal fashion. We should strive to establish a symbiosis between context and knowledge since it boosts productivity and compels us to act professionally. Chapter 2 is completely devoted to exemplifying these aspects.

Software engineering is a highly creative intellectual endeavor. As such, much of the power of a software engineering team emanates from the strength of its members. According to some studies, there is even a $10\times$ variation in performance between different developers [15]. Consequently, acting on humans is the best long-term investment pertaining to software engineering in general. This is the reason why this book devotes so much attention to context, knowledge and experience. We must motivate people to think and act inventively. A precondition for this is to develop context awareness and accrue as much knowledge and experience as possible. The next short example gives a brief teaser.

Table 1.1 Tradeoffs among various quality attributes

	Maintainability					
Portability	↔		Portab.			
Usability	↔			Usability		
Security	×	×	×	×	Security	
Safety	↔			↔		Safety
Performance	×	×	×	×	×	×

The double-sided arrow designates a typically supporting pair, while the cross sign marks the typically conflicting combination (empty cells mean indifference). If you just suck in this table and remember it in a brute-force fashion, then you will neglect the context and miss the point. This is the reason that you should always apply your knowledge and experience in a context-cognizant manner

1.3.1 Interaction of Quality Attributes

You have probably encountered the matrix presented in Table 1.1 that displays the tradeoffs between choosing quality characteristics. We see that some attributes naturally support each other and some are in opposition. Now, such a table can be easily misappropriated by rigidly remembering the combinations and heralding it as an undisputable truth. Any combination that is taken out of the context is harmful.

For example, according to the table, usability conflicts with security. In most cases, this is true, as any extra security measure is simply an additional burden for a user. A totally secure system would probably be equally useless. Nevertheless, these attributes may also complement each other. If you want to buy a property, then downtown is perhaps more attractive than a suburb; the former is generally more secure and usable than the latter (at least, this is true for my small hometown).

Another example is the conflict of performance with maintainability and portability (as a matter of fact, performance does clash with all of them). Interestingly, a maintainable and portable code is exactly the enabler for performance optimizations. When your code is well structured, then you may alter it in a controllable fashion. Fragile legacy code is “untouchable”; thus, speeding it up is a nightmare. How many times have you seen that some code is a total patchwork in the name of performance? The following famous quote from Knuth D. best describes this phenomenon:

Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%.

Nonetheless, to interpret this statement properly, you must consider the context. It isn't enough to only memorize “*...premature optimization is the root of all evil*” and mechanically apply it everywhere. Otherwise, you will miss that 3% of the critical cases.

1.4 Maintainability and Development

Sometimes, maintainability is planned from the very beginning and properly specified, monitored, and managed throughout development. This is the case with mature organizations that are fully aware of the benefits of having an evolvable product. Nonetheless, most organizations neglect maintainability as something that isn't important during software development. Consequently, the resulting product lacks proper tests and documentation and is usually incomprehensible. This negatively impacts future maintenance work and thwarts change management (it lowers a maintenance team's ability to perform consistent impact analysis). Chapter 3 tries to prove the opposite, i.e., to showcase that investing into maintainability may speed up development while retaining high overall quality. Moreover, it will exemplify how safety and maintainability underpin each other. Based on my experience, most companies assume that maintainability is an afterthought of development (a similar blunder is often repeated with security too). Unfortunately, what happens is that an original schedule is missed due to lots of rework, and a botched product remains a constant pain for a maintenance group.

Maintainability is one of the many quality attributes included in the ISO/IEC 25010 standard. As such, it is under the jurisdiction of the software quality area. Quality is achieved through *assurance* and *control*. The former proactively incorporates quality aspects into a work, while the latter tries to ensure that the outcome of a work is of an acceptable quality. Companies that disregard maintainability in the name of speed and productivity habitually do that by minimizing quality assurance efforts. Since measuring productivity is a very difficult task, they have only a vague idea what productivity means. Naturally, they don't catch a situation when most time is spent on fruitless rework (including bug fixing marathons). For them, fixing bugs fast is also associated with their notion of productivity.⁹ Chapter 3 illustrates this condition with an example, where the time for reasoning about a safety aspect of a maintainable code is much less than in the case of an ill-structured variant. This also reduces the chance of shipping a product with an unacceptable level of defects.

⁹It is like pushing a car with a broken tire to simulate progress, instead of stopping and repairing it before continuing again with full speed. Any sort of rework is a waste, and doing more rework swiftly is nothing else than pushing a broken car faster.

1.5 Architecture and Evolution

You've probably heard about the *Big Design Up Front* (BDUF) anti-pattern promoted by some Agile advocates. Some even go further and claim that the best architecture emerges by itself during development. Maybe BDUF isn't always warranted, although safety-critical and mission-critical software cannot be done without producing first the full specification, but the second part is incorrect. The architecture must be carefully crafted and heralded as the core building block of a system. Many Agile methods now incorporate and promote the idea of having a couple of up-front iterations to produce an initial version of the architecture.

This chapter will exemplify through a case study the importance of an architecture regarding evolution. When changes start to rapidly arrive, the only barrier between controlled activities and chaos is the architecture. It gives feedback to the team what is good, where to improve things, how to extend the system in a natural way, and what is clearly out of scope. Most importantly, it allows software engineers to reason about the system without building it.

1.6 Ad hoc Development

This chapter will demonstrate what happens when development and later, evolution are done without an architecture, more precisely, how the project is run when the architecture is implicit rather than being explicit. Please keep in mind that every system does possess some architecture. However, if it is buried and invisible, then nobody can rest assured that whatever the system does is done well.

The actual story of this chapter isn't real, but the sequence of events is truthfully reflecting what I've seen as a pattern during my career. It is simply amazing how many large software companies are following the style demonstrated in this section.

1.7 Disciplined Development

Here, we will demonstrate exactly the opposite compared to the previous situation (see Sect. 1.6). Even though Chap. 4 does illustrate the central role of an architecture, in this chapter we will develop a more extensive example. We will have a system comprised of independent components and will see how we may evolve data too. Databases (irrespectively of which type) contain lot of valuable data that cannot be simply dropped after each release. Therefore, we are going to delve into the details of data migration using appropriate technologies.

The aim of this chapter is to cement in the truth that the more we spend on up-front design pertaining to the architecture, the less we will struggle afterward and we eventually speed things up. This is a lot like the situation with risk management. It does introduce some extra planned work but considerably reduces the amount of unplanned rework. Overall, the total running time of a project is shorter and the experience better compared to the case without any risk handling.

1.8 Reengineering and Reverse Engineering

Inevitably, a time will come to touch a legacy system, which isn't usually a pleasant experience. However, the term *legacy* doesn't always imply an old and huge monster. Sometimes, an ill-developed system becomes legacy at the very first day after its release. This happens when it was developed in a fashion as demonstrated in Chap. 5. Nevertheless, in this chapter, we will see what are our options of dealing with a brittle, undocumented, and complex system. We will develop a procedure that you will be able to follow that is lightweight and works in practice.

1.9 Multifaceted Loose Coupling

The notion of the loose coupling was always popular in software engineering. Briefly, it is a metric that expresses the amount of knowledge one component has regarding the *internals* of another one with which it communicates. That *internal* part is a multidimensional entity (code, data, space, time, etc.). This simple view can be applied in various scenarios, starting at language-level constructs (classes, packages, etc.) up to the level of systems in a complex system of systems. We all strive to achieve loose coupling. There are even different types of architectures aiming to systematize the way constituent elements are created and managed: service-oriented architecture (microservices are its special subtype), enterprise service bus, reactive architecture, etc. The problem arises when any of these are treated as a silver bullet, i.e., by just applying a technology or method, we get loose coupling. Of course, there are situations when loose coupling is disadvantageous or creates additional problems that wouldn't exist in the case of tight coupling. For example, executing a compound transaction locally is surely easier than if those components are distributed over the network. Here, loose coupling in the space dimension does create novel failure cases that we must address.

Many think that if we achieve loose coupling in design (e.g., by applying some form of a service-oriented architecture), then we will automatically get loose coupling in deployment too [17]. Quite often, the restructuring endeavor ends up transforming a big ball of mud (a monolith) into a big pile of small muds (tightly coupled services). The corollary is that both still must be handled in an or-all-nothing manner. No miraculous technology would help here. We will see how to avoid this pitfall and pave the way to leverage sophisticated deployment technologies.

1.10 Monitoring and Logging

When something fails in production, a usual reaction is that it was caused by some programming error. It is true that many defects are created during the construction phase. However, classical programming errors are just a small part of the whole picture. Failures may happen even if the system was perfectly built according to the specification, although specification errors are still the main culprit. At any rate, to be able to react appropriately, you must have a way to trace events in the system and use the collected data for various root cause analyses.

This section will also expound on supervising a distributed software system. In such a system, there are a myriad of places where something can go amiss. As a matter of fact, a failure in a distributed system should be treated as a natural phenomenon that will occur sooner or later. The basis for handling failures and improving resiliency revolves around observability. This chapter will introduce techniques to monitor the system and log pertinent state change events.

1.11 Scale of Deployment

As quality is the responsibility of all participants in software development and maintenance, the same is true for operations. We cannot just hand over our deliverables to a different team, who will then deploy it via a complicated manual process and operate it in production. The days of huge installation manuals should be behind us. The reasons are manifold:

- A manual process requires specialization and isn't scalable. It completely prevents autoscaling of a system during execution.
- A manual process is usually targeted for the production environment and cannot be readily applied on a developer's machine.
- It is hard to keep the manuals up to date with the latest patches to the operating system or application, so most of the time they are stale.
- It is nearly impossible to replicate the exact production setup based upon manual installation at some later time. For example, if the manual states “execute `sudo apt-get update && upgrade`”, then the effect of this line will surely change over time compared to the date/time when it was done during the original installation. An ability to faithfully mirror the production conditions is sometimes crucial to reproduce problems.

We will see different options how to automate the deployment, and we will wrap up our voyage revolving around the DevOps paradigm. This section will explain the different scales of deployment (virtual machines, containers, etc.) and showcase some modern technologies in this area.

1.12 Exercises

1. This is a challenging self-study task. Try to understand the `java.security` package together with the entry `SecurityManager` class, i.e., to decipher the Java's privilege model. What is the purpose of the `doPrivileged` method of the `AccessController` class? Can you reverse engineer the big picture by purely browsing through the technical details of the Java Security API? Have you ever tried to get more insight about what is going on? Now, read about the extended Bell-LaPadula model with selective “declassification” of data (see [8]). It is the theoretical model of Java's security system. After acquiring the big

picture, revisit the Java Security API. Do you now look onto it from a different angle? Can you spot security issues in your existing code?

2. This is another self-study task. Examine the collection of well-known case studies of software failures (visit <http://www.cse.psu.edu/~gxt29/bug/softwarebug.html>). These should remind you, that failures may happen for many reasons, which have nothing to do with typical coding errors.

References

Further Reading

1. Tripathy P, Naik K (2014) Software evolution and maintenance. Wiley, Hoboken, NJ. This book gives you a theoretical background about the software maintenance and evolution knowledge area. It explains all of its major parts, and expands on [2]

Regular Bibliographic References

2. Bourque P, Fairley RE (eds) (2014) Guide to the Software Engineering Body of Knowledge, Version 3.0. IEEE Computer Society, Piscataway, NJ. www.swebok.org. Accessed 04 Jul 2017
3. Baldwin D (2011) Is computer science a relevant academic discipline for the 21st century? *IEEE Comput Soc* 44:81–83
4. Varga E (2011) The holistic approach to software engineering. 5th international conference on engineering and technology ICET-2011, Phuket, Thailand
5. Savić V, Varga E (2017) Extending the SEMAT Kernel with the TDD practice. *IET Softw.* <https://doi.org/10.1049/iet-sen.2016.0305>
6. Lehman MM (1980) Programs, life cycles, and laws of software evolution. *Proceed IEEE* 68:1060–1076
7. Fowler M (2003) TechnicalDebt. martinfowler.com/bliki/TechnicalDebt.html. Accessed 06 Jul 2017
8. Anderson RJ (2008) Security engineering: a guide to building dependable distributed systems, 2nd edn. John Wiley, Hoboken, NJ
9. Denning PJ (2011) The grounding practice. *Commun ACM* 54(12):38–40
10. Jones C (2006) The economics of software maintenance in the twenty first century – version 3. www.compaid.com/caiinternet/ezine/capersjones-maintenance.pdf. Accessed 16 Aug 2017
11. Green R, Ledgard H (2011) Coding guidelines: finding the art in the science. *Commun ACM* 54(12):57–63
12. Jacobson I, Meyer B (2009) Methods need theory. www.drdobbs.com/architecture-and-design/methods-need-theory/219100242. Accessed 18 Aug 2017
13. Jacobson I, Spence I (2009) Why we need a theory for software engineering. www.drdobbs.com/architecture-and-design/why-we-need-a-theory-for-software-engine/220300840. Accessed 18 Aug 2017
14. Bennett K, Rajlich V (2000) Software maintenance and evolution: a roadmap. www0.cs.ucl.ac.uk/staff/A.Finkelstein/fose/finalbennett.pdf. Accessed 18 Aug 2017
15. McConnell S (2011) Origins of 10X – how valid is the underlying research? construx.com/10x_Software_Development/Origins_of_10X_-_How_Valid_is_the_Underlying_Research_.pdf. Accessed 19 Aug 2017
16. Sommerville I, Cliff D, Calinescu R, Keen J, Kelly T, Kwiatkowska M, McDermid J, Paige R (2012) Large-scale complex IT systems. *Commun ACM* 55(7):71–77
17. Pardon G (2008) Loosely-coupled deployment vs loosely-coupled design. www.atomikos.com/Blog/LooselyCoupledDeploymentVsLooselyCoupledDesign. Accessed 23 Aug 2017

In a hectic desire to comply with corporate standards, people blindly follow rules. On one hand, it is a mechanism to escape responsibility, as whatever happens, there is this “*I’ve followed the rules!*” magic statement. Nonetheless, rules devoid of a proper context can be as bad as pure disobedience. They tend to transform into a dogma. A software engineer cannot behave like a context-free language parser, i.e., merely recalling facts and opportunistically applying them everywhere. Such a behavior, according to my experience, is the number one culprit for destroying efficient distributed teamwork, hence causing all sorts of quality problems. Context awareness is like defensive programming, in a sense that it creates a powerful feedback loop-and-control mechanism to reinforce or reject an action or idea. When talking to a dogmatic person, you must provide all directives in advance and predict future variations in the context. Otherwise, the conversation will soon start with “*You’ve told me to do...*” and continues like “*Yes, but you should have known that. . .*” This chapter explains the interplay between context and knowledge (partly embodied as a set of rules and principles) as well as gives examples of how things can turn upside down when actions are thoughtlessly executed. The aim of this chapter is to demonstrate the following concepts: Context, Rules, and Principles.

Story 1 An employee of a large software company was routinely writing large C programs by packing everything into the main procedure. After seeing this, the management tried to remedy the problem by advising him to move out stuff from main into subordinate procedures. In that way, he would have had better complied with the tenets of structured programming. The result was astonishing. He simply created a new procedure `dowork`, put everything there, and called it from `main`. From his viewpoint, this was exactly the management’s demand. *This is the case where a total lack of knowledge prevents someone from deciphering the context.*

Story 2 Shortly after a company had shipped a new version of the product, a customer had reported a critical defect. The first thing that needs to be done in this situation is to understand what really happened. Log files are a very useful source of

information for getting clues about events preceding a failure. At least, this is an expectation regarding logs. In our case, the logs were strangely thrashed with Skipping point i, j sequences seriously thwarting someone's ability to browse them. Considering the latest changes, the next code had popped up on the radar (given here in an abridged form):

```
public void processImage(byte[][] image) {
    try {
        for (int i = 0; i < App.MAX_HEIGHT; i++) {
            for (int j = 0; j < App.MAX_WIDTH; j++) {
                // access image[i][j] and process it
            }
        }
    } catch (IndexOutOfBoundsException e) {
        // log the 'Skipping point i, j' "informational" message
    }
}
```

A developer had simply relied on Java's array bounds checking facility to control the flow of execution (see Sect. 1.6 for more discussion). The fancy log message was there probably for his entertainment. *This is an example where being oblivious of the other contexts (like a production run) as well as lacking Java programming skills may cause harm.*

Story 3 A developer was trying to synchronize his code execution with an external device. There were some strict timing constraints mandated by a device. The `Thread.sleep` method cannot guarantee that the corresponding thread will be wakened and scheduled for execution exactly after the specified timeout. It works on a best effort basis. To improve the accuracy, the developer had set his thread to run on top priority. Seemingly everything was proper, as the timing was good enough. Sometime later (after a couple of more releases), the communication with a device had become unreliable. The first inkling was that something went wrong in the latest release. In some sense, this was true, although it was not a complete picture. In the last release, another guy also leveraged the same problematic tactic and had increased his thread to a maximum priority (this time it was some user interface thread). Naturally, this action has changed the context. The original selfish word view has evaporated. *This is an example where being ignorant of the fact that the context is dynamic and lacking Java programming skills may cause harm.*

Story 4 A company had developed an internal transaction processing framework. It was reused in many products without any issues. A new project was initiated, and it also relied upon that framework. However, one developer started to struggle with his code, as it simply behaved unreasonably. The framework had a feature to call back into a user's class using Java's reflection mechanism. It looked for the public method `executeTransactionally`. This method would contain all business

logic to be wrapped inside a transaction. As it turned out, the framework also called back some “similar” method in a user’s class (in this case, it was `executeTransfer`), when it was not able to find the exact match (if the method’s signature properly matched). This ingenious addition was motivated by a fact that many times, developers misspelled the `executeTransactionally` method. So, the framework’s author thought that it would be cool to incorporate some cleverness into the framework’s engine. Of course, this extra feature was nowhere documented, nor was the “similarity” search confined to some reasonable range. *This is an example of being inconsiderate by sneakily imposing your context onto the others.*

The previous four stories are just a miniature sample of similar stories that I have experienced so far in my practice. There are three broad scenarios pertaining to context and knowledge interaction, as depicted in Figs. 2.1, 2.2, and 2.3. Figure 2.1 shows the case when the other party doesn’t care about the context and mechanically performs actions as instructed. Figure 2.2 shows the case when a lack of knowledge prevents someone to spark discussion to decipher the right context. Both situations are detrimental and are exemplified further in this chapter. Figure 2.3 illustrates the sunshine scenario, where both parties are knowledgeable and context aware (see references [1, 2] to deep dive into context awareness). This is the cornerstone of a successful software maintenance and evolution process. Being able to formulate the right questions is important for two reasons: you will get the right answers to avoid mistakes, and you will be more efficient and succinct in communication. The latter is especially prominent during maintenance, when the original development team is busy on other projects. You would not want to bother them with trivia nor bombard them with vague inquires.

2.1 Lack of Context: Problem

The idea for this worked example is taken from my article [3], which also illustrates how a context may adorn bare binary numbers (crunched by our machines) with additional meaning. The management had issued a decree that every developer should analyze the code base and find ways to simplify the implementation. The goal was to reduce the overall cyclomatic complexity, as it was above some threshold (the customer had stated that they would refuse to accept the software in this condition). The company’s rule for unacquainted programmers was simple: just seek to eliminate as many `if` statements as possible.

McCabe’s cyclomatic complexity is an important quality metric. Lower values are preferable, as they imply a straightforward code. During maintenance and evolution, a considerable effort is spent on understanding the code, so any action to reduce its complexity aids readability and comprehensibility. Cyclomatic complexity basically measures how many independent paths of execution are present in the code base. Therefore, it is tightly associated with the number of branching statements. A lower number also helps testing since a greater code coverage may be achieved with fewer tests. Nonetheless, as with any metric, it can be used or abused.

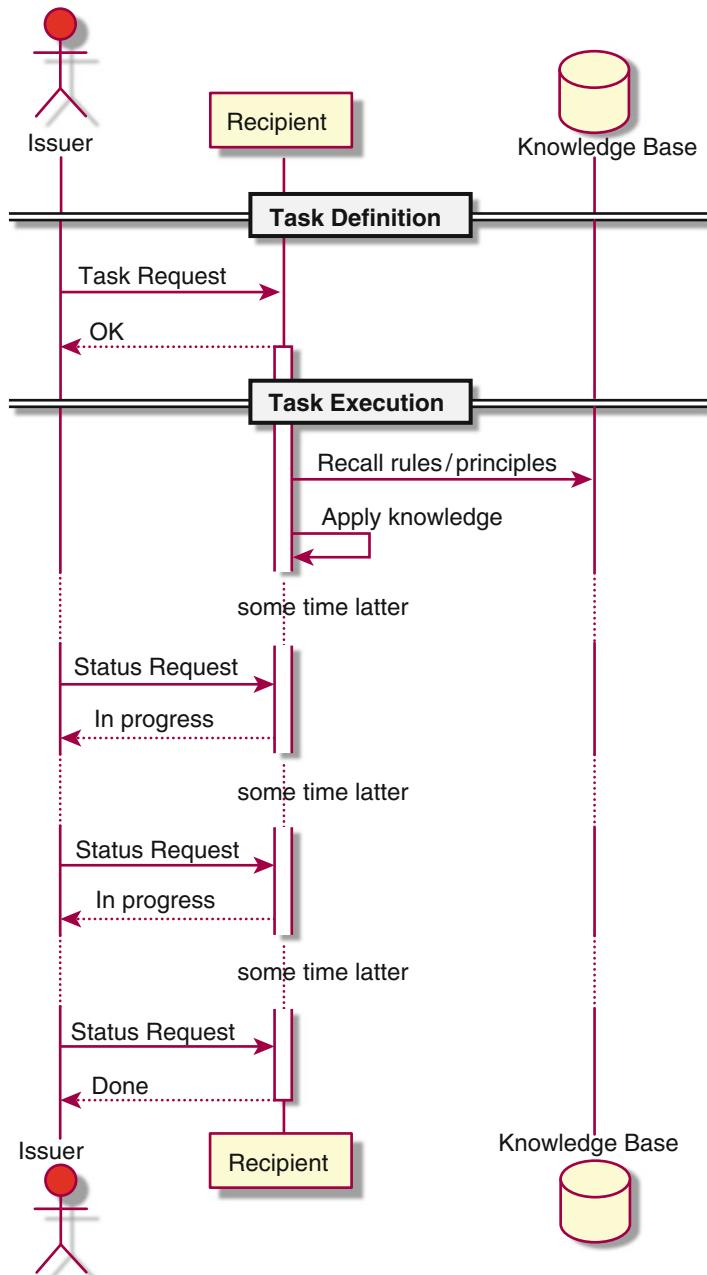


Fig. 2.1 The recipient will literally fulfill all instructions. You need to ensure that all actions make sense in any given situation. The feedback loop is silent. The issuer must hope for the best after getting the “Done” response. Another alternative is excessive micromanagement that is equally troublesome for both parties

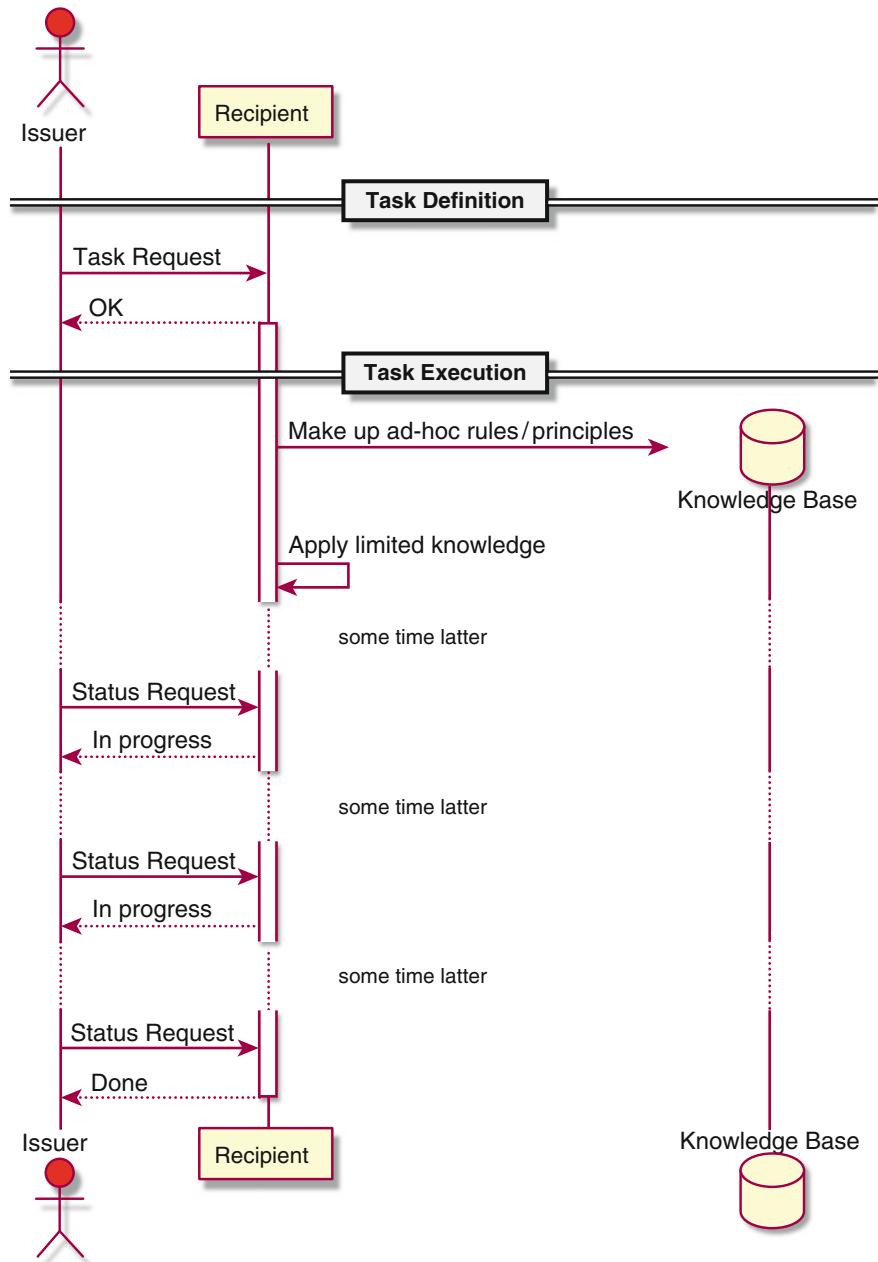


Fig. 2.2 The sequence of interactions is a lot like that in Fig. 2.1. The difference here is that the recipient will make up ad hoc rules/principles, as he judges appropriate

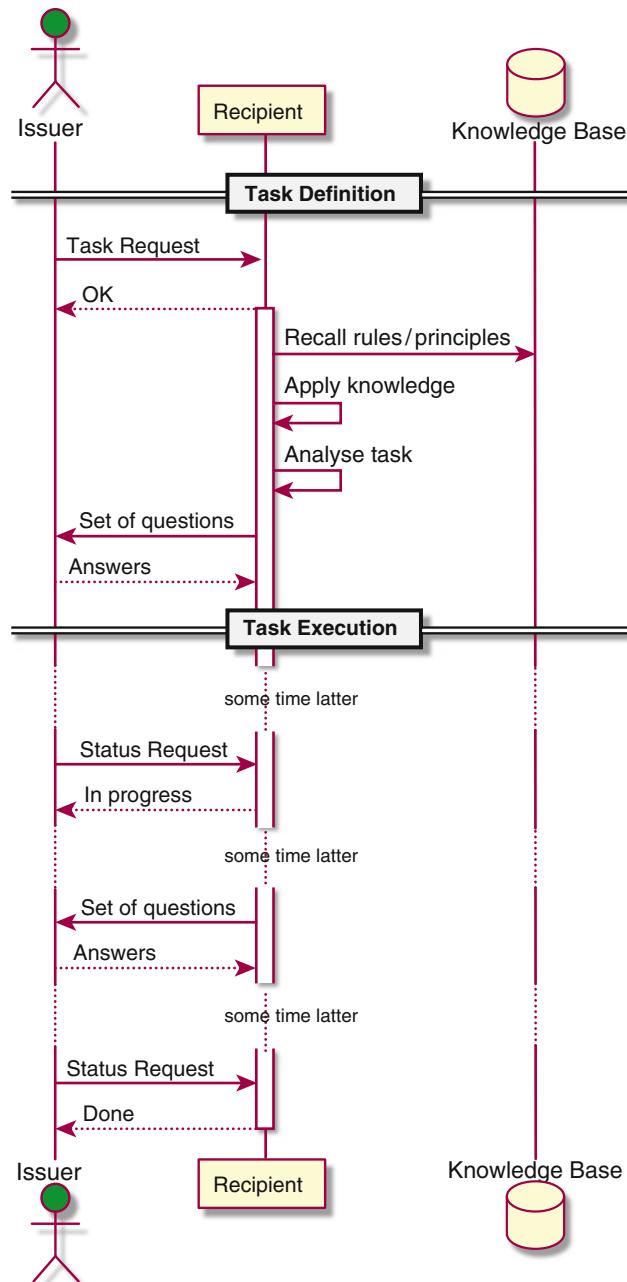


Fig. 2.3 Both actors are knowledgeable and context aware. Consequently, the feedback loop is active during the whole process. The issuer may count on being proactively contacted if something pops up, requiring attention. The outcome is as expected by the issuer. There is less or no need for micromanagement

2.2 Lack of Context: Resolution

A developer had stumbled across the following method to sort an array of doubles into an ascending order¹:

```
public void sort(double[] data) {  
    for (int i = 0; i < data.length - 1; i++) {  
        for (int j = i + 1; j < data.length; j++) {  
            if (data[i] > data[j]) {  
                double temp = data[i];  
                data[i] = data[j];  
                data[j] = temp;  
            }  
        }  
    }  
}
```

Obviously, this is a very poor implementation of a sort algorithm. However, speeding up the system was not a target. Therefore, he had concocted an ingenious variant of this sorting method, which satisfied the vision of reducing the total cyclomatic complexity. After all, that single `if` statement has indeed disappeared from the above code. Here is the new version:

```
public void sort(double[] data) {  
    for (int i = 0; i < data.length - 1; i++) {  
        for (int j = i + 1; j < data.length; j++) {  
            double avg = (data[i] + data[j]) / 2.0;  
            double diff = Math.abs(data[i] - avg);  
  
            data[i] = avg - diff;  
            data[j] = avg + diff;  
        }  
    }  
}
```

This case study may seem very strange, but such dogmatic problem “solving” approaches are quite frequent. Just recall how many times you were forced to “improve” the tests to pass the company’s code coverage threshold. I’m not asserting that a high limit is bad per se. The issue arises when reaching that value by any means becomes the sole goal.

¹You would not want to code up such a sort method, as basic sorting is part of the Java collections framework.

Metrics in general are a useful mechanism to achieve desired project goals, as they may steer the team. Nevertheless, there are many pitfalls regarding metrics, and this case study has demonstrated two of them: making a metric a goal by itself and using only a single metric to improve things. When a metric is used in isolation and out of context, software engineers will eagerly strive to improve the value of the metric. This usually leads to bad side effects that are more severe than the “improvement” alone. A software system is too complex to be described in a single dimension (by using a sole metric). Therefore, the goal/context must always be properly understood, and multiple metrics utilized to balance the outcome. Of course, myriad of metrics are another extreme that should be avoided (for more details and a nice explanation of pitfalls to avoid when using metrics, read [12]).

2.3 Lack of Knowledge: Problem

This is a situation that I had encountered with one of my student. There were two classes (`Payment` and `BonusCalculator`), where `BonusCalculator` was an abstract root entity of the *strategy* hierarchy (application of the *Strategy* design pattern from [4]). So, `Payment` was customized with a concrete strategy as depicted on Fig. 2.4. A client programmer of the `Payment` object wanted to calculate a total payment that erroneously wasn’t available via its public API. The `Payment` class only allowed the retrieval of the concrete `BonusCalculator` object, which contained an alluringly named `calculate` method. I have asked my students for a remedy.

2.4 Lack of Knowledge: Resolution

One student immediately recommended a modification to the class diagram as shown in Fig. 2.5. This change was terrible. First, it had broken the very powerful low coupling principle in the worst possible manner by introducing a circular dependency between classes. Moreover, the exposed `calculate` method wasn’t even the one a client programmer wanted to call. As a matter of fact, the `BonusCalculator` object wouldn’t even know what to do with the `Payment` instance.

If you simply break a rule/principle without even considering the ramifications, then you will surely ramble onto a wrong path. Of course, being oblivious of the existence of a rule/principle isn’t an excuse, as you would just equally commit the same sin. Being fast but wrong is surely much worse than being slow but considerate.

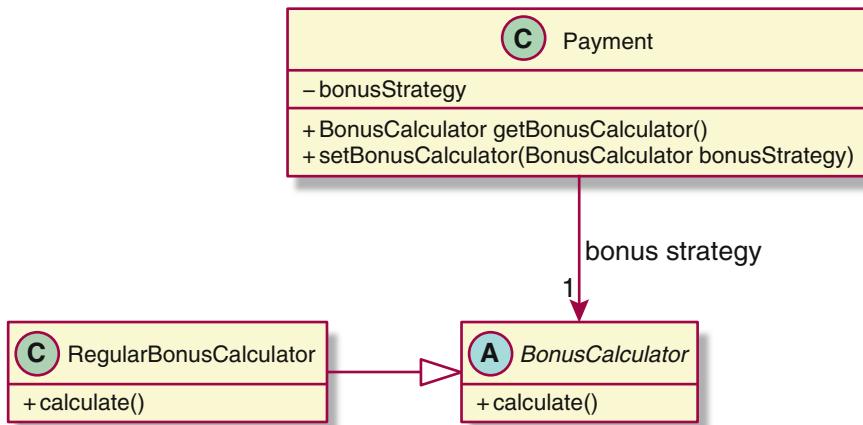


Fig. 2.4 The UML class diagram depicting the original problem. A client programmer cannot call the desired method

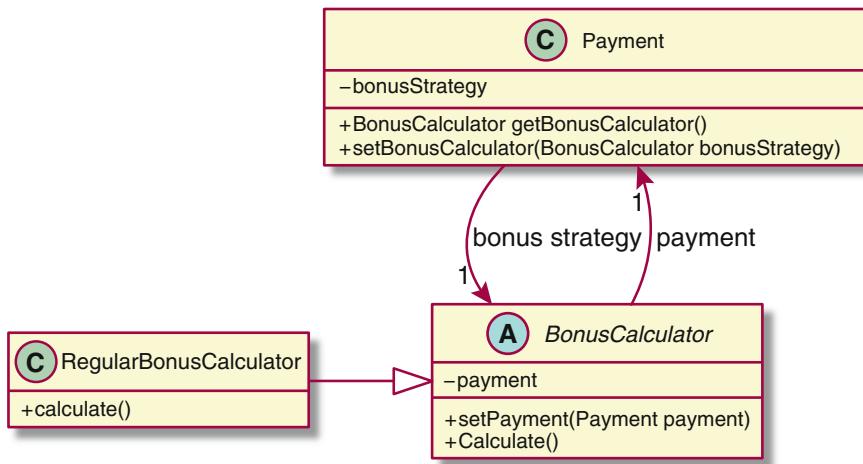


Fig. 2.5 The proposed change to “fix” the issue from Fig. 2.4. The Payment object would inside the setBonusCalculator method call the setPayment method, and pass the `this` reference. A client programmer would then just use: `payment.getBonusCalculator().calculate()`

2.5 Summary

It is impossible to supersede human creativity, although it is fascinatingly simple to paralyze it. Software engineering is an excruciatingly hard intellectual endeavor, and losing cleverness is a path to a disaster. Forcing people to obey rules out of context is a sure failure scenario for a company. As a consultant, I have witnessed more sins made in the name of following corporate canons than anything else related to software development.

Context is the cornerstone of any software engineering activity. The way we communicate and present/process information is vital for obtaining maintainable and evolvable solutions. You may want to read my article [5] about the role of stories in conveying important messages. This article also gives a short case study about the dangers of overbloating the context, i.e., coming up with ill grounded expectations. My blog post [6] demonstrates how a puzzle-based learning approach may help in conveying a message as well as its associated context.

A principle is stable and embodies a core truth, while a tactic is volatile and of a shorter life-span. Nevertheless, rules and principles aren't meant to be blindly followed. Their essential purpose is to prevent you departing from well-established paths in a negligent manner, i.e., to put you into a hard situation when explaining the motive for breaking them. There is even a so-called *deviation principle* that formalizes exceptional conditions. The deviation management, which is part of the continuous code inspection process [7], is based on a strict agreement about how to review and document any departure from known principles/rules.

The key takeaways from this chapter are as follows:

- Rules and principles devoid of context are pure dogmas.
- Rules and principles always operate under some context, even if it is implicit.
- Knowledge is important to recognize and leverage a context.

2.6 Exercises

1. You are given the task to improve the cohesion and the clarity of methods. The management has instructed you to reduce the number of parameters for methods. It is a well-known fact that methods with high numbers of input parameters are susceptible of doing different functionalities (a.k.a. *code smell*). Assume that you've stumbled across the following log analyzer class:

```
public class LogAnalyzer {  
    // Constructor(s)/fields omitted for brevity  
    public int logCount(long startTime, long endTime) {  
        ...  
    }  
    public int logCount(long startTime, long endTime,  
        Level logLevel) {  
        ...  
    };  
    public int logCount(long startTime, long endTime,  
        String message) {  
        ...  
    };  
}
```

Your team member has proposed the following solution:

```
public class LogAnalyzer {  
    private final long startTime;  
    private final long endTime;  
    public LogAnalyzer(long startTime, long endTime,  
                       /* other parameters */) {  
        super(...);  
        this.startTime = startTime;  
        this.endTime = endTime;  
    }  
    // Other constructor(s)/fields omitted for brevity  
    public int logCount() { ... }  
    public int logCount(Level logLevel) { ... };  
    public int logCount(String message) { ... };  
}
```

He has also explained to you that this new version should be used in a slightly different manner. You will need to create a new object for every unique combination of time parameters. Previously, a single instance would do the job for various inputs. What do you think? After all, the goal is accomplished, i.e., the number of parameters for methods is indeed reduced. (Hint: see [12]).

2. How would you avoid indexing outside of the input array's boundaries in the `processImage` method (when the dimension of the image is smaller than what is maximally allowed)? Assume that the image has a varying second dimension (hint: look at the `length` property of a Java array). An Exception class (or its subclass) should designate an exceptional condition, as its name implies. Controlling a flow of execution via the exception mechanism is essentially a disguised `goto` programming approach (with an extra performance penalty).
3. Include all the drawbacks of the modified version of the `sort` method relative to the original one. Is it really the case that the new version is more understandable (even though it has a lower cyclomatic complexity)? What can you conclude from this (hint: think about the direction of the implication between complexity and its metric)?
4. Can we unanimously state that the Bubble sort is deprecated or that the Branchless sort is utterly useless? What if we consider a not so perfect world, where component failures do occur (e.g., a malfunction in the comparator unit)? It turns out that the Bubble sort is more robust regarding errors popping up in the comparator unit than the Quick sort (see [8] for a very detailed explanation). The Branchless version can work despite the comparator being completely out of order. Bringing in redundancy in the system is an effective way to increase fault

tolerance. However, with software the variation must be fundamental. For example, a hybrid sort comprised of three algorithms (Quick, Bubble, and Branchless), coupled with a decision unit, may be a powerful combination. There is a well-documented flight failure of an Ariane 5 rocket [9], where two redundant hardware components were running the same software (they both had failed for the same reason in a very short period of time). At any rate, by varying the context, we come up with diverse conclusions about the viability of some approach.

5. Come up with an acceptable fix to the class diagram in Fig. 2.4? Is simply introducing a new public method, `calculate`, in the `Payment` class a no-brainer? Is it possible to break client code by this action (hint: assume that the `Payment` class isn't `final`)?
6. (For group discussion) Compare and discuss how the communication patterns have changed over time, and find any resemblance with the way developers work today (see [11] for an impact of computer technology on personality). In the past, when writing and sending a message was very expensive, people were forced to thoroughly think about the content. Nowadays, they exchange a barrage of cryptic short messages, with lots of grammatical and typing errors. None of the messages are self-contained nor follow some well-defined path. When developers implement "something" with modern IDEs, the pattern is very similar. They usually come up with something that is rarely the right thing. One of the cornerstones of building evolvable solutions is to properly think about the problem. Software engineers should resist the temptation to frantically experiment about the solution (for a nice recap see [10]).

References

Further Reading

1. Petzold C (2000) Code: the hidden language of computer hardware and software. Microsoft Press, Redmond, WA. This book is all about the socio-technical context surrounding computers and computing/communication. It starts from the early periods of a human history, and analyses our need to communicate. The book shows how different theoretical and practical solutions were built on top of each other until it arrives at a modern computer architecture; it sets up a quest to reach the roots regarding the established binary number system
2. English edition: Dostoevsky F (2008) The idiot (trans: Martin EM). [Digireads.com](#) Publishing, New York, NY. This book is a masterpiece in the domain of human psychology (F. Nietzsche, who rarely gave compliments, called Dostoevsky "*the only psychologist from whom I was able to learn anything.*"). Through the story, that depicts the never-ending battle between good and evil, the book challenges a reader to tune in to detect even the most delicate contextual variations. Such an ability is crucial for becoming a good communicator.

Regular Bibliographic References

3. Varga E (2016) Simplicity can't be attained greedily. www.linkedin.com/pulse/simplicity-cant-attained-greedily-ervin-varga. Accessed 25 Jun 2017
4. Gamma E, Vlissides J, Johnson R, Helm R (1994) Design patterns: elements of reusable object-oriented software. Addison-Wesley, Reading, MA
5. Varga E (2016) Instead of rules tell a story. www.linkedin.com/pulse/instead-rules-tell-story-ervin-varga. Accessed 25 Jun 2017
6. Varga E (2016) Do APIs matter? www.apress.com/gp/blog/all-blog-posts/do-apis-matter/11524110. Accessed 25 Jun 2017
7. Weimer C, Bolger F (2017) Continuous code inspection: advancing software quality at source. PRQA Programming Research, Surrey
8. Ackley DH (2013) Beyond efficiency. Commun ACM 56(10):38–40
9. Lions JL (1996) Arian 5 flight 501 failure – report by the inquiry board. www-users.math.umn.edu/~arnold/disasters/ariane5rep.html. Accessed 18 Aug 2017
10. Armour PG (2013) When faster is slower. Commun ACM 56(10):30–32
11. Holmes N (2011) The profession and digital technology. IEEE Comput 44(12):114–116
12. Bouwers E, Visser J, Deursen AV (2012) Getting what you measure. Commun ACM 55 (7):54–59

During development and maintenance, software engineers spend most of their time reasoning about the system. To speed up this activity, we need to employ a proper architecture and straightforward implementation. Let us see how the latter helps us craft a safety claim about the system. We would like to show how maintainability as a quality attribute directly supports safety. Arguing about safety demands a thorough understanding about how our software works. The next exercise (although a bit altered) is taken from [4]. We provide here one possible solution with an aim to demonstrate the following concepts: Single Level of Abstraction, Layered Design, Cyclomatic Complexity, Safety Argument, and Programming in Intentions.

3.1 Safety Argument: Problem

Suppose that you need to analyze a given program that controls the door lock mechanism in a nuclear waste storage facility. Obviously, this is a safety-critical software since any failure may have dire consequences on the environment and human operators. The lock must ensure that entry to the storeroom is only permitted when radiation shields are in place or when the radiation level in the room falls below the threshold (this is a configuration parameter called `dangerLevel`). There are two rules governing the operation of a lock:

- A door may only be operated by some authorized personnel identified by an entry code. This is a security aspect underpinning the safety concern.
- A door may only be opened if remotely controlled radiation shields are in place within a room or the radiation level in a room is below the threshold.

Below is an excerpt from the door lock controller written in pseudocode. The `Lock` class manages the input of an entry code, `Shield` is responsible for handling the protective shields, and `RadiationSensor` collects the radiation level in the room. The `Door` class locks/unlocks the door. Assume that all these classes are

singletons (see [5] for more details about the *Singleton* pattern) for the sake of simplicity. Moreover, presume that the door is locked at the beginning:

```
int entryCode = Lock.getEntryCode();
// BRANCHING 1
if (entryCode == Lock.AuthorizedCode) {
    int shieldStatus = Shield.getStatus();
    int radiationLevel = RadiationSensor.getLevel();
    boolean safe;
    // BRANCHING 2
    if (radiationLevel < System.dangerLevel)
        safe = true;
    else
        safe = false;
    // BRANCHING 3
    if (shieldStatus == Shield.InPlace)
        safe = true;
    // BRANCHING 4
    if (safe)
        Door.unlock();
    else
        Door.lock();
}
```

3.2 Safety Argument: Resolution

Traditionally, you verify¹ your code by running tests and reviewing it (either alone or with the help of a colleague) to eliminate as many faults as possible. None of these activities can prove that the code will not experience a failure in production.² For a safety-critical software system, such a condition is extremely dangerous. You cannot rely on users reporting an issue and later fixing it in the form of a patch. A safety-critical software system must avoid entering an unsafe state despite failures.

When building a safety argument for the above code, we want to formally structure facts and arguments to support the claim that it is safe. Figure 3.1 shows

¹Verification is the process of ensuring that you properly follow the system specification, hence building the product right. Validation is the complementary process of assuring that your specification is solving the right problem, i.e., building the right product. We will later challenge the given specification and show that it isn't covering all aspects of the domain. Missing important details may be detrimental (for a real case study, see the Warsaw Airbus accident in 1993 at <http://iansommerville.com/software-engineering-book/videos/reliability-and-safety/>).

²This example seems too uncomplicated to warrant a formal verification; however, a more complex example would cause too much detraction.

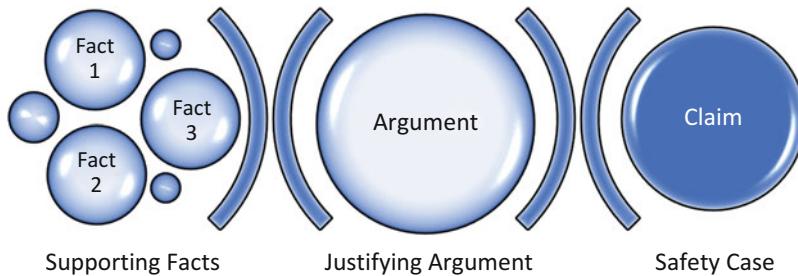


Fig. 3.1 The structure of a safety claim hierarchy. Evidences (facts) support the argument, which justifies the safety claim. All these comprise the overall safety case of a system

this structuring. Of course, we need to know what is the safe state. In our case, it is that entry is forbidden. To demonstrate that our software cannot reach an unsafe state, we will suppose the opposite (a proof by contradiction) and show that we will hit a contradiction for each execution flow. The previous listing contains commented branching points, which represent specific facts about the execution. Figure 3.2 illustrates that we may superficially treat this code as safe (see the Exercises section for more details). Safety arguments are weaker than correctness arguments. We only try to show here that the system cannot reach an unsafe state but still cannot be sure that it fully conforms to the specification.

Why is Fig. 3.2 so convoluted (even for this short piece of code)? The answer is simple: because we have many execution paths to scrutinize. Cyclomatic complexity is the metric associated with the amount of decision points in the code, i.e., it reflects the number of linearly independent paths in the program. A higher value is considered bad. It is calculated from the control flow graph of the program using the formula $E - N + 2P$, where E is the number of edges, N is the number of nodes, and P is the number of nodes with an exit point. In our case, with a single entry/exit point as well as noncompound branching conditions, the formula may be simplified to $B + 1$, where B is the number of `if` statements. All in all, for the above code it is 5. Most quality attributes (testability, maintainability, etc.) are negatively impacted by a higher cyclomatic complexity. This is especially true for dependability (availability, reliability, security, and resiliency) characteristics.

Let us find out why our code has such a relatively high cyclomatic complexity. Evidently, it is performing all the following duties and thus must navigate between multiple alternative scenarios using `if` statements:

- Reading the security code and checking whether it is appropriate
- Reading the status of the shields and setting the control variable
- Reading the radiation level from the sensor, checking it against the threshold, and setting the control variable
- Checking the control variable and correspondingly commanding the door to unlock/lock

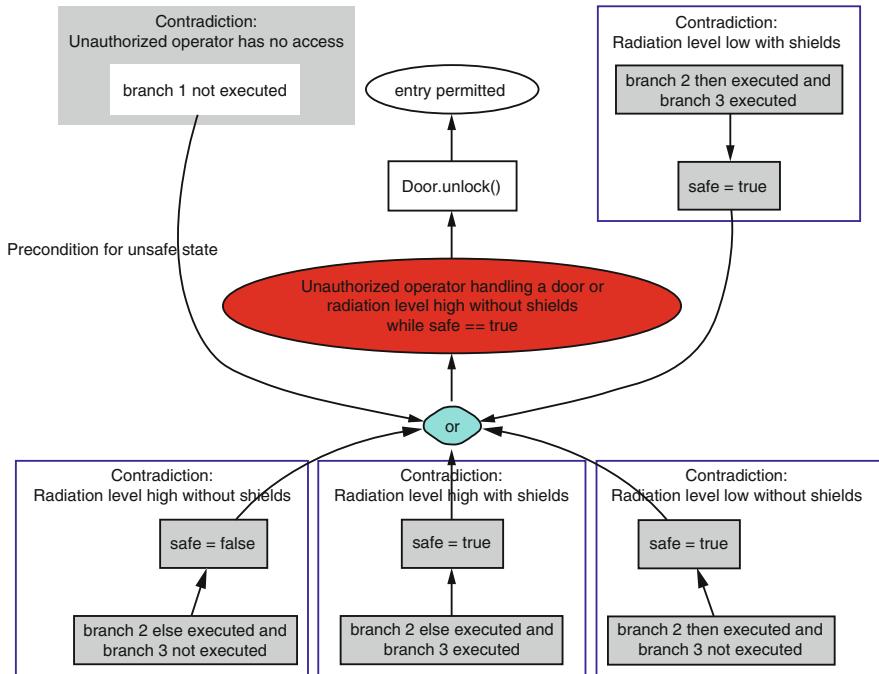


Fig. 3.2 The graphical model of the safety argument. We examine all possible execution paths of the code and show that they all contradict with the unsafe assumption. Therefore, the appropriate assumption is that the system is safe. The red oval designates the precondition for reaching an unsafe state

By careful examination, the above tasks may be grouped into three distinct layers as depicted in Fig. 3.3. The *Device* layer deals with device data and local decisions (e.g., whether the entry code is correct). The *Rules* layer combines individual conditions into a compound safety decision. Finally, the *Action* layer commands the door to unlock/lock depending on the final resolution.

The Single Level of Abstraction principle states that a method should only contain statements at the same abstraction level. This aids comprehensibility and thus increases maintainability. Moreover, if every method does only one thing, then the implementation is straightforward with a low cyclomatic complexity. Below is the refactored door lock control software using these tenets:

```
if (safeToUnlockDoor())
    Door.unlock();
else
    Door.lock();
```

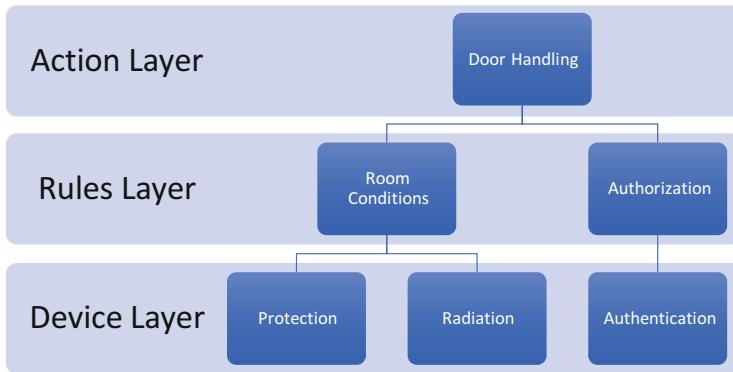


Fig. 3.3 Each layer is responsible for carrying out tasks at that same abstraction level, with each task having a single well-defined responsibility. This arrangement is a consequence of the stratified (layered) design, which is among the most fundamental design principles in software engineering. The general rule is that any layer may only communicate with a layer below (unless it is the lowest layer in the hierarchy). Each layer exposes a well-defined API toward the layer above. In this way, altering a specific layer shouldn't disturb the rest of the hierarchy. The Open Systems Interconnection (OSI) 7 layers reference model is a canonical example for a layered design (any good book pertaining to computer networks will explain it in detail)

```

boolean safeToUnlockDoor() {
    return userAuthorized() && roomConditionsSafe();
}

boolean userAuthorized() {
    return Lock.getEntryCode() == Lock.AuthorizedCode;
}

boolean roomConditionsSafe() {
    return shieldInPlace() || radiationLevelLow();
}

boolean shieldInPlace() {
    return Shield.getStatus() == Shield.InPlace;
}

boolean radiationLevelLow() {
    return RadiationSensor.getLevel() < System.dangerLevel;
}
  
```

The rewritten source code is so intuitive that it, by itself, constitutes the safety argument. Moreover, it leverages various abstraction layers; hence, a developer may incrementally learn the implementation (depending on what level of details is required). For example, if someone wants to judge the higher-level safety rules,

then that person doesn't need to delve down into the Device layer. This was not the case with the original code, where everything was bundled at the same place. Notice that we have also managed to get rid of that artificial control variable `safe`. As a matter of fact, the refactored code doesn't even contain variables. This is an example of a concept known as *intentional programming* [6]. You express your intentions about what your code craves to achieve. Such clear communication helps both the compiler and software engineer in understanding the code. Moreover, by laying out the code in an understandable way, you may keep extra comments on minimum. The code should speak for itself.

3.3 Summary

If you don't mix statements at different abstraction levels inside the same piece of code (for example, in a method) and use layers to delineate those abstractions, then you may produce code with a low cyclomatic complexity. Many scientific studies support the notion that bringing down the cyclomatic complexity (which also generally holds for any other sort of complexity) aids maintenance and evolution. You may have witnessed how the utilization of the principles/techniques from this chapter had increased the readability and comprehensibility of the code. This directly influenced our ability to reason about safety.

None of the principles/techniques/technologies stand isolated from one another. The given context dictates how you should combine them to address the pertinent functional and nonfunctional requirements of a software-intensive system. Software engineers must find a subtle balance regarding how/when/what artifacts to use in a system. In this respect, software engineering is like the other engineering disciplines. A dogmatic approach is our equivalent of an unsafe state in software engineering!

The key takeaways from this chapter are as follows:

- A maintainable code is comprehensible and well structured.
- Maintainability should be the primary focus throughout a software's life cycle.
- Thwarting a code's maintainability must be a judicious and properly managed decision.

3.4 Exercises

1. Create the control flow graph of the original program, and verify that our simplified formula indeed yields the correct result for the cyclomatic complexity.
2. Notice that there are gaps between variable declarations/definitions and their subsequent usage (take `shieldStatus` as an example) in the original program. There is an associated software metric that expresses this distance. Examine the documentation of the open-source tool CheckStyle to learn more

about how it calculates this metric (visit checkstyle.sourceforge.net). Why would you limit the allowed distance via the previously mentioned tool? How hard is it to understand the program when this distance is high (usually, you want to focus on its average value)?

3. There is an adage stating that if something is beautiful, then it should also work, and vice versa. However, you should never be seduced by beauty alone! Despite our effort to refactor the original code, it does contain an insidious safety defect (or more precisely, it contains more than one). Let us focus on the decision to allow entrance when shields are in place. This decision is based on two actions: checking the shield's status and using the retrieved status value to judge safety. What if the shield's status changes in the meantime before we decide to unlock the door? This is a classic example of the problem named *time of check to time of use* and is a special kind of a race condition. Please, remember that reducing the period between the check and the usage doesn't eliminate the issue but only lowers the probability of a failure. How would you remedy this problem?
4. Read carefully the specification, and focus on the section about how shields are operated. It is stated there that they are handled remotely. Is it enough to ensure that only the operator of a door-locking mechanism is authorized? What about the guy operating the shields? This is a fine example, when a security vulnerability may cause a safety hazard, i.e., where security and safety nicely interplay. If shields aren't protected by security measures, then the whole door control system becomes a bare security/safety theater (this quite often happens in practice, especially when such media-buoyed show appeals to a government and commercial sector of a country). Suggest a remedy for this problem.
5. (For group discussion) Suppose that the `then` section of the third `if` statement is wrongly implemented as follows:

```
if (shieldStatus == Shield.InPlace)
    safe = false;
```

Should we still consider the code as safe? Or has it become just less useful? What if during emergency someone needs to enter the room when radiation level is high? Discuss how quality attributes of a system are intertwined (in our case, usability, security, safety, maintainability, etc.), and analyze the different viewpoints that stakeholders have regarding a system. How may priorities assigned to quality aspects cut the Gordian knot? Try to recall a similar mechanism to “untie” the never-ending wishes of a customer (hint: estimation and money).

6. (For group discussion) Think about how a fault-tolerant system design impacts safety. What would happen in the current solution if the radiation sensor or the shield controller would experience a failure? Of course, you don't need to come up with a solution; just spark a discussion, and admire the difficulties that the software industry must solve when creating complex, distributed, and dependable systems of systems.

7. (For group discussion) Think about the connectedness of concepts, taking our small case study as an example. How much has adherence to the Single Level of Abstraction principle helped us satisfy high cohesion and avoid tangling (when a module realizes disparate system requirements)? What about the Open/Closed principle, which advocates that classes should be closed for modification and open for extension? Is there a minimal set of core principles from which we can derive all the others (comparable to the functional complete set of logical operators in Boolean algebra, like a singleton set {NAND})?

References

Further Reading

1. McConnell S (2004) *Code complete*, 2nd edn. Microsoft Press, Redmond, WA. This book is a must read for every software engineer to gain insight into the software construction process. The book has a very detailed treatment of the Single Level of Abstraction principle
2. Martin RC (2009) *Clean code: a handbook of agile software craftsmanship*. Prentice Hall, Upper Saddle River, NJ. This book covers all aspects of agile software development, and has a thorough treatment of a construction process, although it is biased toward Java
3. Martin RC (2011) *The clean coder: a code of conduct for professional programmers*. Prentice Hall, Upper Saddle River, NJ. This book gives the necessary background for programmers to become better team members, and accept the holistic approach of software development. Programmers will understand that development and maintenance/evolution are tightly coupled and interdependent, i.e., a shortsighted, selfish development decision may have serious negative consequences on software evolution

Regular Bibliographic References

4. Sommerville I (2016) *Software engineering*, 10th edn. Pearson, Upper Saddle River, NJ
5. Gamma E, Helm R, Johnson R, Vlissides J (1995) *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, Reading, MA
6. Kamp PH (2012) My compiler does not understand me. *Commun ACM* 55(7):51–53

Architecture plays a central role in software engineering. This chapter will demonstrate through a case study how an architecture enables evolution. We will first revisit two modern definitions of an architecture, and then I will define my own version that puts emphasis on maintenance/evolution. The aim of this chapter is to demonstrate the following topics: architectural pattern, software reuse, software framework, inversion of control, dependency injection, and test-driven development (TDD).

The architecture of a system is primarily accountable for accomplishing Davis's first two principles of software development (see [1] for all his principles):

1. Make quality number 1.
2. High-quality software is possible.

Without a definite architecture, there is no way of controlling nonfunctional attributes (quality characteristics) of the system. Obviously, every system does have an architecture, but if it isn't clearly articulated, then the "wellness" of the system is going to be some emergent property observable only during execution. Expecting to attain quality by chance isn't a wise and scalable tactic. Consequently, we must produce an architecture upfront to incorporate quality aspects early on and to enforce those attributes throughout the software's life cycle.

There are many definitions of a software architecture, but there is no consensus on the definition in industry. Most definitions put emphasis on specific traits of a system, like the structural buildup of elements and their relationships up to the importance of design decisions. Here are the two that I had found most useful:

Design-level considerations that have system-wide implications.

The software architecture of a system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both.

The first definition comes from Steve McConnell. It encompasses all sorts of “high”-level decisions (like modular decomposition, error processing, logging, etc.) that are important to properly deliver and reason about the system. The second definition is from [3], which focuses on structural (static) and behavioral (dynamic) aspects of a system. It also puts forward the notion that architecture is an abstraction that helps stakeholders in reasoning about the system without building it. However, none of these definitions explicitly state how an architecture enables evolution. Therefore, I have created my own definition shown below, which focuses on nonfunctional requirements and contracts as well as their influence onto development and maintenance/evolution:

The software architecture imposes nonfunctional requirements onto a system, specifies contracts between constituent elements, and preserves their integrity as well as consistency during evolution.

The above definition puts forward the idea that an architecture should make development and evolution easy in a desired direction while putting obstacles to actions departing from the envisioned path. Figure 4.1 illustrates this phenomenon. If an architecture is proper, then evolution will seem smooth and natural. There are

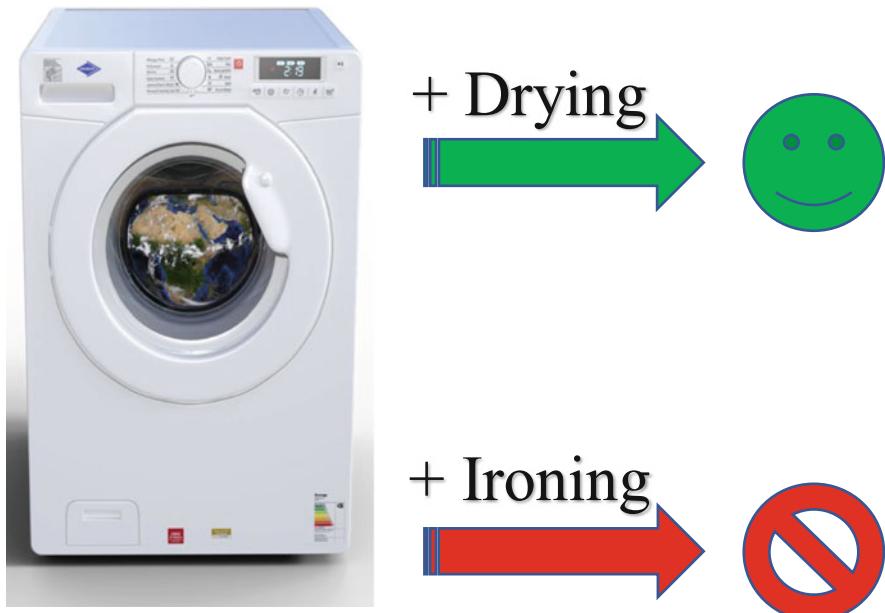


Fig. 4.1 A washing machine has a clear architecture. Some extra features can nicely fit in (washing + drying), while weird extensions are hard or impossible to attain (washing + ironing). Without such a bold architecture, people would feel that anything can be added to it. Communicating the main intention of a product is one of the vital duties of an architecture. We must be able to reason about the system by looking only at its architecture, i.e., it should be obvious, without even trying to build the system, whether it will work or not

many ways to craft an architecture. I recommend the architecture tradeoff analysis method (ATAM) 3.0, as it is pragmatic and includes early technology considerations (see [2] for a detailed treatment of this method). In the next case study, we will showcase the above definition in practice.

4.1 Extension of the System: Problem

The architecture of the initial door lock system is based on the layered approach with well-defined boundaries between layers. The layered design is an example of an architecture pattern that is like a design pattern but with a broader scope. Maintainability is one of the key quality attributes embodied by the layered pattern. By referencing a pattern by name, we succinctly express many details that are contained in the pattern, thus avoiding redundancy and enhancing communication. Figure 3.3 shows the intended layering, where the layers above only talk to the immediate layer below. The idea is to control dependencies and allow independent evolution of the layers in a backward-compatible fashion. We envision that safety rules will change rapidly in the future, so we want to make the middle layer evolvable. We may summarize our intention with the layered architecture in the following way:

- Each layer must be comprised of abstractions relevant for that layer.
- A layer above may only communicate with the immediate layer below.
- Each layer exposes a well-specified contract that serves as an entry point.

Figure 4.2 shows the module decomposition view of the initial system (packages denote layers) that doesn't quite adhere to the previous vision. Each package only exposes a single class to the outside world. In this way, each layer may communicate with the other one through a well-specified entry point (contract). The device layer contains all sorts of devices that are jointly managed by the `DeviceManager` object. It acts as a *Facade* for clients wanting to access devices. The `DoorController` encompasses the main business logic to decide when it is safe to unlock the door. The `DoorLockSystem` is the main class in the system. It just sets the stage for execution and publishes the `allowEntrance` method that accepts an entry code. This method uses the `DoorController` object as a helper to perform the locking/unlocking of the door, depending on the actual conditions.

The initial door lock system, which is presented next, is carefully implemented using TDD. TDD is a design paradigm that forces software engineers to think about testability. Such an approach entails rapid cycles between thinking about tests and development (or even requirements specification). The ordering has turned out to be insignificant, i.e., whether the cycle starts or ends with a test [4]. Most books and webinars lament about TDD as “*write the test first, make it red, then write the code to make it green.*” Many TDD “purists” insist and believe that the test should be written before the target code is created (so that your code doesn't even compile at this point). This is that classical folklore without any scientific underpinning. From

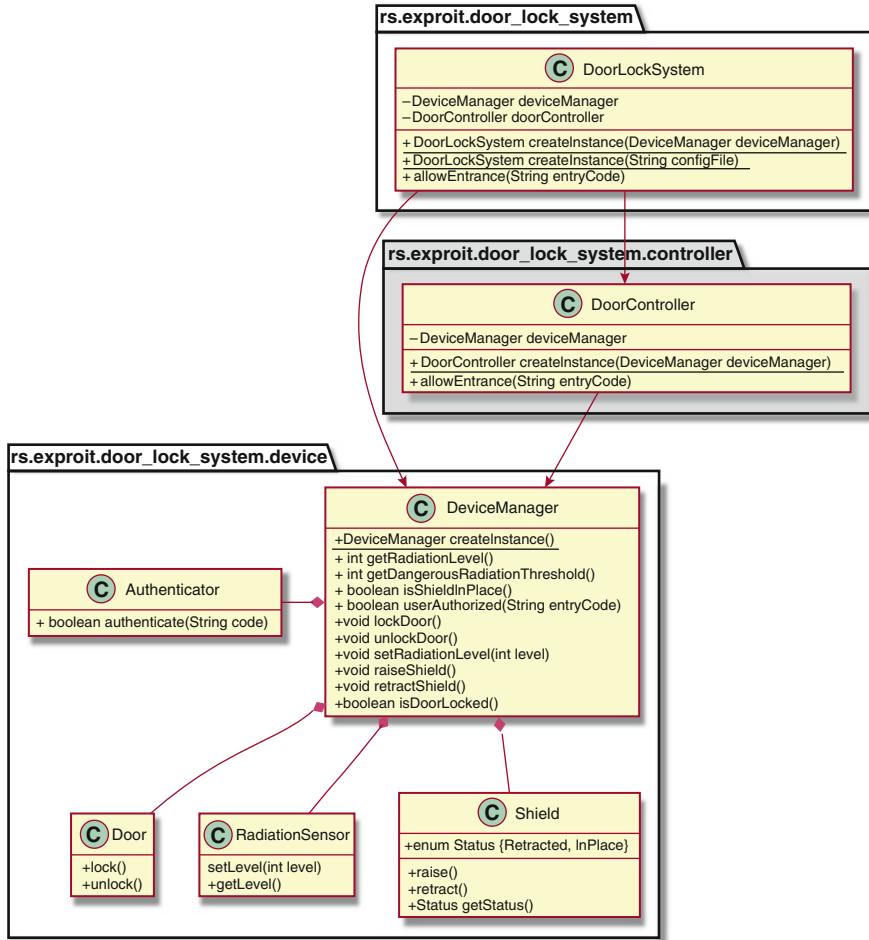


Fig. 4.2 The package structure of the door lock system using the UML class diagram. The `DoorLockSystem` instance doesn't directly use the `DeviceManager` object but rather passes it down to the `DoorController` instance. Unfortunately, this breaks the ideal of our layered pattern, as the upper layer also talks to the lowest layer (skipping the middle one)

the science's viewpoint, the most important thing to remember is to never allow your code (or anything else for that matter) to grow without tests. TDD shrinks the gap between code production and testing. Furthermore, having a comprehensive battery of automated unit tests is compulsory during maintenance and evolution. We are then free to experiment with the code base, rest assured that any regression will be observed by our tests. Of course, this doesn't mean that we should all of a sudden become irresponsible just because we are more protected from entering blind spots while adjusting the code.

As it is so important to understand how the architecture actively shapes evolution, I will present lots of source code for you to see things in action. We will start with one concrete device class, `Door` (as a sample), and follow with the `DeviceManager` class from the device subpackage. The associated unit tests for these classes are also shown (see the accompanying source code for the other classes from this device package):

Listing of the Door Class

```
package rs.exploit.door_lock_system.device;

import static java.lang.System.out;

final class Door {
    private boolean locked;

    public Door(boolean locked) {
        this.locked = locked;
    }

    public void lock() {
        locked = true;
        out.println("Door is locked...");
    }

    public void unlock() {
        locked = false;
        out.println("Door is unlocked...");
    }

    public boolean isLocked() {
        return locked;
    }
}
```

Listing of the Door Unit Test Class

```
package rs.exploit.door_lock_system.device;

import static org.junit.Assert.*;
import org.junit.Test;

import rs.exploit.door_lock_system.device.Door;
```

```

public final class DoorUnitTest {
    @Test
    public final void constructNewInstanceAndCheckStatus() {
        assertFalse(new Door(false).isLocked());
        assertTrue(new Door(true).isLocked());
    }

    @Test
    public void lockDoorAndCheckStatus() {
        Door door = new Door(false);
        door.lock();
        assertTrue(door.isLocked());
    }

    @Test
    public void unlockDoorAndCheckStatus() {
        Door door = new Door(true);
        door.unlock();
        assertFalse(door.isLocked());
    }
}

```

The `Door` class is package visible, and this is the first suggestion that we get from our architecture. Our layered methodology clearly states that an entry point should be well defined with a contract, and inner details hidden from the layers above. In this case, the concrete device isn't something that the layer above is interested in.

The unit test above is named as `DoorUnitTest` instead of simply `DoorTest`. I suggest that you name your tests according to their scope. Here is a simple naming convention to use: <class name><test scope>Test. For example, the scope can be Unit, Integration, System, and Acceptance. This will allow you to selectively run your tests with Maven by specifying the `-Dtest` property on the command line (e.g., `-Dtest=*UnitTest` to exclusively select or `-Dtest=\!*UnitTest` to exclude unit tests):

Listing of the DeviceManager Class (Imports Are Omitted)

```

/**
 * Provides a Facade for working with various devices.
 * It shields the upper layers from device specific details.
 */
public final class DeviceManager {
    private static final String DANGER_LEVEL_DEFAULT = "1000";
    private final String DANGER_LEVEL_KEY = "DANGER_LEVEL";
    private final String SECURITY_CODE_KEY = "SECURITY_CODE";
}

```

```
@SuppressWarnings("serial")
public static final class ConfigurationException
    extends RuntimeException {
    public ConfigurationException(
        String message, Throwable cause) {
        super(message, cause);
    }
}

private final RadiationSensor radiationSensor =
    new RadiationSensor();
private final Shield shield =
    new Shield(Shield.Status.Retracted);
private final Door door = new Door(false);
private final Authenticator authenticator;
private final int dangerousRadiationThreshold;

/**
 * Reads the device configuration and initializes
 * the device layer.
 *
 * @param configFile the configuration file name
 * (for example, device-configuration.properties).
 * @throws IOException if any error occurred while
 * reading the configuration file.
 */
private DeviceManager(String configFile)
    throws IOException {
    assert configFile != null && !configFile.isEmpty();

    Properties configuration = new Properties();
    configuration.load(
        DeviceManager
            .class
            .getClassLoader()
            .getResourceAsStream(configFile));
    dangerousRadiationThreshold =
        Integer.parseInt(
            configuration
                .getProperty(
                    DANGER_LEVEL_KEY,
                    DANGER_LEVEL_DEFAULT));
    authenticator = new Authenticator(
        configuration
```

```
        .getProperty(SECURITY_CODE_KEY)) ;  
    }  
  
    /**  
     * Factory method to get an instance of this class.  
     *  
     * @param configFile the configuration file name  
     * (for example, device-configuration.properties).  
     * @return an instance of this class (may return  
     * same instances on each call).  
     * @throws ConfigurationException in case the  
     * configuration cannot be properly processed.  
     */  
    public static DeviceManager createInstance(String configFile) {  
        try {  
            if (configFile == null || configFile.isEmpty()) {  
                throw new IllegalArgumentException(  
                    "The configuration file name is missing.");  
            }  
  
            return new DeviceManager(configFile);  
        } catch (Exception e) {  
            throw new ConfigurationException(  
                "Unable to read the configuration file", e);  
        }  
    }  
  
    public int getRadiationLevel() {  
        return radiationSensor.getLevel();  
    }  
  
    public int getDangerousRadiationThreshold() {  
        return dangerousRadiationThreshold;  
    }  
  
    public boolean isShieldInPlace() {  
        return shield.getStatus() == Shield.Status.InPlace;  
    }  
  
    public boolean userAuthorized(String entryCode) {  
        return authenticator.authenticate(entryCode);  
    }
```

```
public void lockDoor() {
    door.lock();
}

public void unlockDoor() {
    door.unlock();
}

public void setRadiationLevel(int level) {
    radiationSensor.setLevel(level);
}

public void raiseShield() {
    shield.raise();
}

public void retractShield() {
    shield.retract();
}

public boolean isDoorLocked() {
    return door.isLocked();
}
}
```

Listing of the DeviceManager Unit Test Class (Imports Are Omitted)

```
package rs.exploit.door_lock_system.device;

public final class DeviceManagerUnitTest {
    private final static String CONFIG_FILE_OK =
        "device-configuration.properties";
    private final static String CONFIG_FILE_INVALID =
        "device-configuration-wrong.properties";

    @Test
    public void createNewDeviceManagerAndCheckInitialState() {
        DeviceManager deviceManager =
            DeviceManager.createInstance(CONFIG_FILE_OK);
        assertNotNull(deviceManager);
        assertEquals(
            10,
            deviceManager.getDangerousRadiationThreshold());
        assertEquals(0, deviceManager.getRadiationLevel());
        assertFalse(deviceManager.isShieldInPlace());
    }
}
```

```
        assertTrue(deviceManager.userAuthorized("test"));
        assertFalse(deviceManager.isDoorLocked());
    }

@Test
public void changeShieldStatus() {
    DeviceManager deviceManager =
        DeviceManager.createInstance(CONFIG_FILE_OK);
    deviceManager.raiseShield();
    assertTrue(deviceManager.isShieldInPlace());
    deviceManager.retractShield();
    assertFalse(deviceManager.isShieldInPlace());
}

@Test
public void changeRadiationLevel() {
    DeviceManager deviceManager =
        DeviceManager.createInstance(CONFIG_FILE_OK);
    deviceManager.setRadiationLevel(57);
    assertEquals(57, deviceManager.getRadiationLevel());
}

@Test
public void lockAndUnlockDoor() {
    DeviceManager deviceManager =
        DeviceManager.createInstance(CONFIG_FILE_OK);
    deviceManager.lockDoor();
    assertTrue(deviceManager.isDoorLocked());
    deviceManager.unlockDoor();
    assertFalse(deviceManager.isDoorLocked());
}

@Test(expected = IllegalArgumentException.class)
public void
createNewDeviceManagerWithEmptyConfigurationFileName()
    throws Throwable {
    try {
        DeviceManager.createInstance("");
    } catch (ConfigurationException e) {
        throw e.getCause();
    }
}

@Test(expected = IllegalArgumentException.class)
public void
```

```
createNewDeviceManagerWithNullConfigurationFileName()
    throws Throwable {
try {
    DeviceManager.createInstance(null);
} catch (ConfigurationException e) {
    throw e.getCause();
}
}

@Test(expected = NumberFormatException.class)
public void
createNewDeviceManagerWithInvalidConfigurationFile()
    throws Throwable {
try {
    DeviceManager.createInstance(CONFIG_FILE_INVALID);
} catch (ConfigurationException e) {
    throw e.getCause();
}
}
}
```

The constructor is private since object creation is handled by the appropriate *Factory Method*. This allows the producer to choose the best creation strategy without putting any burden on a client, i.e., it delays the design decision regarding instance creation. Always try to resist the temptation to fix things too early. The input parameter is checked with an assertion since we are inside a private code. Any inappropriate input here would indicate a bug. Notice how the configuration is loaded from the classpath inside the private constructor of the `DeviceManager` class. Finally, you should strive to make your code robust, like here, by providing a sensible default value for the dangerous radiation level. It is always better to run with a decent default value (when possible) than to give up and exit with an error message.

Inside the public *Factory Method* `createInstance`, we must explicitly check and throw an exception if the argument isn't correct. In other words, the precondition here cannot be assumed. Proactively defending against bad input or contract violation is part of the paradigm called *Defensive Programming*. You may also notice that any low-level exception is converted into an instance of the `ConfigurationException` class. This protects upper layers from catching unknown (raw) exceptions. This step is suggested by the layered architecture itself since it dictates that each layer should contain elements at the same abstraction level (device-level problems are not in scope of the controller layer). Of course, information isn't lost by this wrapping since calling the `getCause` method will reveal the source exception. However, this extra insight would be valuable for only those parts of the system that also know how to handle various error conditions.

The matching unit test class contains both positive (with valid data) and negative (with invalid data) tests. This is very important to assure that the code properly realizes the defensive programming concept. The test data is segregated from the production data by following the well-known Maven standard.¹ A unit test basically belongs to the white-box testing category. This test has an intimate knowledge of how the target method is implemented. This is exemplified here too. We explicitly check the source exception, for example, by ensuring that it is an instance of the `NumberFormatException` class in case of a gibberish numeric input.

Shielding your production code with a full battery of automated tests is crucial for later maintenance and evolution. Well-written tests catch regression and serve as an excellent documentation about the system. Writing tests before doing changes on a legacy software is a very efficient strategy. Usually, legacy code isn't properly prepared for maintenance/evolution. The biggest fear is to break something in production. One good tactic is to try to cover with tests the section of code that you intend to alter. You may read more about this in [5], as that book will equip you with practical advice on how to achieve this feat. Nevertheless, I must warn you that most of the techniques presented in [5] are applicable only in situations where you don't care about preserving backward compatibility of APIs.

Below are the listings of the `DoorController` and main `DoorLockSystem` classes, respectively (you may find the matching unit tests in the source code of this book). Please, note that the safety rules are implemented exactly as described in Chap. 3. The bolded `allowEntrance` method is the entry point into the controller layer:

Listing of the `DoorController` Class

```
package rs.exploit.door_lock_system.controller;

import rs.exploit.door_lock_system.device.DeviceManager;

public final class DoorController {
    private final DeviceManager deviceManager;

    private DoorController(DeviceManager deviceManager) {
        this.deviceManager = deviceManager;
    }

    /**
     * Factory method to get an instance of this class.
     *
     * @param deviceManager the instance of the device
     * manager class.
    }
```

¹The test data is located under the `src/test/resources`, while the production data under the `src/main/resources` folder.

```
* @return an instance of this class (may return same
* instances on each call).
*/
public static DoorController createInstance(
    DeviceManager deviceManager) {
    return new DoorController(deviceManager);
}

public boolean allowEntrance(String entryCode) {
    if (isSafeToOpenDoor(entryCode)) {
        deviceManager.unlockDoor();
    } else {
        deviceManager.lockDoor();
    }
    return !deviceManager.isDoorLocked();
}

private boolean isSafeToOpenDoor(String entryCode) {
    return userAuthorized(entryCode) &&
           roomConditionsSafe();
}

private boolean userAuthorized(String entryCode) {
    return deviceManager.userAuthorized(entryCode);
}

private boolean roomConditionsSafe() {
    return shieldInPlace() || radiationLevelLow();
}

private boolean shieldInPlace() {
    return deviceManager.isShieldInPlace();
}

private boolean radiationLevelLow() {
    return deviceManager.getRadiationLevel() <
           deviceManager.getDangerousRadiationThreshold();
}
```

Listing of the DoorLockSystem Class

```
package rs.exploit.door_lock_system;

import rs.exploit.door_lock_system.controller.DoorController;
import rs.exploit.door_lock_system.device.DeviceManager;
```

```
public final class DoorLockSystem {  
    @SuppressWarnings("serial")  
    public static final class ApplicationException extends  
        RuntimeException {  
        public ApplicationException(  
            String message, Throwable cause) {  
            super(message, cause);  
        }  
    }  
  
    private final DeviceManager deviceManager;  
    private final DoorController doorController;  
  
    private DoorLockSystem(String configFile) {  
        try {  
            deviceManager =  
                DeviceManager.createInstance(configFile);  
            doorController =  
                DoorController.createInstance(deviceManager);  
        } catch (Exception e) {  
            throw new ApplicationException(  
                "Cannot start the system.", e);  
        }  
    }  
  
    private DoorLockSystem(DeviceManager deviceManager) {  
        this.deviceManager = deviceManager;  
        doorController =  
            DoorController.createInstance(deviceManager);  
    }  
  
    /**  
     * Factory method to get an instance of this class.  
     *  
     * @param deviceManager the instance of the device  
     * manager class.  
     * @return an instance of this class (may return same  
     * instances on each call).  
     * @throws ApplicationException if anything goes wrong  
     * during startup.  
     */  
    public static DoorLockSystem createInstance(  
        DeviceManager deviceManager) {  
        return new DoorLockSystem(deviceManager);  
    }  
}
```

```
/*
 * Factory method to get an instance of this class.
 *
 * @param configFile the configuration file name (for
 * example, device-configuration.properties).
 * @return an instance of this class (may return same
 * instances on each call).
 * @throws ApplicationException if anything goes wrong
 * during startup.
 */
public static DoorLockSystem createInstance(String configFile) {
    return new DoorLockSystem(configFile);
}

/**
 * Main entry method of the system. This method
 * locks/unlocks the door, and returns back
 * a status information about the outcome.
 *
 * @param entryCode the security code entered by the
 * personnel.
 * @return true if the allowance is permitted taking into
 * account all conditions, false otherwise.
 */
public boolean allowEntrance(String entryCode) {
    return doorController.allowEntrance(entryCode);
}
}
```

Notice how the code base is consistent regarding instance creation. Both classes implement the similar factory method as the `DeviceManager` class. Such consistency is very important to make the code easier to maintain. For maintainable code *read-time convenience* is more important than *write-time convenience*. In other words, devote time and energy into writing down the code in a proper form, following some established convention² since it will be read many more times later. By focusing on logical consistency and integrity of the system, you essentially combat the accidental complexity. In our case, by only looking at one class, the maintainer would know how instance creation works in the others too. A similar wisdom goes with symmetrical APIs, something we will revisit in Chap. 6.

²This is the reason why it is important to agree on a shared code style, possibly checked and enforced by tools (e.g., CheckStyle at checkstyle.sourceforge.net). However, keep in mind that the code style must be first accepted by all software engineers and just later checked by tools. Don't use a tool in the background to reshape whatever code people happen to check in. In that way, the code in the repo will be alienated from all team members.

The `ApplicationException` in the `DoorLockSystem` wraps any lower-level exception. This is the same tactic as the one used in the device layer (see the Exercises section for a comment about the controller layer).

To see things in action, you need to execute the following command from the project's root folder (I assume that you already have Maven installed on your machine): `mvn test`. Since the tests use the JUnit 4 framework (visit junit.org), the code base requires the following dependency inside the Maven's `pom.xml` file:

```
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
    <scope>test</scope>
</dependency>
```

As conjectured, the next version of the software must replace the rigid rules with a more flexible solution. Basically, we need to augment the system with a rules engine. This is the point where all the previous hard work in making the architecture proper pays off. We know exactly where to incorporate the changes without concern that something will break. The architecture clearly confines the scope of the change, and the automated unit tests ensure that we will not introduce regression. Now, we need to decide whether to develop the solution in-house or look for an already available framework. Frameworks are units of reuse, where besides bare code you also reuse design.³ A good framework also gives you guidance on how to incorporate it into your solution. Nowadays, it is always worthwhile to look around first since most of the time, a custom solution would simply reinvent a wheel. Of course, there are situations when none of the available stuff is appropriate; however, you must be sure about this. There are lots of options to consider, and you may want to leverage software economics as a science of choice (see [6] for a good treatment of this area). Sometimes, people like to embark on development just for fun (seek first an approval from stakeholders, as you intend to play with their money).

4.2 Extension of the System: Resolution

Here are the main requirements pertaining to the extension of our system (these codify the essence of our layered architecture pattern):

- We are only allowed to alter the controller layer.
- None of the other layers should be touched.
- All previous tests without modification should pass.

³Frameworks are useful to support and convey scientific research ideas as well. Instead of just writing about theory and giving cryptic pseudocode, you may deliver a sophisticated framework ready for others to use (see [7] for an example).

There is a sophisticated, lightweight open-source rules engine for Java (visit github.com/j-easy/easy-rules/wiki) that perfectly fits our requirements. Our goal is to solely modify the controller layer without breaking compatibility, i.e., keeping the public application programming interface (API) of the `DoorController` class intact. The intention is to restructure the rules to become independent entities that may be combined in an arbitrary fashion driven by the *Easy Rules* framework.

First, we need to include the dependency toward this framework inside our build environment and bump the version number from 1.0.0 to 1.1.0 (we are using the semantic versioning scheme; see semver.org for more details):

```
<dependency>
    <groupId>org.jeasy</groupId>
    <artifactId>easy-rules-core</artifactId>
    <version>3.0.0</version>
</dependency>
```

Next, we must implement our rules around the `Rule` abstraction. This is the benefit of utilizing a framework. Part of that reused design space is the set of potent abstractions that will make our code more expressive and understandable. Anybody who knows the matching framework will recognize all associated elements in our system too. In Easy Rules, a business rule has the following properties (cited from the Web site):

- **Name:** a unique rule name within a rules namespace
- **Description:** a brief description of the rule
- **Priority:** rule priority regarding to other rules
- **Conditions:** a set of conditions that should be satisfied, given a set of facts to apply the rule
- **Actions:** a set of actions to perform when conditions are satisfied (and that may add/remove/modify facts)

There are two types of rules: basic and composite (application of the *Composite* design pattern). Figure 4.3 shows the main abstractions regarding rules in the Easy Rules framework.

Figure 4.4 shows the object diagram for the rules used by our system. Please, note that we must make two small changes to accommodate the logical AND binding of rules inside a composite. Previously, the safety condition for a room was a logical OR between having a *low radiation* and *shield in place* conditions. Now, we need to transform this into a logical AND between *high radiation* and *shield down* conditions. In other words, we apply the well-known De Morgan's law ($P \vee Q \equiv \neg(\neg P \wedge \neg Q)$). This will be evident from the listing of the `RoomConditionsSafe` class below.

Here are the listings of the restructured `DoorController`, `UnsafeToOpenDoor`, `RoomConditionsSafe`, and `UserAuthorized` classes, respectively (the rest of the classes are available as part of the accompanying

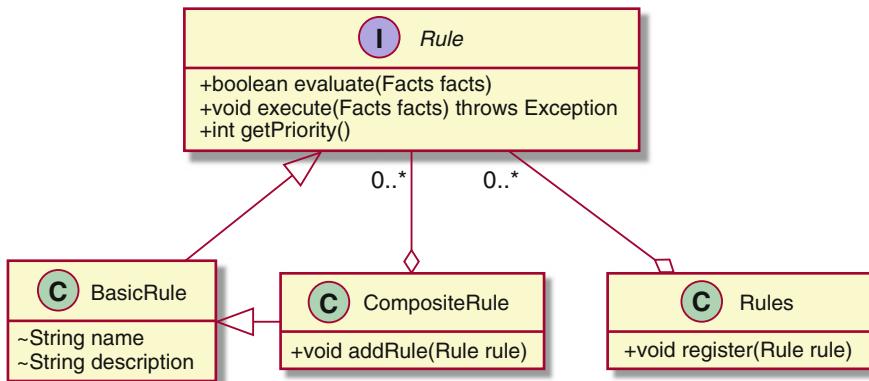


Fig. 4.3 The relationships between abstractions pertaining to rules. The Rules class is simply a container for Rule objects. The Facts class is a map of `<String, Object>` that contains arguments for rules (mostly used inside the corresponding evaluate method). A rule may even produce new facts as part of its execution. The principle is very easy: a rule is evaluated, and if the outcome is true, then that rule's action is run. A composite rule ensures that all contained rules are satisfied before its action is affected. The whole rule processing is orchestrated by the main RulesEngine class

source code). The last one is implemented via annotations, which demonstrates one drawback of this approach with this framework, as we will soon discuss:

Listing of the Restructured DoorController Class (Import Are Omitted)

```

public final class DoorController {
    static final String DEVICE_MANAGER_KEY = "deviceManager";
    static final String ENTRY_CODE_KEY = "entryCode";

    private final DeviceManager deviceManager;
    private final RulesEngine rulesEngine;
    private final Facts facts = new Facts();
    private final Rules rules = new Rules();

    private boolean isRunUnderJUnitTest() {
        for (StackTraceElement element :
            Thread.currentThread().getStackTrace()) {
            if (
                element
                    .getClassName()
                    .startsWith("org.junit.")) {
                return true;
            }
        }
    }
}
  
```

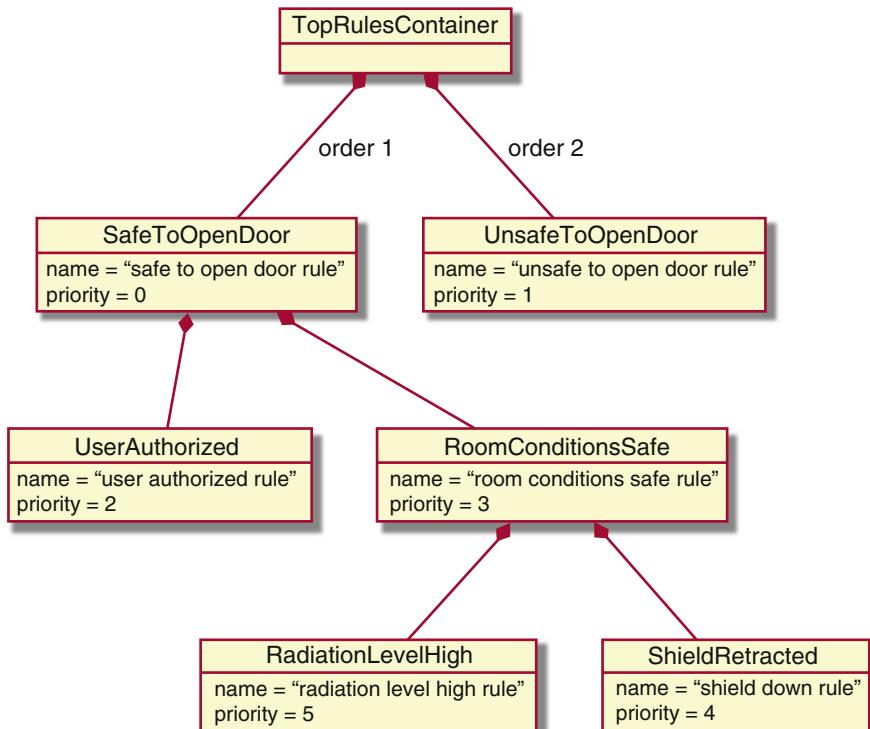


Fig. 4.4 The instantiation of rules inside our restructured door-locking system. Each rule has a designated priority for ordering purposes. From the perspective of the top rules container, it will first process the left branch, and afterward, the right branch (as denoted by the order property). The `UnsafeToOpenDoor` rule is always triggered if the opposite `SafeToOpenDoor` one was skipped, and vice versa. This is dictated by the `skipOnFirstAppliedRule` attribute of the `RulesEngine` instance. We will also set the `skipOnFirstFailedRule` property to avoid unlocking a door in case of an error

```

    return false;
}

private DoorController(DeviceManager deviceManager) {
    this.deviceManager = deviceManager;
    rulesEngine =
        aNewRulesEngine()
            .withSkipOnFirstAppliedRule(true)
            .withSkipOnFirstFailedRule(true)
            .withSilentMode(!isRunUnderJUnitTest())
            .build();
}
  
```

```

        facts.put(DEVICE_MANAGER_KEY, deviceManager);
        rules.register(new SafeToOpenDoor());
        rules.register(new UnsafeToOpenDoor());
    }

    /**
     * Factory method to get an instance of this class.
     *
     * @param deviceManager the instance of the device
     * manager class.
     * @return an instance of this class (may return same
     * instances on each call).
     */
    public static DoorController createInstance(
        DeviceManager deviceManager) {
        return new DoorController(deviceManager);
    }

    public boolean allowEntrance(String entryCode) {
        facts.put(ENTRY_CODE_KEY, entryCode);
        rulesEngine.fire(rules, facts);
        return !deviceManager.isDoorLocked();
    }
}

```

Listing of the Restructured UnsafeToOpenDoor Class

```

package rs.exploit.door_lock_system.controller;

import org.jeasy.rules.api.Facts;
import org.jeasy.rules.core.BasicRule;
import rs.exploit.door_lock_system.device.DeviceManager;

final class UnsafeToOpenDoor extends BasicRule {
    public UnsafeToOpenDoor() {
        super("unsafe to open door rule",
              "checks whether it is unsafe to unlock the door");
    }

    @Override
    public boolean evaluate(Facts facts) {
        return true;
    }
}

```

```

@Override
public void execute(Facts facts) throws Exception {
    DeviceManager deviceManager =
        (DeviceManager) facts
            .get(DoorController.DEVICE_MANAGER_KEY);
    deviceManager.lockDoor();
}

@Override
public int getPriority() {
    return 1;
}
}

```

Listing of the Restructured RoomConditionsSafe Class (Imports Are Omitted)

```

final class RoomConditionsSafe extends CompositeRule {
    RoomConditionsSafe() {
        super("room conditions safe rule",
              "checks whether conditions are safe in the room");
        addRule(new RadiationLevelHigh());
        addRule(new ShieldRetracted());
    }

    @Override
    public boolean evaluate(Facts facts) {
        return !super.evaluate(facts);
    }

    @Override
    public int getPriority() {
        return 3;
    }
}

```

Listing of the Restructured UserAuthorized Class (Imports Are Omitted)

```

@Rule(name = "user authorized rule",
      description = "checks if the user is authorized")
public final class UserAuthorized {
    @Condition
    public boolean userAuthorized(
        @Fact(DoorController.DEVICE_MANAGER_KEY)
        DeviceManager deviceManager,
        @Fact(
            DoorController.ENTRY_CODE_KEY) String entryCode) {

```

```

        return deviceManager.userAuthorized(entryCode);
    }

    @Action
    public void doNothing() throws Exception { }

    @Priority
    public int rulePriority() {
        return 2;
    }
}

```

The `DoorController` sets up the rules system inside its private constructor. Notice the private `isRunUnderJUnitTest` method. It returns `true` if the class is run under the supervision of the JUnit framework. The idea is to turn on automatically the verbose output mode of the `RulesEngine` class. This is a very bad practice! First, it introduces a hidden dependency onto the JUnit framework. Moreover, it bloats the production code with testing-related stuff. Finally, the code will stop working if tested under a different testing framework (see the Exercises section for a hint toward a better approach).

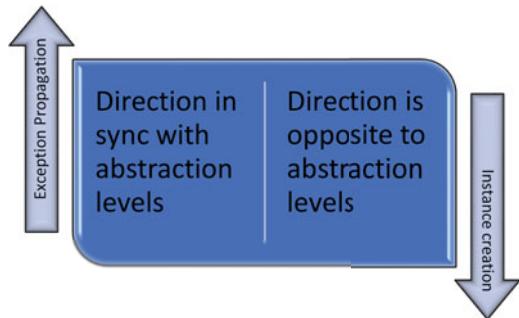
The classes that are extended from `BasicRule` or `CompositeRule` are package private, while the `UserAuthorized` class is forced to be public. This is the consequence of using the `@Rule` annotation. Any public entity would be visible outside of the confining package (layer), which is harmful. Moreover, the class must have a method annotated with `@Action` even if it doesn't do anything (as in our case). Again, we can see our architecture in action! This problem wouldn't even exist without an architecture, as any solution could do (see in Chap. 5 what happens when development and evolution are done in an ad hoc manner). However, the architecture in place gives a clear signal when something violates its structure. In this case, it is this unacceptable leakage of a package private class.

A switch to the Rules Engine framework had inverted the control, i.e., we had managed to apply the *Inversion of Control* principle [8]. Previously, our code had explicitly called and executed the safety rules encoded as methods of the `DoorController` class. Now, we just initialize the Rules Engine framework with our rules and let it call back into our code when appropriate. Such inversion is typical when you rely on frameworks (e.g., the built-in Java Swing framework calls back into your custom GUI components each time a user makes an action on the UI impacting your components).

4.3 Elimination of a Technical Debt: Problem

Version 1.1 of our software still doesn't conform 100% to our architectural principles, and this makes the code a bit harder to maintain. Notice that we do have low-level details in the uppermost layer (the configuration file and the

Fig. 4.5 The aim of the dependency injection activity coupled with inversion of control is to turn the direction of the instance creation mechanism to be in sync with abstraction levels



DeviceManager class). Consequently, the highest layer had a direct dependency onto the lowest one. We would like to only depend on the layer below, which is the controller layer. Knowing details from the device layer (like the way it is configured) impedes our desire to evolve them independently. We had already solved the exception propagation; something similar is required now for the object instantiation mechanism. All this is represented on Fig. 4.5.

We can also look at this from the viewpoint of technical debt. If you have an architecture, then you may display on the cause/effect diagram (a.k.a. fishbone or Ishikawa diagram) the contribution of each module toward the total technical debt. The architecture would then answer the questions which technical debts incur interest payments (you may read about different types of technical debt in [9]) and which may be paid off later, as needed. Figure 4.6 shows this diagram in our case. Obviously, the bad technical debt lurks in the highest layer. The other two are localized, and may be treated independently (see Exercise 7). Our next task is to eliminate the technical debt with an interest payment.

4.4 Elimination of a Technical Debt: Resolution

We will utilize Google's Guice framework (visit <https://github.com/google/guice>), which is a lightweight dependency injection framework for Java 6+ (for an introduction to dependency injection read [10]). Hence, we must augment our build file with the following dependency:

```
<dependency>
    <groupId>com.google.inject</groupId>
    <artifactId>guice</artifactId>
    <version>4.1.0</version>
</dependency>
```

Dependency injection is a paradigm, where object creation is separated from its usage. The idea is to externally inject dependencies. It is possible to implement all this even without any framework just by handing over all dependencies as input

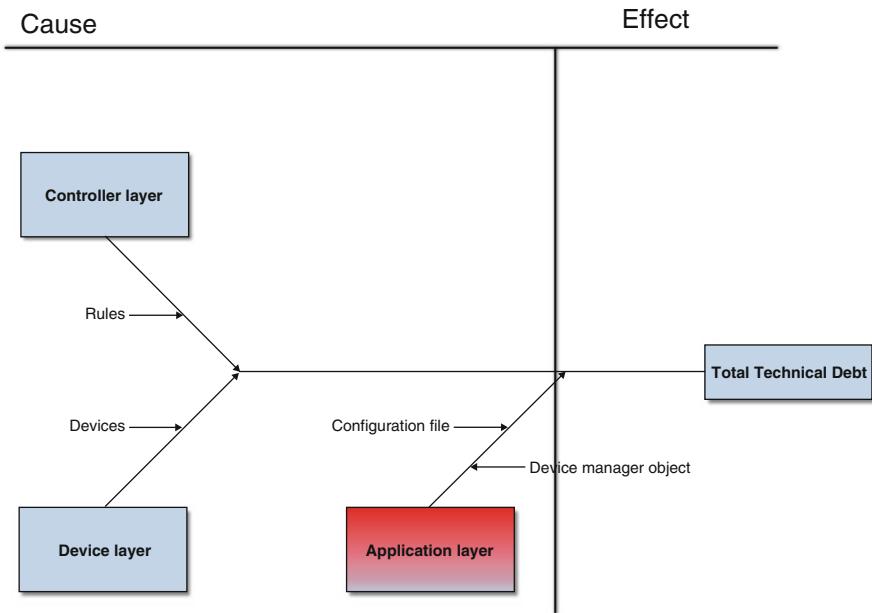


Fig. 4.6 All three layers contribute to the total technical debt regarding instance creation (I recommend the SmartDraw tool for composing fishbone diagrams; visit www.smartdraw.com). However, only the application layer is an issue for now, as that establishes a bad coupling toward the device layer. The others may wait their turn to be paid off. All this evaluation is valid in the context of our layered architecture. You may notice that without an architecture, we could not prioritize the payoff schedule

parameters (either to constructors or methods). Nevertheless, a framework brings in structure and consistency. Furthermore, it enables us to strengthen our code base to proactively disallow any departures from the architecture, as this section will demonstrate.

Guice isolates an object graph setup in separate entities called *modules*. This is a fine example of the *Separation of Concerns* principle. Each module describes an object dependency graph. Modules may be hierarchically organized as shown in Fig. 4.7. This organization completely isolates parent modules from internal details of their child modules. Classes on the other hand, don't need to contain code for creating subordinate objects. These will be magically provided by the framework. This considerably decreases the number of factories in the code base. In our case, they vanished.

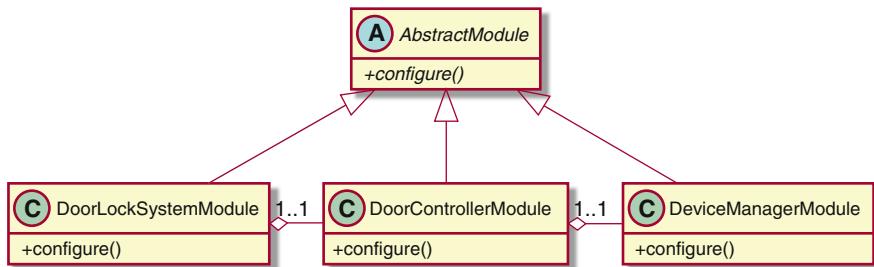


Fig. 4.7 The relationships between modules. All of them override the configure method to define dependencies

Here is the listing of the restructured **DoorLockSystem** class:

Listing of the New **DoorLockSystem** Class (The Inner **ApplicationException** Class Is Omitted)

```

package rs.exploit.door_lock_system;

import com.google.inject.Guice;
import com.google.inject.Inject;
import com.google.inject.Injector;
import com.google.inject.Singleton;

import rs.exploit.door_lock_system.conf.DoorLockSystemModule;
import rs.exploit.door_lock_system.controller.DoorController;

@Singleton
public final class DoorLockSystem {
    private static Injector injector;
    private final DoorController doorController;

    @Inject
    DoorLockSystem(DoorController doorController) {
        this.doorController = doorController;
    }

    /**
     * Gets an instance of this class.
     *
     * @return a sole instance of this class.
     */
    public static synchronized DoorLockSystem getInstance() {
        if (injector == null) {
    
```

```

        injector = Guice.createInjector(
            new DoorLockSystemModule());
    }
    return injector.getInstance(DoorLockSystem.class);
}

/**
 * Main entry method of the system. This method
 * locks/unlocks the door, and returns back
 * a status information about the outcome.
 *
 * @param entryCode the security code entered by the
 * personnel.
 * @return true if the allowance is permitted taking into
 * account all conditions, false otherwise.
 */
public boolean allowEntrance(String entryCode) {
    return doorController.allowEntrance(entryCode);
}
}

```

The client API had changed a little bit⁴; there is a single instance retrieval method `getInstance` instead of factories. The `DoorLockSystem` is a *Singleton*, which is the default scope in most dependency injection frameworks. However, this fact is explicitly stated here with the `@Singleton` annotation. The `DoorLockSystem` only depends on the Guice framework, and the `DoorController` object is injected automatically as a side effect of calling the `getInstance` on the `Injector` object. Again, the reason that the `DoorLockSystem` also follows this convention is to preserve consistency.

The `DoorController` class is now extremely simple, with only a single package private constructor annotated with `@Inject` (see the accompanying source code). The `DeviceManager` class had only kept its package private constructor, which looks like as follows:

⁴Even though clients may trivially adapt to this change, it is still a nonbackward-compatible change. Consequently, the version number must be raised to 2.0.0. If you control your clients, then doing these kinds of restructurings isn't a big deal. However, with a huge unknown consumer base, you are stuck with your old API forever. The only thing you may try is to mark methods as deprecated with the `@Deprecated` annotation. Creating maintainable APIs is a special topic, and you may consult my book for an extensive treatment [11].

```
@Inject
DeviceManager(
    @Named(CONFIGURATION_PARAMETER_NAME) String configFile) {
try {
    Properties configuration = new Properties();
    configuration.load(
        DeviceManager
            .class
            .getClassLoader()
            .getResourceAsStream(configFile));
    dangerousRadiationThreshold = Integer.parseInt(
        configuration.getProperty(
            DANGER_LEVEL_KEY,
            DANGER_LEVEL_DEFAULT));
    authenticator = new Authenticator(
        configuration.getProperty(SECURITY_CODE_KEY));
} catch (Exception e) {
    throw new ConfigurationException(
        "Unable to read the configuration file", e);
}
}
```

The most important change is that there is no possibility anymore to unintentionally create a dependency between layers regarding instance creation. All the factories are gone, and constructors are package private. I advise you to avoid putting the `@Inject` annotation onto private fields, as this practice may hinder unit testing. Exercise 2 contains a hint as to why previous versions of the `DoorLockSystemUnitTest` class weren't quite proper. Here is the restructured version that uses the Mockito framework (see the `pom.xml` file in the source code for the dependency definition):

Listing of the `DoorLockSystemUnitTest` Class (The Imports Are Omitted)

```
public class DoorLockSystemUnitTest {
    private final static String ENTRY_CODE = "test";

    private final DoorController doorController =
        mock(DoorController.class);
    private final DoorLockSystem doorLockSystem =
        new DoorLockSystem(doorController);

    @Before
    public void setupDummyController() {
        when(
            doorController
```

```
        .allowEntrance(eq(ENTRY_CODE)))
        .thenReturn(true);
when(
    doorController
    .allowEntrance(
        AdditionalMatchers.not(eq(ENTRY_CODE))))
    .thenReturn(false);
}

@Test
public final void allowEntranceWithCorrectEntryCode() {
    assertTrue(doorLockSystem.allowEntrance(ENTRY_CODE));
    verify(doorController).allowEntrance(eq(ENTRY_CODE));
}

@Test
public final void disallowEntranceWithIncorrectEntryCode() {
    assertFalse(doorLockSystem.allowEntrance(
        ENTRY_CODE + "garbage"));
    verify(doorController)
        .allowEntrance(eq(ENTRY_CODE + "garbage"));
}
```

In the above test class, we mock the real `DoorController` class and setup expectations regarding its usage. Inside each test case, we also verify that the mock was indeed called with proper input parameters. Next are the listings of the Guice modules:

Listing of the DoorLockSystemModule Class (Import Are Omitted)

```
public final class DoorLockSystemModule extends AbstractModule {  
    @Override  
    protected void configure() {  
        install(new DoorControllerModule());  
    }  
}
```

Listing of the DoorControllerModule Class (Import Are Omitted)

```
public final class DoorControllerModule extends AbstractModule {  
    @Override  
    protected void configure() {  
        install(new DeviceManagerModule());  
    }  
}
```

Listing of the DeviceManagerModule Class (Import Are Omitted)

```
public final class DeviceManagerModule extends AbstractModule {  
    private static final String CONFIG_FILE =  
        "device-configuration.properties";  
  
    @Override  
    protected void configure() {  
        bindConstant()  
            .annotatedWith(  
                Names.named(  
                    DeviceManager.CONFIGURATION_PARAMETER_NAME))  
            .to(CONFIG_FILE);  
    }  
}
```

Apparently, the parent module knows nothing about what is happening inside the child module. It only knows that it must install it, and the child will take care of the rest. Such clear boundaries between modules as well as the fact that they are isolated in a separate package makes the kernel (classes that perform the core application logic) lean and maintainable.

4.5 Summary

Let's revisit the constituent parts of our definition of an architecture and explain how this chapter had illuminated them:

- **Imposes nonfunctional requirements onto a system.** The architecture actively encodes the most salient quality attributes (part of the so-called architecturally significant requirements) of the software. Functions are later added onto this foundation. Without an exact architecture, functionalities could be delivered in any form (most of them would be regarded as unacceptable by clients, as everybody has some expectation of how *well* the system should perform its job). The architecture cannot emerge by chance; it must be carefully created by an architect (or a team of architects with the chief architect serving as an arbitrator). In our case study, maintainability and evolvability were the crucial quality attributes. These had influenced our decision to use the layered architecture pattern.
- **Specifies contracts between constituent elements.** Major building blocks of a system inevitably need to communicate and exchange data. Consequently, each of them must specify a well-defined API; otherwise, we cannot have control over dependencies between elements and changes to the system. In our example, the essential APIs were entry points into layers.

- **Preserves the integrity as well as consistency of the previous two items during evolution.** At every moment in time, the architecture gives clear cues what is currently good and wrong as well as what is possible in the future. As new features are added to the system, its complexity increases and its structure deteriorates. Some features cannot even be added without completely ruining the existing solution. Without an architecture, you can only guess what will happen and be able to find that out in an expensive way (after trying to perform the change). In our case study, we had observed that the architecture was the key gatekeeper against disadvantageous alterations.

In version 1.1, we had successfully reused an open-source Rules Engine framework. This was possible due to the well-isolated section of the code base where changes have had happened and the relatively lightweight nature of the Easy Rules framework. In general, reuse isn't that smooth in practice. There are many frictions between the assumptions that reusable components have about the environment and our beliefs toward their behavior and quality characteristics (for a nice summary see [12]). At any rate, if we do possess a proper architecture, then at least we have a solid basis against which to evaluate potentially reusable software artifacts. An architecture is even a better filter of candidate products than the requirements themselves. Moreover, open-source software is frequently used as closed source [13]. This was the case here with the Easy Rules, Guice, and Mockito frameworks.

In software, higher quality is cheaper, i.e., investing in quality as a planned activity entails less expensive rework later. Quality is inherently a vague term, so it must be formulated in an explicit way. Team members should establish a common quality standard and reflect that in the product. The architecture is the best medium to convey this shared quality expectation. Explaining quality is quite tedious, but documenting the architecture is rather systematized (see [3] for examples). Therefore, the architecture helps conveying the accepted facets of quality inside the product to stakeholders [14].

The key takeaways from this chapter are listed below:

- The architecture plays the central role in building a long-lasting evolvable software system.
 - You must invest enough upfront design time to produce the solid initial architecture of the system.
 - Without an architecture, the development and evolution of a system would be a continuous random drifting instead of an engineered undertaking.
-

4.6 Exercises

1. The `DeviceManager` class is missing an important negative test. Can you spot which one? Add it to the test suite.
2. The `DoorController` class doesn't have separate unit tests, as it was tested indirectly through a higher layer. Does this practice trigger some warning signs?

Can we even talk about a unit test if the test is summoned from another part of the system? Implement the corresponding `DoorControllerUnitTests` class.

3. The controller layer doesn't wrap exceptions but rather silently passes them up. This looks strange. Why? When you do have a strong foundation in the form of an architecture and conventions, any discrepancy is painfully sticking out. When everything floats in the air, nobody thinks about detours, as there are no checkpoints against which to compare a chosen route. Fix the problem by introducing a new exception class that will wrap lower-level ones.
4. It is a bad practice to put security codes in plain text inside configuration files (especially in production). Devise a strategy to solve this issue.
5. Fix the problem with the leaking internal class `UserAuthorized` by avoiding the usage of Easy Rules annotations.
6. Resolve the problem of running the rules engine in a test regime without introducing extra code into production. Also, eschew any hidden dependencies onto the concrete testing framework. Look at *Spring Profiles*, part of the Spring framework, as an inspiration (visit docs.spring.io/spring-boot/docs/current/reference/html/boot-features-profiles.html) if you haven't already used it before. You should also make test runs more verbose by attaching a custom `RuleListener` to the `RulesEngine` object by invoking the `.withRuleListener` builder method. This tactic can prove useful even in production, when more logs would help resolve problems (we will talk more about logging in Chap. 9).
7. Perform the similar dependency injection restructuring inside the controller and device layers. Notice that these activities will not impact other layers.
8. Using string literals inside the `@Named` annotation isn't quite a scalable approach, i.e., it could cause maintenance headaches in a large code base. An alternative is to use custom annotations and mark them with the `@BindingAnnotation` meta-annotation (you may consult [github.com/google/guice/wiki/Binding Annotations](https://github.com/google/guice/wiki/BindingAnnotations)). Refactor the `DeviceManagerModule` class to utilize such a custom annotation. The annotation itself may look like as follows:

```
@BindingAnnotation  
@Target({ PARAMETER, METHOD })  
@Retention(RUNTIME)  
public @interface ConfigurationFile {}
```

Inside the `DeviceManagerModule` class you would use the `.annotatedWith` method passing in the `ConfigurationFile.class` as a parameter. Inside the `DeviceManager` class, you would annotate the input `configFile` parameter with the above annotation.

References

Further Reading

1. Davis MA (1995) 201 principles of software development. McGraw-Hill, New York, NY. This book gives you a lucid list of principles, that you should obey during development to craft an evolvable software system
2. Kazman R, Cervantes H (2016) Designing software architectures: a practical approach. Addison-Wesley Professional, Boston, MA. This book is expanding on [3], and contains the latest description of the ATAM method (version 3.0). It is the most practical ATAM variant, as it includes technology considerations early on. The book also comes with 3 extensive case studies showcasing ATAM 3.0 in action
3. Kazman R, Clements P, Bass L (2012) Software architecture in practice, 3rd edn. Addison-Wesley Professional, Boston, MA. This is one of the most prominent book about software architecture. It introduces a powerful concept of an Architecture Influence Cycle, which presents the architecture from multiple viewpoints (contexts). Moreover, it equips the reader will practical advises how various architectural tactics may help in attaining the desired quality attributes of a system

Regular Bibliographic References

4. Fucci D, Erdogmus H, Turhan B, Oivo M, Juristo N (2017) A dissection of the test-driven development process: does it really matter to test first or to test-last? *IEEE Trans Softw Eng* 43 (7):597–614
5. Feathers M (2004) Working effectively with legacy code. Prentice Hall, Upper Saddle River, NJ
6. Tockey S (2005) Return on software: maximizing the return on your software investment. Addison-Wesley Professional, Boston, MA
7. Varga DE, Beretka FS, Noce C, Sapienza G (2015) Robust real-time load profile encoding and classification framework for efficient power systems operation. *IEEE Transa Power Syst* 30 (4):1897–1904
8. Fowler M (2005) Inversion of control. martinfowler.com/bliki/InversionOfControl.html. Accessed 20 Aug 2017
9. McConnell S (2008) Managing technical debt – white paper. Construx Software, Bellevue, WA
10. Evans JB, Verburg M (2012) The well-grounded java developer: vital techniques of Java 7 and polyglot programming. Manning, Shelter Island, NY
11. Varga E (2016) Creating maintainable APIs: a practical, case-study approach. Apress, Berkeley, CA
12. Garlan D, Allen R, Ockerbloom J (2009) Architectural mismatch: why reuse is still so hard. *IEEE Softw* 26. <https://doi.org/10.1109/MS.2009.86>
13. Torchiano M, Morisio M (2004) Overlooked aspects of COTS-based development. *IEEE Softw* 21:88–93
14. O'Reilly Radar (2015) Better code is cheaper. soundcloud.com/oreilly-radar/better-code-is-cheaper. Accessed 24 Aug 2017

Part II

Forward Engineering

Incredibly often, this is the default regime of many companies. Quality is usually invisible (from the viewpoint of customers and management), and many problems are only noticeable by software engineers. Unfortunately, software engineers are bad at expressing quality problems in a form that is understandable and quantifiable in business terms. The business people cannot decipher why and how some technical conundrum is going to impact the schedule or how it will influence the cash flow. On the other hand, software engineers get nervous that nobody sees the upcoming danger. Eventually, quality problems will become externally visible, but too late. With rapid Agile delivery cycles, quality issues are especially upsetting. In this case, software maintenance as a phase comes rather early. Consequently, most future changes occur on a production code. If that code is of a low quality, then it will devour considerable effort, resources, and money to keep the product running. According to [2], around 15% of the source code gets changed between releases. This is a huge amount of code. It isn't hard to see that without proper quality management, the software maintenance costs may easily cripple a company or prevent it from capitalizing on innovation. The latter has a similar consequence, as a more vivid competitor will outrun such a company on the market. This chapter presents a long software development/evolution story to highlight many problems that will surface when quality is not treated seriously. It showcases how ignoring maintainability instigates an exponential growth in expenses.

Of course, quality problems aren't only confined to construction. They are also spread out to requirements, design, testing, documentation, etc. The stories below do incorporate lack of quality as an overall concern. Watch out how requirements are handled in a totally informal way, notice how quality is treated as something nice to have after the more important functionalities are finalized, etc.

NOTE: The code listings are presented in a form that I usually find in production systems. I have intentionally resisted the temptation to beautify them too much. If I would show here what I've really seen in my career, then you would think that I've had made all that up.

5.1 Initial Version: Problem

We start our journey in a fictional company called the ACME Corporation (a popular name for a corporation in various cartoons) that seems capable of delivering any kind of a product. We are using ACME in the same ironic fashion as in Road Runner cartoons since dreaming about a product without considering quality is futile rather than cogent, as the Greek derivation of *ACME* would imply.

ACME has decided to develop from scratch a variant of the Computer-Aided Design (CAD) software, more specifically an Electronic Design Automation (EDA) system, in Java. This is the company's first software product aimed as a commercial off-the-shelf (COTS) asset. The design plans for a separate internal linear algebra module, whose constituent part is the class for representing matrices. For now, there is no vision to make this linear algebra module a shared company property. The task to implement the `Matrix` class was given to a developer (he wasn't a junior but neither a senior). Everybody thought that handling matrices is a no-brainer since it is a so well-defined problem. He received the following specification as a starting point:

- The class should be named as `Matrix` (inside the `com.example.eda.linalg` package) and must have a constructor receiving the dimensions of the matrix.
- The elements of the matrix are double-precision floating-point numbers.
- The class should have operators for addition and multiplication of matrices as well as multiplying the matrix with a scalar value.
- The class must define a proper equality operator.
- The class should have methods to get and set an element of the matrix.
- The instance of the class should be usable as a hash map key.
- The class should print out the content of a matrix on the console.

The management has decreed that the project is already late, and this task must be finished as soon as possible (the famous ASAP acronym). Unit testing and documentation should wait until the class is fully implemented. It is enough to perform smoke testing (a form of testing when only a couple of sample tests are run to ensure the basic functioning of the system).

5.1.1 Buy vs. Build

Before hastily embarking to build a custom solution, companies should always contemplate what is the better choice: build or buy (in the case of open-source, this would translate to use/reuse). This isn't a simple decision and lots of options need to be pondered. The formal procedure behind this is known as *reuse-based software engineering* [5]. This is a complex topic and out of the scope of this book.

I've chosen here the path with building a proprietary matrix-handling package to illuminate the process of development and evolution. Otherwise, you would

probably want to search the Internet first for available packages. One excellent starting point for linear algebra could be the Apache Commons Math library (visit commons.apache.org/proper/commons-math/). Of course, you should also need to perform market research what others offer in the realm of EDA (e.g., look at KiCad EDA at kicad-pcb.org).

5.2 Initial Version: Resolution

The developer has managed to finish the job on time from the viewpoint of the management. The separate QA department also performed their tests in a hurry and announced that the class is behaving according to the specification. The developer was immediately shuffled to another task. However, he was trying to convince the management that the job wasn't done yet. The unit tests are missing and nothing is documented. He has emphasized that these shortcomings would jeopardize further evolution of the software. Nevertheless, the management believed that the confirmation from the QA department is more eminent than his opinion, and if something "works," then all the rest is simply luxury for spare time. Here is the 1.0.0 version of the code:

Listing of the 1.0.0 Version of the Matrix Class with Line Numbers

```
1 package com.example.eda.linalg;
2
3 import java.util.ArrayList;
4
5 /**
6  * Class to work with matrices.
7 */
8 @SuppressWarnings("serial")
9 public class Matrix extends ArrayList<Double> {
10     private int n;
11     private int m;
12
13     public Matrix(int m, int n) {
14         super(m * n);
15         this.m = m;          // number of rows
16         this.n = n;          // number of columns
17
18         // Initialize elements with zero.
19         for (int i = 0; i < m; i++) {
20             for (int j = 0; j < n; j++) {
21                 add(elementIndex(i, j), 0.0);
22             }
23     }
24 }
```

```
23         }
24     }
25
26     private int elementIndex(int i, int j) {
27         return i * n + j;
28     }
29
30     public double getElement(int i, int j) {
31         return get(elementIndex(i, j));
32     }
33
34     // Returns back the previous value.
35     public double setElement(
36         int i, int j, double value) {
37         return set(elementIndex(i, j), value);
38     }
39
40     // Returns back the sum of this matrix and
41     // the other one.
42     public Matrix addMatrix(Matrix other) {
43         Matrix newMatrix = new Matrix(m, n);
44         for (int i = 0; i < n; i++) {
45             for (int j = 0; j < m; j++) {
46                 newMatrix.setElement(
47                     j, i,
48                     getElement(j, i) +
49                     other.getElement(j, i));
50             }
51         }
52         return newMatrix;
53     }
54
55     // Returns back the product of this matrix and
56     // the other one.
57     public Matrix multiplyMatrix(Matrix other) {
58         Matrix newMatrix = new Matrix(m, other.n);
59         for (int i = 0; i < m; i++) {
60             for (int j = 0; j < n; j++) {
61                 for (int k = 0; k < n; k++) {
62                     newMatrix.set(
63                         elementIndex(i, j),
64                         newMatrix.getElement(i, j) +
65                         getElement(i, k) *
66                         other.getElement(k, j)
67                     );
68     }
```

```
68             }
69         }
70     }
71     return newMatrix;
72 }
73
74 // Returns back the product of this matrix and
75 // the scalar value.
76 public Matrix multiplyWithScalar(double value) {
77     Matrix newMatrix = new Matrix(m, n);
78     for (int i = 0; i < m; i++) {
79         for (int j = 0; j < n; j++) {
80             newMatrix.setElement(
81                 i, j, getElement(i, j) * value);
82         }
83     }
84     return newMatrix;
85 }
86
87 public static void main( String[] args ) {
88     Matrix a = new Matrix(3, 3);
89     for (int i = 0; i < 3; i++)
90         for (int j = 0; j < 3; j++) {
91             a.set(
92                 a.elementIndex(i, j),
93                 (double) (i + j));
94         }
95
96     Matrix b = new Matrix(3, 3);
97     for (int i = 0; i < 3; i++)
98         for (int j = 0; j < 3; j++) {
99             b.set(
100                 b.elementIndex(i, j),
101                 (double) (i + j));
102         }
103
104     System.out.println(a.toString());
105     System.out.println(b.toString());
106     System.out.println(
107         a.multiplyWithScalar(10.0).toString());
108     System.out.println(
109         a.multiplyMatrix(b).toString());
110     System.out.println(a.addMatrix(b).toString());
111     System.out.println(a.equals(b));
112     System.out.println(a.hashCode());
```

```
113     a.setElement(2, 2, 111.0);
114     System.out.println(a.getElement(2, 2));
115     System.out.println(a.equals(b));
116     System.out.println(a.hashCode());
117 }
118 }
```

The main method contains the smoke tests, which generate the following output:

```
[0.0, 1.0, 2.0, 1.0, 2.0, 3.0, 2.0, 3.0, 4.0]
[0.0, 1.0, 2.0, 1.0, 2.0, 3.0, 2.0, 3.0, 4.0]
[0.0, 10.0, 20.0, 10.0, 20.0, 30.0, 20.0, 30.0, 40.0]
[5.0, 8.0, 11.0, 8.0, 14.0, 20.0, 11.0, 20.0, 29.0]
[0.0, 2.0, 4.0, 2.0, 4.0, 6.0, 4.0, 6.0, 8.0]
true
1080652063
111.0
false
1085616415
```

The above output seemingly reassures that everything is ok with the Matrix class. Version 1.0.0 has entered production together with the packaged EDA system. Let us start dissecting the major setbacks in this code. We will introduce some theory as we talk about each problem. Obviously, we cannot tackle every issue nor perform an exhaustive analysis. The main goal is to accentuate some crucial techniques so that you don't repeat similar mistakes in your product. Another aim is to demonstrate how the cause-effect chain works regarding unintentional technical debt and evolution.

The code is the reflection of the classical proverb "*birds of a feather flock together*." In other words, it mirrors the organization's spirit pertaining to quality. If there is no incentive from the top management to seek quality in all areas, then quality will lag. The empty phrase "*we have top quality products*" in this case means nothing and cannot attain quality by repeating it on the company's website, marketing brochures, etc. People are part of an organization. A professional software engineer obeying the code of ethics of the profession [3] should strongly oppose to doing a shoddy work. The catchphrase "*they told me to develop something quickly*" is a lame excuse. A true professional will leave the company where achieving a high-quality result is impossible or providing feedback about concerns is penalized. Software engineers who are content with the corporation's low-quality viewpoint aren't behaving professionally.

Companies where quality is an afterthought don't perform risk management either. Version 1.0.0 is also the consequence of bad risk handling. They had assigned the job to an inexperienced developer without making periodic reviews

to assure that he is properly implementing the specification. He was left alone to make all necessary decisions.

The crux of the problem starts at line 9. The decision to inherit from the `ArrayList` class is the classical example of breaking the *Liskov Substitution Principle*(LSP) [1] (see also Chap. 2). The `Matrix` class is an *abstract data type* (ADT), which denotes a real-world concept from mathematics. It has an internal representation (dimensions and the content of the matrix) and a set of operators acting on that representation. The idea is to hide internal data from users of the class and only allow access to data via operators.¹ This would guarantee consistency and integrity of data. We say that the set of operators comprise the application programming interface (API) of the class. Data types may have relationships with other types. Inheritance is the way to establish a strong *is-a* relationship between two entities in an object-oriented design. Java supports a single class inheritance model via the `extends` keyword. The child inherits all accessible data and operators from its parent. Of course, if the parent class is inherited from another class, then the leaf class cumulatively inherits everything from the chain. At any rate, the *is-a* relationship suggests that the child may be substituted for any of its ancestors. We can also formulate this notion through the API, which is an important complementary aspect. We presume that the child has an API compatible with any of its ancestors, and vice versa.² This is basically the idea of the LSP. If this principle is broken, then all sorts of problems emerge (this will be demonstrated in a later section).

In our case, the motivation to inherit from the `ArrayList` (a list is also an abstract data type) is to achieve speedup in the implementation. However, such “productivity” gain is vain, as the penalty is too high. Interestingly, situations when LSP is blatantly broken in practice aren’t that rare. The Java Development Kit (JDK) itself has couple of educational specimens. `Properties` is extended from `Hashtable`, which is unfortunate, as the former only handles key value pairs that are both strings. Someone can easily blow up the integrity of the `Properties` by adding a non-string pair. Another, even more obvious example is of the `Stack` extending the `Vector` class. This is like our case here of the `Matrix` extending the `List` abstraction. Now, this last musing opens an interesting question. If a developer

¹The hiding of internal details is known as *information hiding*, and the associated protection mechanisms in place as *encapsulation*. Inheritance is said to break encapsulation since you need sometimes to relax the protection facility of the class to allow the child to inherit nonpublic stuff.

²I’ve found in books and articles a totally wrong example of an inheritance between `Rectangle` and `Square` abstractions. This problem is best manifested by trying to specify the API of these entities. The `Rectangle` has `width` and `height` as properties, while the `Square` cannot have these. Ignoring one of them, or making sure that they are always equal, is simply awkward. Imagine the surprise of a client who tries to manipulate the `Square` as a `Rectangle`. He sets `width` and `height` to different values, calls the `area` function, and gets back `width2` or `height2` instead of `width * height`. Not to mention the case of the `Square` implementation by throwing an `IllegalStateException` in an attempt to set `width` and `height` differently. For a client, one option is to start defending against such surprises by inserting the `if (rectangle instanceof Square)` check, which is an act of despair.

had been honestly trying to look for examples from referential sources (what should be better for Java than the JDK itself) and if that source contains `Stack` extends `Vector`, then who should we blame? The `Stack` mistake has been present since JDK 1.0. Here is an excerpt from its Javadoc documentation as of Java SE 8 (after nearly two decades):

The `Stack` class represents a last-in-first-out (LIFO) stack of objects. It extends class `Vector` with five operations that allow a vector to be treated as a stack. The usual push and pop operations are provided, as well as a method to peek at the top item on the stack, a method to test for whether the stack is empty, and a method to search the stack for an item and discover how far it is from the top.

When a stack is first created, it contains no items.

A **more complete and consistent** set of LIFO stack operations is provided by the `Deque` interface and its implementations, which should be used in preference to this class. For example:

```
Deque<Integer> stack = new ArrayDeque<Integer>();
```

Is it fair to only say that a “more complete and consistent” implementation is available, or would it be better to frankly admit that `Stack` is a terrible violation of the LSP? Does this text above suggest that breaking the LSP is merely an annoyance?

On line 8, the developer has accepted the proposal from its Eclipse Integrated Development Environment (IDE) to insert that statement. Suppressing warnings in this fashion is very dangerous, especially when it is done as a pure reaction to silence the IDE or compiler. That warning is related to the missing serial version identifier, which might be a problem if the instance of the `Matrix` class is targeted for serialization. The `ArrayList` implements the `Serializable` interface, and this aspect was automatically inherited when it shouldn’t have been (the Java SE 8 API documentation extensively explains the reasons why the default `serialVersionUID` value isn’t appropriate for production and how it hinders evolution).

Lines 10–16 highlights the problem with naming fields and variables. When you use such cryptic names, you also require superfluous comments like those on lines 15 and 16. The developer should have named `n` as `columns` and `m` as `rows`. This is the application of the *read-time convenience* over *write-time convenience* principle. Using longer and more descriptive variable names lessens the accidental complexity of the code, as you don’t need to remind yourselves what those abbreviations mean. Moreover, it reduces the chance of introducing an error by mixing up `n` and `m`. Finally, on lines 10 and 11, the private fields should be `final` to make the intent more expressive. A nonfinal field suggests that its value may change over the lifetime of an object, but this isn’t the case here. Therefore, the code must be written in a manner to convey these assumptions clearly. Again, this extra care also protects you from inadvertently altering the matching field’s value.

Lines 26–38 introduce three methods for dealing with elements of the matrix. The `elementIndex` calculates the index into an array based upon the coordinates of the element. The `getElement` reads out the value of an element, while `setElement` sets the element's value. Notice the cryptic usage of parameter names `i` and `j`. You can only decipher their meaning by looking deeper into the implementation. What an abomination! Imagine how hard it is to use these methods if you must ensure that the ordering of values is right when making calls. The compiler or IDE cannot help you. The `setElement` returns the old value, as is also the case with the `ArrayList`'s `set` method. Nonetheless, there was no conscious decision about this by the developer. The sheer heuristic was to radiate out whatever the parent does. An API cannot just happen by chance; it must be carefully designed based upon use cases. There is no bigger danger to spoil evolution than a bad publicly exposed API. Look at the JDK's `Thread` class and its deprecated methods: `resume`, `destroy`, `suspend`, `stop`, and `countStackFrames`. They are still part of the code base, require maintenance, and will probably never die out.

Lines 40–85 implement the matrix addition, multiplication, and multiplication with a scalar value (an operation known as scaling). These return a new instance of the matrix that holds the result of the operation. This is in par with the functional programming paradigm with immutable data structures. However, this concept of immutability isn't consistently followed. The previously mentioned `setElement` does mutate the current instance. An inconsistent API is troublesome for clients, as they need to switch context as they browse the API. There is another insidious inconsistency in these methods, as a direct consequence of cryptic variable names. In the `multiplyMatrix` and `multiplyWithScalar` methods, the variable `i` denotes rows and `j` denotes columns. This is exactly the opposite in the `addMatrix` method. Look inside this method and observe the call like `getElement(j, i)`. This style surely cannot scale.

Observe the awkward methods names for the arithmetic operators. The developer has properly reckoned that the `add(Matrix matrix)` variant would be confusing since the `ArrayList` already contains an `add` method (these add methods are semantically totally different and wouldn't fit as overloaded alternatives). For this reason, he has decided to introduce the `Matrix` suffix. This is again a fine illustration of what happens when the LSP is disobeyed. One incongruous decision gives rise to further illogical measures.

The consistency of the coding style is very important. A standard established between team members greatly reduces accidental complexity. Observe that all over the code base, you have block markers (denoted as `{` and `}`) even if the block consists of one statement. However, this isn't the case on line 78. This was perhaps a typo that could be easily overlooked during maintenance (even by the same person). This is where tool support is indispensable. A similar issue appears on lines 89–94 and 97–102, although the closing brace is misplaced on lines 94 and 102. These two misplacements don't cause errors but are surely misleading.

Besides the coding style, it is also important to remain faithful to the class design. The developer's original intent was to manage elements via the

`getElement` and `setElement` methods (see also Exercise 2). He has almost succeeded. Regrettably, the `multiplyMatrix` directly calls the `set` method instead of invoking the `setElement`, as the `addMatrix` and `multiplyWithScalar` do. The same problem is present in the tests too.

The main method contains some testing code. Observe the low quality of these tests. All of them use integers converted to double and so work only with a subset of the values from the real numbers domain. Both test matrices are square matrices. This is again just a small subset of possible matrices. The output must be manually evaluated and is rather obscure. You need to consider the implementation of the main method to understand it. What is the meaning of that `false` value on the console? Well, you must examine line 115 to find out. On lines 104–110, the developer redundantly called the `toString` method. This should be avoided as any unnecessary clutter lowers readability. A similar mistake is done by marking methods inside an interface as `public`, which is a discouraged habit.

Finally, lines 18–23 showcase what happens when the specification is vague and contains vacuums. Any executable code (the execution model doesn't matter) is totally specified from the machine's viewpoint. Accordingly, we cannot implement the system without filling in any missing details from the specification. Discrepancies arise from our understanding and anticipation of what the machine is supposed to do compared to what it does as well as what others expect. In our case, the developer has surmised that initializing the matrix with zeros is useful. After all, every declared field in Java is initialized to some default value (concretely, a double member field will be `0.0` if not otherwise defined). However, when this kind action is kept secret, nobody guarantees that the same behavior will exist in future versions. This is the reason why relying on undocumented features is hazardous. The next subsection elaborates this challenge with more details.

5.2.1 Behavior Radiation

Every software bears a behavior that becomes observable during execution (otherwise, the client may question the utility of the system). There are generally two categories of behavior: publicly announced through APIs and unconsciously radiated. To make this more comprehensible, look at Figs. 5.1 and 5.2. It shows two types of cables: an unshielded twisted pair (UTP) and an HDMI cable. Ideally, we would like our APIs to be as explicit as possible, as cable manufacturers would like their cable to be as fully shielded as possible. However, details are omitted (see Exercise 3), or we simply don't know how to safely insert more constraints without making the system too rigid. At any rate, if the client observes an undocumented behavior, then unintentionally we have a crosstalk (in electronics, it is a phenomenon where the transmitted signal in a wire induces a side effect in another circuit). Any change that we make in respect to the observed behavior will generate unpleasant effects on the client's side. For example, if some client counted on having the matrix initialized with zeroes, and we alter this assumption in a later release, then the corresponding client will break.



Fig. 5.1 A UTP cable cannot prevent crosstalk and is like a system where APIs are lax and expose too many internal details



Fig. 5.2 A fully shielded HDMI cable eliminates electromagnetic interference in the same manner as the proper API embodies information hiding and encapsulation to protect clients from entropy

What can we do if we have a crack in the shielded cable or when we leak out internal behavioral details? In both cases, we try to herald the offending hatch as a trait if that is important to the coalition of systems. In the case of an API, two preconditions must be met:

- The client must provide feedback about the crosstalk.
- We must transform the leakage into an explicit detail in the API.

There is a formal procedure to accomplish the above-mentioned feat. It is called Consumer-Driven Contracts [4]. It is a paradigm that has its origin in service-oriented architectures (SOA) but can be applied in other contexts as well. The idea

is very simple. A consumer provides information to the producer what it expects from the system. Preferably, such a feedback is handed over as a set of automated tests. In our case, a client who is interested in preserving the initialization of a matrix with zero could pass the unit test below to the developer of the Matrix class. The producer of the Matrix class would afterward ensure that all future releases will perform this initialization:

Listing of the Matrix Initialization Consumer-Driven Contract (Imports Are Skipped)

```
public class MatrixInitializationConsumerDrivenContract {  
    @Test  
    public final void initializeMatrixWithZeros() {  
        Matrix matrix = new Matrix(3, 3);  
        for (int rows = 0; rows < 3; rows++) {  
            for (int columns = 0; columns < 3; columns++) {  
                assertEquals(  
                    0.0,  
                    matrix.getElement(rows, columns),  
                    0.0);  
            }  
        }  
    }  
}
```

5.3 Non-square Matrix Multiplication Bug: Problem

Soon after releasing version 1.0.0, an outrage occurred among clients of the Matrix class. They have stumbled across a bug when trying to multiply nonsquare matrices. Their code had spilled out the following stack trace (this was for the case of multiplying 3×4 and 4×3 matrices):

```
Exception in thread "main"  
java.lang.IndexOutOfBoundsException: Index: 12, Size: 12  
at java.util.ArrayList.rangeCheck(ArrayList.java:653)  
at java.util.ArrayList.get(ArrayList.java:429)  
at com.example.eda.linalg.Matrix.getElement(Matrix.java:31)  
at com.example.eda.linalg.Matrix.multiplyMatrix(Matrix.java:59)  
...
```

Although these users were local, the bug had caused them a latency in the project. They were blocked and demanded a hot fix. The management had immediately assigned the bug to the first available developer (the original developer wasn't vacant).

5.4 Non-square Matrix Multiplication Bug: Resolution

The first thing that a developer should do is to try to reproduce the error in his environment. In this case, this wasn't difficult. The developer has modified the tests in the `main` method to use 3×4 and 4×3 matrices instead of 3×3 (see Exercise 5). After receiving the same exception, he has started considering the `multiplyMatrix` method (the localization of the bug was quite quick). So far, everything went smoothly. He has fired up his debugger and stepped through the code. He soon realized that the method wrongly referenced `n` instead of `other.n` inside the second `for` loop. He has hoped that this is the only problem. Unfortunately, the exception didn't go away. After spending a lot of time to fully understand the `elementIndex`, `getElement`, and `setElement` methods (including the inherited `get` and `set` methods), he has finally concluded that the `elementIndex` method was improperly called on the `this` instance instead of the newly created `Matrix`.

The new release went out with version 1.0.1. Below are the fixed `multiplyMatrix` method, the altered tests, and their output:

Listing of the Fixed `multiplyMatrix` Method (Fixes Are Bolded)

```
public Matrix multiplyMatrix(Matrix other) {  
    Matrix newMatrix = new Matrix(m, other.n);  
    for (int i = 0; i < m; i++) {  
        for (int j = 0; j < other.n; j++) {  
            for (int k = 0; k < n; k++) {  
                newMatrix.set(  
                    newMatrix.elementIndex(i, j),  
                    newMatrix.getElement(i, j) +  
                    getElement(i, k) *  
                    other.getElement(k, j));  
            }  
        }  
    }  
    return newMatrix;  
}
```

Listing of the Altered Tests in the Main Method

```
public static void main( String[] args ) {  
    Matrix a = new Matrix(3, 4);  
    for (int i = 0; i < 3; i++)  
        for (int j = 0; j < 4; j++) {
```

```
a.set(a.elementIndex(i, j), (double) (i + j));  
}  
  
Matrix b = new Matrix(4, 3);  
for (int i = 0; i < 4; i++)  
    for (int j = 0; j < 3; j++) {  
        b.set(b.elementIndex(i, j), (double) (i + j));  
    }  
  
System.out.println(a);  
System.out.println(b);  
System.out.println(a.multiplyWithScalar(10.0));  
System.out.println(a.multiplyMatrix(b));  
System.out.println(a.addMatrix(b));  
System.out.println(a.equals(b));  
System.out.println(a.hashCode());  
a.setElement(2, 2, 111.0);  
System.out.println(a.getElement(2, 2));  
System.out.println(a.equals(b));  
System.out.println(a.hashCode());  
}
```

The Output of the Test Run

```
[0.0, 1.0, 2.0, 3.0, 1.0, 2.0, 3.0, 4.0, 2.0, 3.0, 4.0, 5.0]  
[0.0, 1.0, 2.0, 1.0, 2.0, 3.0, 2.0, 3.0, 4.0, 3.0, 4.0, 5.0]  
[0.0, 10.0, 20.0, 30.0, 10.0, 20.0, 30.0, 40.0, 20.0, 30.0, 40.0,  
50.0]  
[14.0, 20.0, 26.0, 20.0, 30.0, 40.0, 26.0, 40.0, 54.0]  
[0.0, 2.0, 4.0, 4.0, 2.0, 4.0, 6.0, 6.0, 4.0, 6.0, 8.0, 8.0]  
false  
161416833  
111.0  
false  
315311745
```

The developer has removed the extra `toString` method calls from the tests. This is a viable tactic. Instead of asking for a huge cleanup endeavor (there is a possibility that it will be refused by the management, as it carries a considerable risk of introducing breaking changes), you may want to use each bug fixing session as an opportunity to continuously improve the existing code base. Nevertheless, as this was an urgent hot fix; he had no time to transform these tests into automated unit test cases.

This bug was a corollary of the rush work in the initial release. The extra wasted time and money on this bug fix should have been spent on the initial release with a tremendous positive effect. In this case, the code is still of a low quality, and the next developer would again spend precious time to understand it.

5.5 Inadequate Dimensionality: Problem

After a while, just before the release date of the EDA software, the QA department had noticed a strange behavior of the system. For some class of input, the system produced an output which was complete garbage, in a sense that the calculations were wrong. In some other occasions, it had reported an error with a strange stack trace (at least, they didn't understand what the heck an `IndexOutOfBoundsException` means). The team responsible for the development of the EDA system was trying to localize the issue. After a lot of struggle, they had managed to find a bug in their code, which had caused the `Matrix` class to fail. However, the problem was quite subtle. For some type of input, the `Matrix` had thrown an exception, but for the other, it had just emitted garbage. The EDA team considered this as a completely unacceptable behavior. They had complained that they would have had caught this bug much earlier if the `Matrix` class had reported back an exception for the wrong input.

5.6 Inadequate Dimensionality: Resolution

The management has immediately reacted and assigned a more experienced developer to fix the problem. As before, the first task he needed to do was to reproduce the bug. He has experimented with the test cases and came up with the following two scenarios that reflected the bug report:

- For the 4×4 and 4×3 matrices, the `addMatrix` throws an exception.
- For the 4×3 and 5×3 matrices, both `addMatrix` and `multiplyMatrix` produce garbage.

Here is the listing of the modified tests. Notice that this time, the developer has decided to create automated unit test cases to prevent regression. He has also included the special case of working with square matrices (see Exercise 7). In the first run, some of the tests below were red since they have exposed the bugs. After the fix, they have turned to green:

Listing of the Modified Tests (Imports Are Skipped)

```
public final class MatrixUnitTest {  
    private final Matrix matrix4x4 = new Matrix(4, 4);
```

```
private final Matrix matrix4x3 = new Matrix(4, 3);
private final Matrix matrix5x3 = new Matrix(5, 3);

private void fillMatrix(
    Matrix matrix, int rows, int columns) {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < columns; j++) {
            matrix.setElement(i, j, (double) (i + j));
        }
    }
}

@Before
public void initializeMatrices() {
    fillMatrix(matrix4x4, 4, 4);
    fillMatrix(matrix4x3, 4, 3);
    fillMatrix(matrix5x3, 5, 3);
}

@Test
public void checkPrintoutOfMatrix() {
    assertEquals(
        "[0.0, 1.0, 2.0, " +
        "1.0, 2.0, 3.0, " +
        "2.0, 3.0, 4.0, " +
        "3.0, 4.0, 5.0]",
        matrix4x3.toString());
}

private void validateContent(
    Matrix matrix, int rows, int columns,
    double tolerance, double... elements) {
    int idx = 0;

    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < columns; j++) {
            assertEquals(
                elements[idx++],
                matrix.getElement(i, j),
                tolerance);
        }
    }
}

@Test
public void multiplyMatrixWithScalar() {
    final Matrix result =
```

```
        matrix4x3.multiplyWithScalar(10.0);
    assertNotNull(result);
    validateContent(
        result, 4, 3, 0.0,
        0.0, 10.0, 20.0,
        10.0, 20.0, 30.0,
        20.0, 30.0, 40.0,
        30.0, 40.0, 50.0);
    }

@Test
public void multiplyMatrixWithItself() {
    final Matrix result =
        matrix4x4.multiplyMatrix(matrix4x4);
    assertNotNull(result);
    validateContent(
        result, 4, 4, 0.0,
        14.0, 20.0, 26.0, 32.0,
        20.0, 30.0, 40.0, 50.0,
        26.0, 40.0, 54.0, 68.0,
        32.0, 50.0, 68.0, 86.0);
}

@Test
public void multiplyMatrix4x4With4x3() {
    final Matrix result =
        matrix4x4.multiplyMatrix(matrix4x3);
    assertNotNull(result);
    validateContent(
        result, 4, 3, 0.0,
        14.0, 20.0, 26.0, 20.0,
        30.0, 40.0, 26.0, 40.0,
        54.0, 32.0, 50.0, 68.0);
}

@Test
public void addMatrixToItself() {
    final Matrix result = matrix4x4.addMatrix(matrix4x4);
    assertNotNull(result);
    validateContent(
        result, 4, 4, 0.0,
        0.0, 2.0, 4.0, 6.0,
        2.0, 4.0, 6.0, 8.0,
        4.0, 6.0, 8.0, 10.0,
```

```
    6.0, 8.0, 10.0, 12.0);
}

@Test
public void compareMatrices() {
    assertTrue(matrix5x3.equals(matrix5x3));
    assertFalse(matrix4x3.equals(matrix5x3));
}

@Test
public void getMatrixHashKey() {
    assertEquals(1599200769, matrix4x4.hashCode());
}

@Test
public void readingAndWritingMatrixElements() {
    assertEquals(4.0, matrix4x4.getElement(2, 2), 0.0);
    matrix4x4.setElement(2, 2, 111.0);
    assertEquals(111.0, matrix4x4.getElement(2, 2), 0.0);
}

@Test(expected = IllegalArgumentException.class)
public void addMatricesWithInvalidDimensions() {
    matrix4x4.addMatrix(matrix4x3);
}

@Test(expected = IllegalArgumentException.class)
public void multiplyMatricesWithInvalidDimensions() {
    matrix4x3.multiplyMatrix(matrix5x3);
}
}
```

The experienced developer knew very well that the test cases should adhere to the same quality standards as the real code. A bug in the test code may consume the same amount of energy as a defect in the production code. A good battery of tests could also serve as an excellent reference, so they should be lean and tidy. The `fillMatrix` and `validateContent` private methods are introduced to avoid duplication, which is very important during maintenance. Less code means less maintenance costs. Don't forget that you must keep your tests always up to date with the production code. Otherwise, their value will quickly evaporate.

After bumping up the version number to 1.0.2, the developer has submitted the following fixes to the `addMatrix` and `multiplyMatrix` methods, respectively:

```
if (other.n != n || other.m != m) {
    throw new IllegalArgumentException(
        "The dimensions of matrices don't match");
}

if (other.m != n) {
    throw new IllegalArgumentException(
        "The dimensions of matrices don't match");
}
```

The above checks are part of the philosophy called *Defensive Programming*. The idea is to incorporate safety fences in your code to guard it from erroneous input. As this example has demonstrated, this extra protection could be helpful for clients too. After all, it is usually better to alert users as soon as possible than to silently trash around. This strategy is known as *fail-fast*.

In general, such a decision should be part of an architecture, and you can tune your exception handling policy for *correctness* or *robustness*. For example, if you are reading out temperature sensor values and you miss a single reading, this isn't a catastrophe (as when the user tries to multiply matrices with invalid dimensions). You wouldn't want to fail here. Therefore, you should prefer robustness and just use the cached value (the correctness wouldn't be jeopardized since successive sensor readings are anyhow near to each other). Of course, when such misses start to accumulate and rise above some threshold, you would signal a problem.

5.7 Retrieve Meta Data: Problem

Finally, the EDA software was launched and it started to attract attention. Some clients reported back issues with the performance, while a couple of them demanded advanced features. In the meantime, the developers of EDA had also filled in their backlog with the next enhancements to the software. The initial release only supported single modeling sessions, i.e., it was not possible to work with multiple electronic designs in parallel. At any rate, the developers of EDA (the clients of the `Matrix` class) needed a way to easily retrieve dimensions of the matrix. This was not possible with the current version.

5.8 Retrieve Meta Data: Resolution

Reading out the dimensions of the matrix is an obvious request that was completely overlooked in the original design. Well, the omission may be amazing, but considering the poor requirements work at the beginning, this shouldn't surprise us. The developers of the EDA system were very busy and had no time to wait for another release of the `Matrix` class. Of course, they had issued a request for an extension, but in parallel, they had been laboring on the work-around. As the clients of the

Matrix class were diverse (the EDA development department was comprised of multiple distributed teams), each of them came up with a different solution.

This example showcases what happens when you don't think enough about reusability and extensibility of your classes. The Matrix class was not final and had a huge exposed API (the consequence of breaking the LSP by carelessly extending the ArrayList class). If you give out such a class to clients (the same story also applies to coarser-grained reusable units), some of them will use it in a way that you would never imagine. Therefore, it is better to limit the options than to deal with a chaos.

The next listing shows three variations of solving the dimension retrieval facility in the Matrix class. The first one is using the classical redundancy of data to provide back dimensionality. The second one is relying on reflection to access the private members of the parent class.³ This is a very dangerous approach, as reaching out to private members will make your code brittle. However, the third approach is ingenious (see Exercise 8). This client has found out that adding an extra item to the underlying collection doesn't impact the normal operation of the matrix. On the other hand, it allows someone to store additional metadata, such as the number of rows (the number of columns is easily derived from the size of the collection and the number of rows):

Abbreviated Listings of the Various Solutions to Get the Dimensions of the Matrix

```
public class MatrixWrapperA extends Matrix {  
    private final int rows;  
    private final int columns;  
  
    public MatrixWrapperA(int rows, int columns) {  
        super(rows, columns);  
        this.rows = rows;  
        this.columns = columns;  
    }  
  
    public int getRows() {  
        return rows;  
    }  
  
    public int getColumns() {  
        return columns;  
    }
```

³You can use `javap -c <class file name>` to disassemble the Java class file. Also, you may use your IDE to reveal the structure of the compiled class.

```
        }
    }

public class MatrixWrapperB extends Matrix {
    public MatrixWrapperB(int rows, int columns) {
        super(rows, columns);
    }

    public int getRows()
        throws NoSuchFieldException,
               SecurityException,
               IllegalArgumentException,
               IllegalAccessException {
        final Field rows =
            Matrix.class.getDeclaredField("m");
        rows.setAccessible(true);
        return rows.getInt(this);
    }

    public int getColumns()
        throws NoSuchFieldException,
               SecurityException,
               IllegalArgumentException,
               IllegalAccessException {
        final Field columns =
            Matrix.class.getDeclaredField("n");
        columns.setAccessible(true);
        return columns.getInt(this);
    }
}

public class MatrixWrapperC extends Matrix {
    public MatrixWrapperC(int rows, int columns) {
        super(rows, columns);
        add((double) rows);
    }

    public int getRows() {
        return (int) Math.round(get(size() - 1));
    }

    public int getColumns() {
        return (int) Math.round(size() / get(size() - 1));
    }
}
```

The accompanying source code contains unit tests for the above variants. From now on, clients will reference their wrappers instead of the `Matrix` class. This means they will need to switch back to `Matrix` once it gets embellished with a method to retrieve the dimensions. You should have been persuaded by now that a messy API complicates both the producer's and consumer's lives.

5.9 Concurrency Extension: Problem

The developers of ACME EDA had requested a concurrent version of the `Matrix` class, as the original wasn't thread-safe.⁴ They had demanded a separate class. They wanted to keep the original in single threaded situations for performance reasons.⁵ All this concurrency support was planned to be shipped with the next release of the EDA software that would utilize the inherent parallelism in modern multicore computers.

5.10 Concurrency Extension: Resolution

The task was assigned to a new developer, who knew nothing about the `Matrix` class. So, he first needed to understand it. You should have noticed already that the same piece of code was learned up so far by different software engineers. This underpins the observation that software engineers spend most of their time on analyzing and grasping the code base. Thus, a high-quality code is the prime enabler of productive maintenance and evolution.

The developer has cloned the implementation of the `Matrix` class and renamed it as `ConcurrentMatrix` (again using the JDK as a motivating reference that contains classes like `ConcurrentLinkedQueue`, `ConcurrentHashMap`, etc.). He then started to analyze the interaction between the `ConcurrentMatrix` and `ArrayList` classes. He has realized that the connection is represented via the `get` and `set` methods, which should be ideally confined inside the `getElement` and `setElement` methods. However, as he saw that `set` is also called directly, he thought that protecting the `get` and `set` methods would automatically guarantee

⁴It could be used by multiple threads if, once initialized, it would never undergo a concurrent mutation through the `setElement` method. At this stage, nobody was contemplating use cases, which is the first thing you should do before touching an API.

⁵The JDK also leverages a similar tactic with the `StringBuffer` and `StringBuilder` classes. The former is thread-safe, while the latter is not. The JDK documentation recommends that you should use `StringBuilder` whenever it is possible, as it will be faster under most circumstances. Now, blindly mimicking the way things are done at some other places isn't a smart scheme. You must evaluate the cost/benefit ratio. Moreover, you must base your decision on facts, and in the case of performance, these must emanate from measurements. Without profiling your system, there is no way to exactly know where are the bottlenecks. Intuition may terribly fail with performance optimization attempts.

thread-safety. He had been pondering what is the minimalist change that would fulfill this. After browsing the JDK classes, he has recalled that the `Vector` class has the same API as the `ArrayList` class (after `Vector` got retrofitted into the `Collections` library). So, he just altered the `ConcurrentMatrix` to extend `Vector` instead of `ArrayList`. The `Vector` class is thread-safe, although its efficacy isn't that great.

Finally, he has also cloned the tests and run them against the new class. After being assured that everything is ok, he published a new version of the linear algebra package (this time containing two classes) with version 1.1.0.

5.10.1 Duplication vs. Reuse

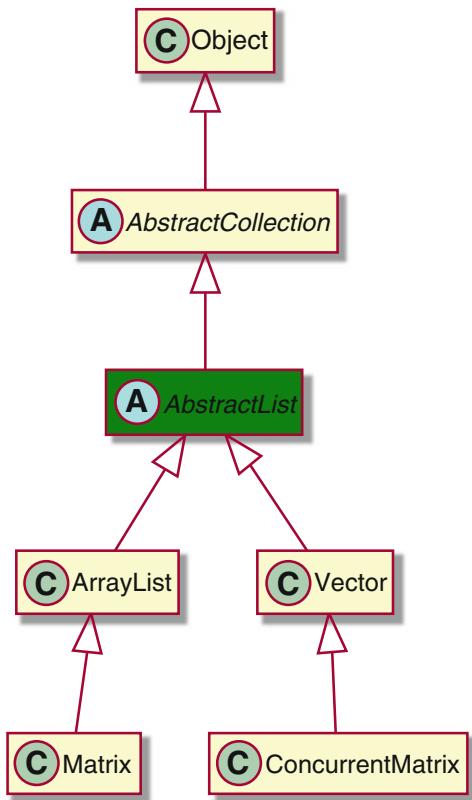
Should we regard duplication as reuse? After all, the developer has quickly copy-pasted the existing code and made a single change to it. Well, copy-paste may deceptively look like reuse, but it is not, as it hinders maintenance and evolution. Recall the 15% rule regarding the volume of code changes in each upgrade. With duplication, you are increasing this quantity. Nonetheless, don't forget that the tests are also part of the code base, and they were duplicated too.

Someone may argue that with copies, you don't need to care about design tradeoffs linked with a shared code base. While this alleged freedom may seem enticing, the price is astonishing. Don't forget that you are responsible for keeping all copies in good health and knowing exactly where are those copies and how they evolve. As a counterargument, I've heard this many times: managers reply that people continuously fork open-source projects. Isn't that an allowable duplication-based reuse? Well, the answer is yes for a very simple reason. You are not responsible for those forks, so you shouldn't care how many of them exist in the observable Universe.

5.10.2 Similarity of Code vs. Class Hierarchy

It is instructive to compare the similarity of the `Matrix` and `ConcurrentMatrix` classes from two perspectives: source code and class inheritance diagram. At the level of the source code, these classes are more than 99% alike (the same is true for the `MatrixUnitTest` and `ConcurrentMatrixUnitTest` classes that you may find in the accompanying source code of this book). However, the corresponding Unified Modeling Language (UML) class diagram is striking, as depicted in Fig. 5.3. It tells us a completely dissimilar message. It appears that a nearly identical code is far away in the class hierarchy. This highlights the danger of using purely reverse engineering tools to report back the status of the software. You must also consider the source code to seize the situation. At any rate, if you notice this kind of discrepancy, then it should be an alarm that the code isn't in proper shape.

Fig. 5.3 The nearest common ancestor of the `Matrix` and `ConcurrentMatrix` classes is the `AbstractList` class. Ironically, none of them is a list, and a client holding the reference to the `AbstractList` class cannot do anything useful with the matrix objects



5.10.3 Problem Space vs. Solution Space

This wisdom applies to requirements work during the initial development as well as while handling change requests. The problem space should talk about the problem itself and only contain descriptions related to the *What* aspect. The solution space should elaborate the implementation, i.e., speak about the *How* part. When these are mixed up, all sorts of issues pop up. Keep in mind that people are very bad at expressing their problems. Often, they will present the problem in the form of a solution that even if you fully satisfy would not solve their real difficulty.

In our case, the request to deliver a concurrent version of the `Matrix` class belongs to the solution space. Instead of continuing to address this request, the maintainer should have had pushed back with the *Why* question (see [6] for the 5 *Whys* method to determine the root cause). If you miss an opportunity to decipher what you truly need to solve, then any “solution” will just trigger a new wave of badly articulated change requests (as the next section will exemplify). The reason is simple. You don’t know what bothers the client, and whatever you deliver isn’t a solution, even if you think it is perfect (read [7] for a superb narrative regarding this phenomenon).

5.11 Concurrency Bug: Problem

The developers of the EDA system were once again slowed down. They had noticed a strange behavior of the `ConcurrentMatrix` class, when used in a heavily multithreaded setup. Namely, the arithmetic operations had sporadically returned invalid results. The bug report stated that the failure usually happens if another thread mutates one of the participant matrices in the current operation. The EDA developers had demanded an exception whenever the `ConcurrentMatrix` class detects a collision that will surely cause an invalid output. At any rate, they had opted for the optimistic locking mechanism opposed to pessimistic locking due to performance reasons.⁶

As this seemed to be a “classical” bug report, it got assigned to a junior software engineer. This is a usual approach in many companies. Hotshots are reserved for greenfield projects, while maintenance is left to inexperienced personnel. After all, they should just “fix” things. This reduces the morale of the maintenance team, as their job isn’t valued as much as of those who are doing initial development. However, if the first version of the software isn’t properly done (as in our case), then the roles are immediately switched. The guys working on maintenance and evolution are undertaking the proper development, rather than the team that delivered the original version. Moreover, evolution is mostly comprised of all sorts of enhancements, so it isn’t purely “fixing” stuff at all. Sometimes, even a fix necessitates considerable rework.

5.12 Concurrency Bug: Resolution

The software engineer has uncritically followed the instructions, and after a while, he has figured out the “solution.” The trick was to utilize the fail-fast feature of the iterator of the parent class. Namely, the `Vector` class gives back an iterator over array elements that throws the `ConcurrentModificationException` exception when it detects concurrent modification of the underlying collection. In other words, it signals back an exception if during iteration, another thread alters the content of the collection. This appeared as a miraculous answer to the optimistic locking proposal from the EDA development team. The next listing shows the

⁶Pessimistic locking is a tactic where the resource is locked down during the whole operation. When such locking is coarse grained (including the whole resource) and the operation takes a bit longer to execute, it may have adverse effects on performance (other threads may wait to grab the lock instead of doing something useful). The optimistic locking doesn’t lock down the resource in advance but allows the operation to execute on a resource that might get undesirably modified in the meantime. If such a conflict occurs, then the operation is aborted and must be retried. Optimistic locking is generally suitable when the probability of a conflict is low. This probability should be judiciously estimated based upon measurements (simulation is a very handy way to acquire measurements). In our case, the previous probability was derived from a gut feeling (a very bad estimator) of the development team.

updated methods from the `ConcurrentMatrix` class. Of course, these were packed inside version 1.1.1 of the linear algebra library:

Listing of the Modified Methods in the `ConcurrentMatrix` Class

```
private ConcurrentMatrix copy() {
    ConcurrentMatrix newMatrix = new ConcurrentMatrix(m, n);
    newMatrix.clear();
    for (double element : this) {
        newMatrix.add(element);
    }
    return newMatrix;
}

public ConcurrentMatrix addMatrix(ConcurrentMatrix other) {
    if (other.n != n || other.m != m) {
        throw new IllegalArgumentException(
            "The dimensions of matrices don't match");
    }

    ConcurrentMatrix newMatrix = new ConcurrentMatrix(m, n);
    ConcurrentMatrix thisCopy = copy();
    other = other.copy();
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            newMatrix.setElement(
                i, j,
                thisCopy.getElement(i, j) +
                other.getElement(i, j));
        }
    }
    return newMatrix;
}

public ConcurrentMatrix multiplyMatrix(
    ConcurrentMatrix other) {
    if (other.m != n) {
        throw new IllegalArgumentException(
            "The dimensions of matrices don't match");
    }

    ConcurrentMatrix newMatrix = new ConcurrentMatrix(m, other.n);
    ConcurrentMatrix thisCopy = copy();
    other = other.copy();
```

```

        for (int i = 0; i < m; i++) {
            for (int j = 0; j < other.n; j++) {
                for (int k = 0; k < n; k++) {
                    newMatrix.setElement(
                        i, j,
                        newMatrix.getElement(i, j) +
                        thisCopy.getElement(i, k) *
                        other.getElement(k, j));
                }
            }
        }
        return newMatrix;
    }

    public ConcurrentMatrix multiplyWithScalar(double value) {
        ConcurrentMatrix newMatrix = new ConcurrentMatrix(m, n);
        ConcurrentMatrix thisCopy = copy();
        for (int i = 0; i < m; i++)
            for (int j = 0; j < n; j++) {
                newMatrix.setElement(
                    i, j, thisCopy.getElement(i, j) * value);
            }
        return newMatrix;
    }
}

```

He has introduced a new private method `copy` that internally uses an iterator (the `for-each` construct hides it from you). His idea was to make copies of input matrices at the entrance of arithmetic methods and work only on those copies. If the content of the matrix would vary while iterating over it, then the code would throw an exception. He has run the existing tests and all were green (see Exercise 9). This was enough for him to declare that the bug is fixed, as he has presumed that the iterator indeed behaves as expected.

He was complaining to the management that he lost a lot of time to notice the different roles of the loop control variables in the `addMatrix` method. Furthermore, he was puzzled why the `multiplyMatrix` calls `set` directly instead of using the `setElement` method. He has decided to improve the code by aligning these parts with the rest of the code base. Thanks to the consequences of duplication, he has forgotten to do the same in the `Matrix` class. Now, the two copies have started to diverge. Finally, to “save” space, he has reused the input parameter `other` and reassigned it to the `copy` of the second matrix (see the `addMatrix` and `multipleMatrix` methods).⁷

⁷This is a very bad practice where you don't save anything! Such micro-optimizations are fundamentally pointless (like messing with the ordering of `case` statements inside the `switch`

5.13 Heisenbug: Problem

The developers of the EDA system had integrated version 1.1.1 of the linear algebra package into the new release. The software had been catching concurrent modification attempts and would have restarted the whole user session had such an event occurred. At least, it was trying to avoid processing garbage. Unfortunately, the users of the EDA system had been experiencing random failures. Sometimes, the software would produce a totally unreasonable output and some other time, just an inaccurate result. Nobody knew what is going on. The bug report only stated that the software is unstable, without any further clues about how to reproduce the issue.

The above scenario is one of the worst that you can encounter. The funny name for such bugs is *Heisenbug*. This type of the bug is usually instigated by a nondeterministic behavior of the system, whose root cause is frequently linked to a concurrency defect. The most irritating fact is that a Heisenbug is extremely susceptible to environmental changes, so you cannot easily reproduce it locally (at least, not by using standard testing and debugging methods). A tiny difference in the timing introduced by a debugger may eliminate the possibility for a failure (a visible exposure of the defect). At any rate, this problem report was assigned to a senior software engineer.

5.14 Heisenbug: Resolution

The senior software engineer knew that this task can only be solved by thinking about the real need of the EDA development team as well as by analyzing the `ConcurrentMatrix` class. He didn't even try to attack the challenge by writing more automated tests. He has decided to go over first the current code base and see what it is actually doing. Such an analysis is easier with a UML sequence diagram depicting a concrete use case. Figure 5.4 shows what that diagram looks like for calculating $C = A * B$, where A and B are matrices with proper dimensions.

Suppose that we have two independent active users, U_1 and U_2 , of the linear algebra library that share matrices A and B (both possess references onto them). Assume that matrices A and B have some initial value at time t_0 . At time t_1 , the U_1 starts the multiplication by invoking `A.multiplyMatrix(B)`. U_1 wants to calculate $C = A_{t_0} * B_{t_0}$. Therefore, U_1 first creates copies of those matrices, i.e., U_1 would like to work on snapshots while calculating the product. Let us investigate how U_2 may hinder this vision. U_2 may try to alter A or B while they are copied.

The fail-fast iterator should presumably detect this condition. What about the other moments? If B is altered while A is being copied in time t_2 , then nothing unusual will happen. However, in this case, U_1 will unintentionally produce $C = A_{t_0} * B_{t_2}$. This is exactly the situation that U_1 wanted to eschew. In general, U_1

or turning switch into a nested if-else structure) since the Java compiler and the Java HotSpot Virtual Machine are much better in doing this for you. Many times, you just impede the effort of these components to do their job.

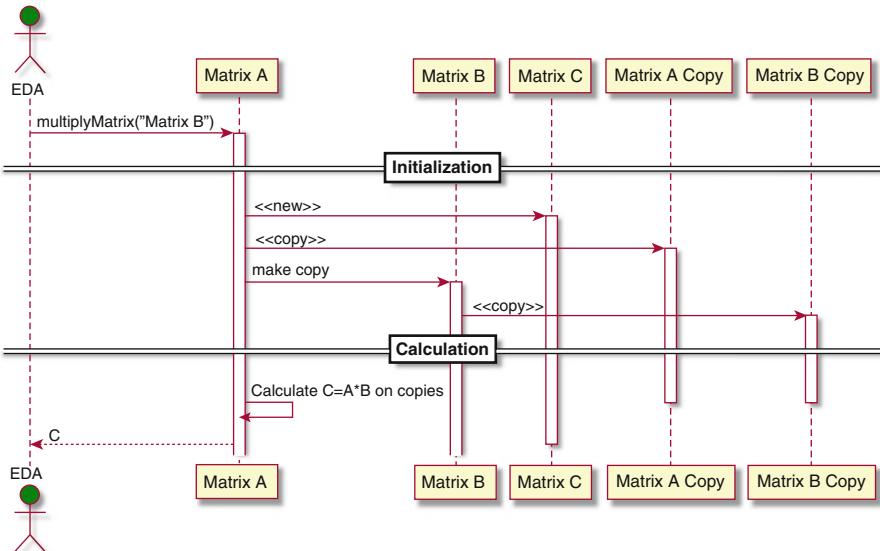


Fig. 5.4 The UML sequence diagram that shows the steps for calculating the product of two matrices. There are two standpoints that must be taken into account: detection of concurrent modifications while making copies and possible changes between actions during initialization. The calculation proceeds on the copies and is completely local to the current thread

calculates $C = A_{tx} * B_{ty}$, where $t_0 \leq t_x \wedge t_0 \leq t_y$. All in all, this isn't at all what was planned.

It is also not clear how users should react when they detect a concurrent modification. Restarting the whole session (as it was done in the latest release) negatively impacts usability, especially if such conflicts frequently occur.

There is another shocking fact in the implementation of this idea. The senior software engineer has discerned that the `copy` method is deeply flawed. The fail-fast iterator of the `Vector` class cannot guarantee the detection. Here is the excerpt from the JDK API documentation:

Note that the fail-fast behavior of an iterator cannot be guaranteed as it is, generally speaking, impossible to make any hard guarantees in the presence of unsynchronized concurrent modification. Fail-fast iterators throw `ConcurrentModificationException` on a **best-effort** basis. **Therefore, it would be wrong to write a program that depended on this exception for its correctness: the fail-fast behavior of iterators should be used only to detect bugs.**

So, the last two versions were simply a wasted effort. The developers of the EDA team had only a muddled impression what they want and had failed to communicate clearly their core problem. Moreover, they had also spoiled their code by referencing two versions of the matrix class with separate wrappers. Since the

stability of the whole system was shattered, the team had decided to perform a repair.

The senior engineer has mandated that threads must not share mutable matrices. All shared matrices must be immutable.⁸ Local instances may be mutable. Consequently, he has suggested that the team rework the current classes into `MutableMatrix` and `ImmutableMatrix`.⁹ He has also pointed out that this is still far from being ideal but will solve the stability issue for now. A cleaner solution would be to mimic the Java Collections framework by offering read-only views of matrices (see the `unmodifiable*` set of methods in the `Collections` utility class). The task of creating `MutableMatrix` and `ImmutableMatrix` classes was given to another developer. This was an incompatible change, so the version number got bumped to 2.0.0. Here are the abridged listings of these classes without private fields and a constructor (see the accompanying source code for the unit tests).

Of course, the `ImmutableMatrix` class that extends `Vector` isn't immutable per se since someone can always call the `set` method directly and mutate it. For this reason, it is still safer to extend from `Vector`, although this maneuver cannot deliver the main guarantee pertaining to immutability:

Abbreviated Listing of the `MutableMatrix` Class¹⁰

```
public class MutableMatrix extends ArrayList<Double> {
    ...
    public void addMatrix(MutableMatrix other) {
        if (other.n != n || other.m != m) {
            throw new IllegalArgumentException(
                "The dimensions of matrices don't match");
        }

        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                setElement(
                    i, j,
                    getElement(i, j) +
                    other.getElement(i, j));
            }
        }
    }
}
```

⁸This topic touches the object ownership matter. Part of the defensive programming tactic is to never return a reference to your underlying collection. Instead, you may pass back a read-only view so that you retain full ownership over your internal data structure.

⁹Immutable classes are thread-safe. Java has many examples and the most famous is the `String` class. Internally, it contains multiple optimizations to make it efficient, but the core idea is that a `String` object cannot mutate its state.

¹⁰The developer had taken time now to incorporate the change into `addMatrix` so that the loop variables are in sync with the rest of the code base.

```
        }
    }
}

public MutableMatrix multiplyMatrix(
    MutableMatrix other) {
    if (other.m != n) {
        throw new IllegalArgumentException(
            "The dimensions of matrices don't match");
    }

    MutableMatrix newMatrix =
        new MutableMatrix(m, other.n);
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < other.n; j++) {
            for (int k = 0; k < n; k++) {
                newMatrix.setElement(
                    i, j,
                    newMatrix.getElement(i, j) +
                    getElement(i, k) *
                    other.getElement(k, j));
            }
        }
    }
    return newMatrix;
}

public void multiplyWithScalar(double value) {
    for (int i = 0; i < m; i++)
        for (int j = 0; j < n; j++) {
            setElement(i, j, getElement(i, j) * value);
        }
}
}
```

Abbreviated Listing of the ImmutableMatrix Class

```
public class ImmutableMatrix extends Vector<Double> {
    ...
    // Returns back a new modified matrix.
    public synchronized ImmutableMatrix setElement(
        int i, int j, double value) {
        ImmutableMatrix newMatrix =
            (ImmutableMatrix) clone();
```

```
        newMatrix.set(elementIndex(i, j), value);
        return newMatrix;
    }

    public ImmutableMatrix addMatrix(ImmutableMatrix other) {
        if (other.n != n || other.m != m) {
            throw new IllegalArgumentException(
                "The dimensions of matrices don't match");
        }

        ImmutableMatrix newMatrix =
            new ImmutableMatrix(m, n);
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                newMatrix.set(
                    elementIndex(i, j),
                    getElement(i, j) +
                    other.getElement(i, j));
            }
        }
        return newMatrix;
    }

    public ImmutableMatrix multiplyMatrix(
        ImmutableMatrix other) {
        if (other.m != n) {
            throw new IllegalArgumentException(
                "The dimensions of matrices don't match");
        }

        ImmutableMatrix newMatrix =
            new ImmutableMatrix(m, other.n);
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < other.n; j++) {
                for (int k = 0; k < n; k++) {
                    newMatrix.set(
                        newMatrix.elementIndex(i, j),
                        newMatrix.getElement(i, j) +
                        getElement(i, k) *
                        other.getElement(k, j));
                }
            }
        }
        return newMatrix;
    }
```

```
public ImmutableMatrix multiplyWithScalar(double value) {
    ImmutableMatrix newMatrix =
        new ImmutableMatrix(m, n);
    for (int i = 0; i < m; i++)
        for (int j = 0; j < n; j++) {
            newMatrix.set(
                elementIndex(i, j),
                getElement(i, j) * value);
        }
    return newMatrix;
}
```

All methods of the immutable version return a fresh copy of the matrix. The arithmetic methods call `set` directly on a newly created matrix since it is confined inside the matching method. It would be an error to call `setElement`. You should be very careful with the `clone` method. In our case, the JDK documentation for the `clone` method of the `Vector` class states the following:

Returns a clone of this vector. The copy will contain a reference to a clone of the internal data array, not a reference to the original internal data array of this Vector object.

Moreover, since the content type of a matrix is an immutable wrapper for the primitive type `double`, using `clone` here is ok. Lastly, notice that it isn't possible to combine mutable and immutable versions in arithmetic expressions. For example, it is impossible to add an immutable matrix to a mutable one. This will be fixed later.

5.15 Printout Improvement: Problem

The EDA system had become more complex over time. The developers were working on repairing defects and implementing new features. To have better visibility, they had requested a better printout of matrices. The current functioning of the `toString` method was cumbersome since without knowing in advance the dimensions, it wasn't possible to interpret the output. Moreover, they wanted to have an ability to get dimensionality information and get rid of those wrappers. Finally, the `equals` method was rather unreliable, as it didn't take dimensionality into account.

5.16 Printout Improvement: Resolution

The developer has decided to create three separate independent tasks:

- Improving the printout of matrices
- Fixing the problem in the equals method
- Introducing the dimensionality getter methods

It is always beneficial to reduce the work in progress. Doing one task at a time allows you more control and is faster due to the fewer context switches. Here is the implementation of the new `toString` method with the unit test case (see Exercise 10):

Listing of the New `toString` Method

```
@Override
public String toString() {
    StringBuilder buff = new StringBuilder();
    for (int row = 0; row < rows; row++) {
        for (int column = 0; column < columns; column++) {
            buff.append(String.format(
                "%-15.3f", getElement(row, column)));
            if (column != columns - 1) {
                buff.append(' ');
            }
        }
        buff.append('\n');
    }
    return buff.toString();
}
```

The Unit Test for the Modified `toString` Method

```
@Test
public void checkPrintoutOfMatrix() {
    assertEquals(
        "0.000      1.000      2.000      \n" +
        "1.000      2.000      3.000      \n" +
        "2.000      3.000      4.000      \n" +
        "3.000      4.000      5.000      \n",
        matrix4x3.toString());
}
```

The major downside of the current code base was the duplication, so he needed to incorporate every change on both places (on four because the unit tests were replicated too). Nonetheless, he has decided to make the code more readable by

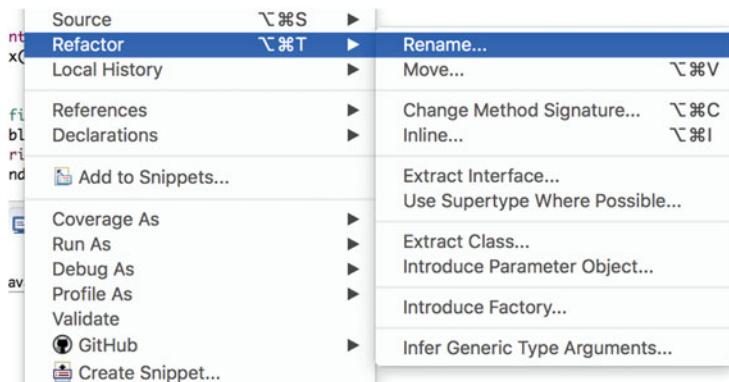


Fig. 5.5 The drop-down context menu for invoking the *Rename* refactoring. The refactoring is a safe action, as the IDE ensures that all changes are semantically sound

renaming the private fields and loop control variables. Here are the steps that he has taken using his Eclipse IDE¹¹ (the story is similar in other IDEs as well):

- He marked the private field member `n`, invoked the *Rename...* refactoring as shown in Fig. 5.5, and entered `columns`. He did the same for the field member `m`, but renamed it to `rows`. He has repeated the same steps with the input parameters of the constructor. Finally, he has deleted the unnecessary comments there, as the fields now properly reflect their meaning.
- He has first reassured himself that `i` everywhere denotes a row, while `j` denotes a column (the `addMatrix` method was fixed last time). This time, he used the *Find/Replace...* command from the *Edit* menu to reach the *Replace All* option, as depicted in Fig. 5.6. This global replacement may introduce undesired side effects, so it is mandatory that you recheck your code afterward. This is the reason why any alteration must be supported by high-quality automated tests.
- After executing the unit tests for both matrices, they have failed on the `checkPrintoutOfMatrix` test case. This was expected since the output from `toString` was reformed. There is a neat trick in Eclipse IDE that the developer has utilized. By double-clicking on the line signifying the failure reason inside the *Failure Trace* section, as shown in Fig. 5.7, Eclipse brings up the dialog box showing the difference between expected and actual outputs. The developer has just copy-pasted the content from the *Actual* part, and the previous unit test listing shows the outcome.

All in all, in three major steps and with the help of the Eclipse IDE, the developer has managed to accomplish the first task (see Exercise 11) as well as beautify the code

¹¹ At the time of this writing, I've used the Eclipse Oxygen Release (4.7.0).

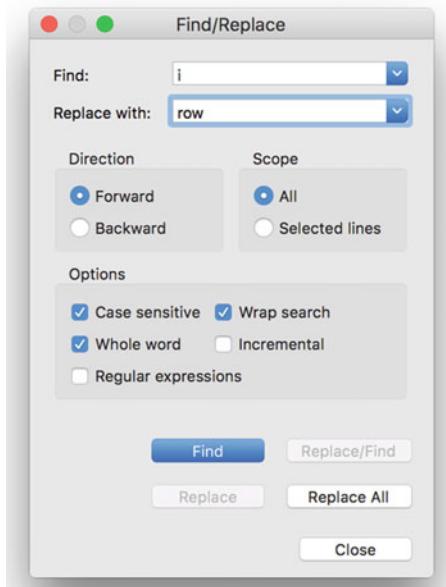


Fig. 5.6 The dialog box with all relevant fields filled in to replace all occurrences of *i* with *row*. A similar action was executed for the *j* variable

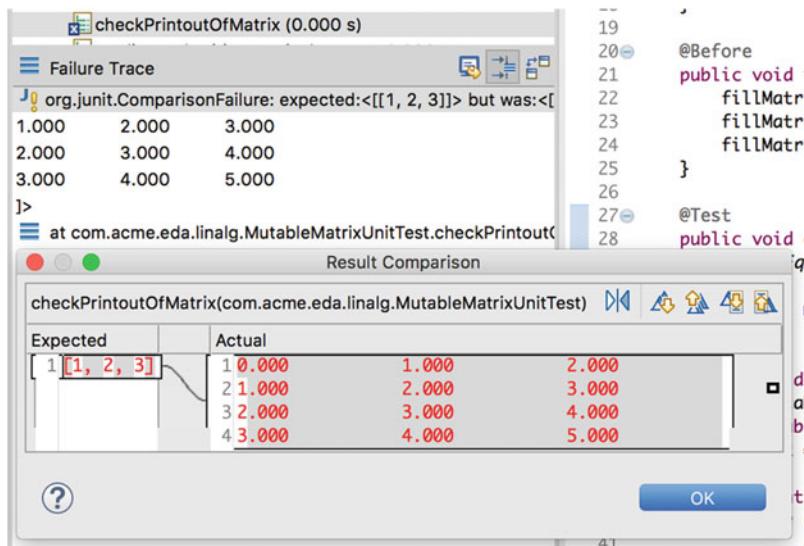


Fig. 5.7 The dialog box displaying the difference between expected and actual outputs. The actual content may be copied over into a unit test after ensuring that it is correct

base. Now, he has started the next one regarding the `equals` method. As a first measure, he created proper unit tests to expose the issue. These are shown next:

Listing of the Unit Tests to Exercise the Equals Method for the MutableMatrix

```
@Test
public void equalsShouldBeReflexive() {
    assertTrue(matrix4x3.equals(matrix4x3));
}

@Test
public void equalsShouldBeFalseWithNull() {
    assertFalse(matrix4x3.equals(null));
}

@Test
public void equalsShouldBeFalseWhenContentDiffers() {
    assertFalse(matrix4x3.equals(matrix5x3));
}

@Test
public void equalsShouldBeFalseWhenDimensionalityDiffers() {
    MutableMatrix matrix3x3 = new MutableMatrix(3, 3);
    MutableMatrix matrix1x9 = new MutableMatrix(1, 9);
    MutableMatrix matrix9x1 = new MutableMatrix(9, 1);

    assertFalse(matrix3x3.equals(matrix1x9));
    assertFalse(matrix3x3.equals(matrix9x1));
    assertFalse(matrix9x1.equals(matrix1x9));
}
```

The last test has failed, as the inherited `equals` method knew nothing about the dimensions of the matrix. At least, the issue has been fully reproduced, and everything was clear as to what must be done. Here is the implementation of the overridden `equals` method:

Listing of the New Equals Method in the MutableMatrix Class

```
@Override
public boolean equals(Object other) {
    if (this == other) {
        return true;
    } else if (other instanceof MutableMatrix) {
```

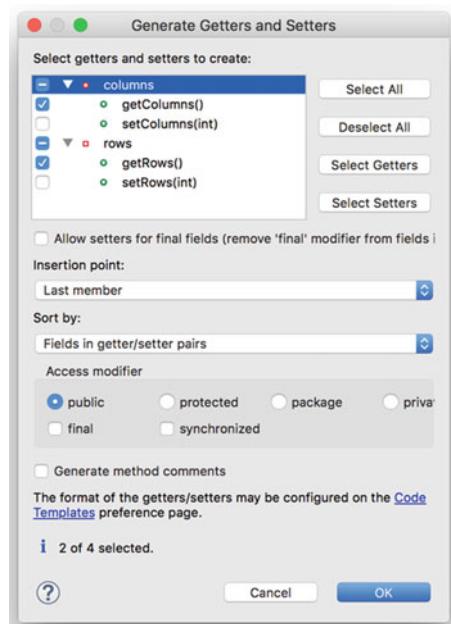
```

        MutableMatrix matrix = (MutableMatrix) other;
        return rows == matrix.rows &&
               columns == matrix.columns &&
               super.equals(matrix);
    }
    return false;
}

```

As all tests were passing now, the developer has concluded that the second task is completed too. Now, the only remaining duty was to put in those trivial dimensionality getters. The developer has again leveraged the Eclipse IDE to speed things up. He has marked the private fields of the class and selected from the *Source* drop-down menu the *Generate Getters and Setters...* item, as shown in Fig. 5.8. Here is the listing how these autogenerated methods look like (see the accompanying source code for unit tests):

Fig. 5.8 The IDE makes it easy to generate some mundane code, like getters and setters. Of course, you need to cover these autogenerated stuff with tests too. You never know when will you need to modify these methods by hand, and tests are excellent to catch typos



Listing of the Dimensionality Getters

```
public int getColumns() {  
    return columns;  
}  
  
public int getRows() {  
    return rows;  
}
```

After increasing the version number to 2.1.0,¹² the developer has declared his job as done.

5.17 Comparison Bug: Problem

In our private lives, we like surprises, as they bring something unusual and pleasant into foreground. On the contrary, software engineers hate them, as they never represent something enjoyable. A *surprise* is always an unexpectedly bad thing. This was the case here too. The developers of the EDA system had been struggling for a long time with a pesky bug. Eventually, they had managed to trace it back to the `equals` method. It turned out that `equals` disobeys the symmetry constraint.

5.18 Comparison Bug: Resolution

The JDK API documentation clearly states the following:

It is **symmetric**: for any non-null reference values x and y , $x.equals(y)$ should return true if and only if $y.equals(x)$ returns true.

Unfortunately, the unit tests haven't covered this aspect. This is a fine example that even if the code faithfully implements the tests (as kind of a specification in TDD), that doesn't imply completeness and correctness (see [8] for a more sobering illustration of the relationship between specification and implementation). So, the developer has created first a unit test to expose the problem, which is given next:

¹²The inclusion of those getters is indeed a new feature. On the other hand, the `toString` and `equals` methods are now behaving differently, so strictly speaking, this isn't a backward-compatible change. However, it would not warrant the brand-new 3.0.0 version.

Listing of the Unit Test to Verify the Symmetric Behavior of the Equals Method

```
@Test
public void equalsShouldBeSymmetric() {
    ArrayList<Double> content = new ArrayList<>(matrix4x3);
    assertTrue(content.equals(matrix4x3));
    assertEquals(matrix4x3.equals(content));
}
```

The bolded assertion has failed. The principal reason is that the two assertions call different versions of the `equals` method. The former uses the `equals` defined inside the `ArrayList` class, while the latter the one found inside the `MutableMatrix` class (the same story applies for the `ImmutableMatrix` class too). The `ArrayList`'s `equals` doesn't consider the dimensionality (see the previous phase). It turns out that there is no way to correctly implement the `equals` method in this case (see [9] for more details). Moreover, it is also interesting that you can populate an array with the matrix, as shown above. For an `ArrayList` the matrix appears as an ordinary list.

This is the first moment when the early decision to brake the LSP has materialized as an unsurmountable challenge (besides an ugly API of the matrix entities). In other words, an irresponsible act of the management to instigate rush work with that unfortunate inheritance simply wrecked evolution. At this moment, the developer had two conundrums: what to do with the test and what to say to the developers of the EDA system.

He had decided to simply inform the developers of the EDA software about the problem and proclaim that matrices must be targets of `equals`, i.e., they must receive the `equals` method calls. For the sake of completeness, he has raised the version number to 2.1.1, although nothing useful was delivered.

5.18.1 How to Manage a Failing Test Case

The developer was cognizant that he cannot simply check in a code with failing unit tests (don't forget that the same test has failed for both variants of matrices). This would blow up the continuous integration server's build job and raise a false alarm. The alarm would be unreal since the code cannot be reworked to pass the tests.

Many would just comment out the tests and possibly put a justification. I don't recommend this approach for the following reasons:

- The commented-out block of code sticks out as something terribly wrong and makes the source look unprofessional.
- Whatever is inside comments is never touched by the compiler and will rot over time. Nobody will ever take the effort to uncomment the code just to make it compile (let alone to keep it useful).

- There is no trace in test reports of such nonexistent code. It is easy to forget them.

There is a much better alternative. The JUnit framework and, for that matter, many similar testing frameworks allow you to mark a test as dormant. There is an annotation `@Ignore` for this purpose. In our case, the above test could have been annotated with:

```
@Ignore("Cannot correctly implement equals with broken LSP")
```

The ignored code will still compile with the rest of the tests, and the JUnit framework will report all such ignored tests, as shown in Fig. 5.9. Our developer has used this same style.

Of course, you shouldn't abuse the above annotation and mark every failing test with it. That would defeat its purpose. You must have strong arguments to ignore some test. For example, the Cucumber BDD framework has a notion of *Work In Progress* (WIP), and any scenario that is under development should be annotated with `@wip`. Cucumber also has an option to fail the tests if the number of WIPs exceed some threshold. Furthermore, Cucumber can also fail the test suite if any

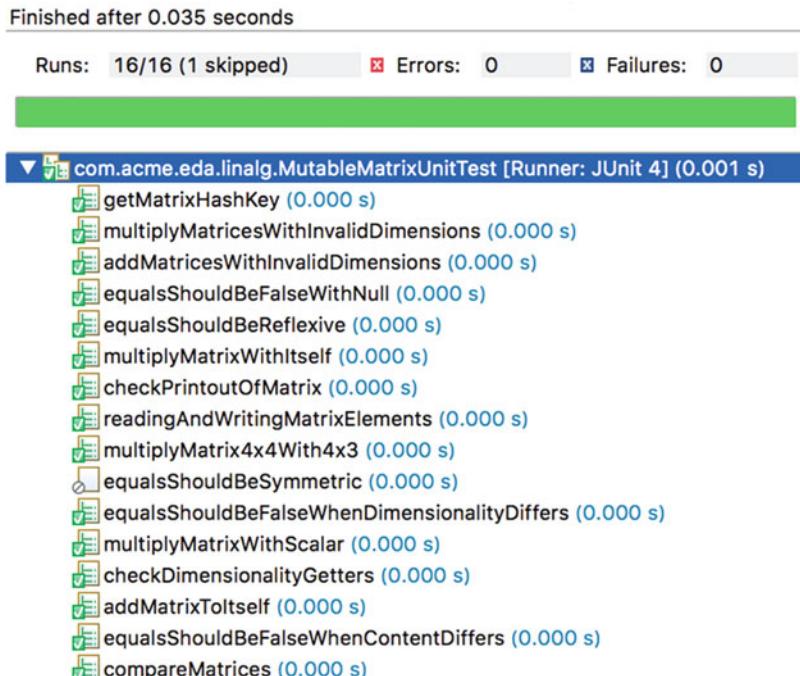


Fig. 5.9 The unit test report unambiguously showing the ignored `equalsShouldBeSymmetric` test case. It is surely harder to miss it now than if it's a dead code inside a comment

scenario marked as WIP turns green. This is an indication that you have probably forgotten to remove the `@wip` annotation from finished scenarios. As you may see, there is so much support when you keep your tests alive and annotate them properly. If all these would have been commented out, then no tool could help you to manage failing tests.

5.19 Sparse Matrix Extension: Problem

The EDA software had started to gain market share, and its user base had been steadily growing. Among them were also big companies, whose products had put a lot of stretch on the EDA system. Some issued a bug report that the EDA software cannot handle overly complex models. The software either reported an out of memory error or simply became sluggish. This was probably due to the large graphs associated with complex models, and those were represented by incidence matrices in the system. An incident matrix has as many rows and columns as there are vertices and edges in the graph, respectively. If vertex X is incident upon edge Y , then the matrix has 1 at row X and column Y . Otherwise, it has 0.

Further analysis had revealed that incidence matrices were mostly filled with zeros. This excessive storage of zeros was causing problems. The company had decided to stop using dense matrices to represent models. They had requested the implementation of the sparse matrix type. The task was assigned to an experienced developer.

5.20 Sparse Matrix Extension: Resolution

The developer has noticed that the system already possesses two matrix types: mutable dense and immutable dense. Consequently, he has decided to introduce two additional ones: mutable sparse (`MutableSparseMatrix`) and immutable sparse (`ImmutableSparseMatrix`). He didn't want to change the existing matrices by inserting the word `Dense` into their name.

Afterward, scrutinizing the existing matrices has concluded that they are comprised of methods grouped into two sets: those that manipulate elements of the matrix and those that perform arithmetic operations. The latter just use the element handlers. Therefore, he just needed to find the suitable data structure to store elements and redefine the element accessors. He has figured out that a hash table would be perfect here. The key would be the already existing element index calculated upon the (row, column) coordinates. For the mutable variant, he could use `Hashtable`, and for the immutable version, he may extend from `HashMap`. Moreover, he was happy seeing that he may completely reuse the unit tests for the other matrix types (see Exercise 12). Here is the complete implementation of the `MutableSparseMatrix` class (see the accompanying source code for the tests):

Listing of the MutableSparseMatrix Class

```
/**  
 * Class to work with mutable sparse matrices.  
 */  
@SuppressWarnings("serial")  
public class MutableSparseMatrix  
    extends HashMap<Integer,Double> {  
    private int columns;  
    private int rows;  
  
    public MutableSparseMatrix(int rows, int columns) {  
        super(rows * columns);  
        this.rows = rows;  
        this.columns = columns;  
    }  
  
    private int elementIndex(int row, int column) {  
        return row * columns + column;  
    }  
  
    public double getElement(int row, int column) {  
        Double value = get(elementIndex(row, column));  
        return value != null ? value : 0.0;  
    }  
  
    // Returns back the previous value.  
    public double setElement(  
        int row, int column, double newValue) {  
        Double oldValue =  
            put(elementIndex(row, column), newValue);  
        return oldValue != null ? oldValue : 0.0;  
    }  
  
    // Adds the other matrix to this one.  
    public void addMatrix(MutableSparseMatrix other) {  
        if (other.columns != columns || other.rows != rows) {  
            throw new IllegalArgumentException(  
                "The dimensions of matrices don't match");  
        }  
  
        for (int row = 0; row < rows; row++) {  
            for (int column = 0; column < columns; column++) {  
                setElement(  
                    row, column,
```

```
        getElement(row, column) +
        other.getElement(row, column));
    }
}

// Returns back the product of this matrix
// and the other one.
public MutableSparseMatrix multiplyMatrix(
    MutableSparseMatrix other) {
    if (other.rows != columns) {
        throw new IllegalArgumentException(
            "The dimensions of matrices don't match");
    }

    MutableSparseMatrix newMatrix =
        new MutableSparseMatrix(rows, other.columns);
    for (int row = 0; row < rows; row++) {
        for (int column = 0; column < other.columns; column++) {
            for (int k = 0; k < columns; k++) {
                newMatrix.setElement(
                    row, column,
                    newMatrix.getElement(row, column) +
                    getElement(row, k) *
                    other.getElement(k, column));
            }
        }
    }
    return newMatrix;
}

// Multiplies this matrix with the scalar value.
public void multiplyWithScalar(double value) {
    for (int row = 0; row < rows; row++)
        for (int column = 0; column < columns; column++) {
            setElement(
                row, column,
                getElement(row, column) * value);
        }
}

@Override
public String toString() {
    StringBuilder buff = new StringBuilder();
    for (int row = 0; row < rows; row++) {
```

```
        for (int column = 0; column < columns; column++) {
            buff.append(String.format(
                "%-15.3f", getElement(row, column)));
            if (column != columns - 1) {
                buff.append(' ');
            }
        }
        buff.append('\n');
    }
    return buff.toString();
}

@Override
public boolean equals(Object other) {
    if (this == other) {
        return true;
    } else if (other instanceof MutableSparseMatrix) {
        MutableSparseMatrix matrix = (MutableSparseMatrix)
other;
        return rows == matrix.rows &&
               columns == matrix.columns &&
               super.equals(matrix);
    }
    return false;
}

public int getColumns() {
    return columns;
}

public int getRows() {
    return rows;
}
}
```

The bolded code sections are those that differ from the `MutableMatrix` class, which is irritating. Notice that the initialization with zeros has disappeared from the constructor. However, the management has emphasized that restructuring of the code base is already part of the plan. Now, the new release of the EDA system must be quickly delivered; otherwise, the company will lose the most precious users. The developer has increased the version number to 2.2.0 and made a release of the linear algebra library. The EDA system was also shipped soon after.

5.21 Sparsity Bug: Problem

The users had been complaining again that the initial problem of running out of memory wasn't properly solved. Namely, the software would run fast at the beginning but over time, it would slow down and eventually abort with an out of memory error. An additional headache for the ACME Corporation was a new competitor on the market with their cool and sophisticated EDA system. The competitor's product had offered several features completely missing in the ACME EDA (like circuit fault analysis, advanced simulations, etc.). All in all, the management had again asked for an immediate fix.

5.22 Sparsity Bug: Resolution

Sure enough, it was obvious that something wrong is going on with the functioning of the sparse matrices. Initially, they are running properly and over time, their performance drops. They behave after a while like dense matrices. The developer who was assigned to this task has devised a good tactic to catch the root cause. He has implemented the following test case:

Listing to Test Whether the Sparse Matrix Will Remain Sparse over Time

```
@Test
public void ensureThatTheSparseMatrixRemainsSparse() {
    MutableSparseMatrix matrix4x4 =
        new MutableSparseMatrix(4, 4);
    for (int i = 0; i < 4; i++) {
        for (int j = 0; j < 4; j++) {
            matrix4x4.setElement(i, j, 1.0);
        }
    }
    assertEquals(16, matrix4x4.size());
    for (int i = 0; i < 4; i++) {
        for (int j = 0; j < 4; j++) {
            matrix4x4.setElement(i, j, 0.0);
        }
    }
    assertEquals(0, matrix4x4.size());
}
```

This test has failed since after setting the matrix's elements to zero, its size hasn't dropped back to zero. Once the bug was reproduced, the fix was easy. Here is the modified `setElement` method:

Listing of the Modified setElement Method in the MutableSparseMatrix Class

```

public double setElement(
    int row, int column, double newValue) {
    Double oldValue;

    if (Math.abs(newValue) < EPSILON) {
        oldValue = remove(elementIndex(row, column));
    } else {
        oldValue = put(elementIndex(row, column), newValue);
    }
    return oldValue != null ? oldValue : 0.0;
}

```

The `setElement` method first checks whether the input parameter is inside the $[0.0 - \epsilon, 0.0 + \epsilon]$ range or not. The constant ϵ is some tolerance, although our sparse matrix would only store 1.0 or 0.0. The developer has released a new 2.2.1 version of the linear algebra library.

5.23 Maintainability Issues: Problem

The EDA software was finally stable but the crisis had not gone away. The competitor was fierce and aggressive in its campaigns and had been promoting some advanced features missing from the ACME EDA. The ACME corporation had determined to address this issue by boosting the capabilities of its EDA system. Unfortunately, at this moment further evolution was impossible without a general revision. The developers of both the ACME EDA as well as the linear algebra library were stuck with an unmaintainable code base. The inconsiderate duplication and rushed work had resulted in a class diagram depicted in Fig. 5.10.

The clients of the linear algebra library held references to four different matrix classes that behaved like totally isolated entities. Any new feature would need to be replicated on eight places (into four implementation classes supported by four identical test cases). The management had given a green light for developers to restructure the system just enough to enable further progress.

5.24 Maintainability Issues: Resolution

The joint agreement and work plan among developers was an accomplishment of the following goals:

- A matrix entity must be represented by an interface describing its API.
- The clients of the linear algebra library must not know anything about concrete implementation classes.

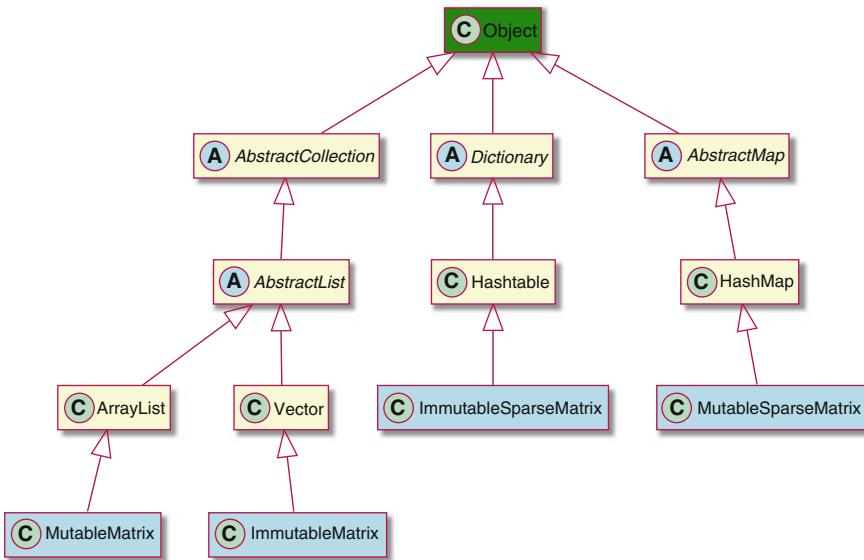


Fig. 5.10 The common ancestor of various matrix types is the `Object` class, which is anyhow the root of all classes in Java. Each leaf class had its own separate but nearly identical unit test class. This situation was unbearable

- There should be a single entry point for retrieving matrix instances. The clients should be able to specify the desired capabilities of the returned object.
- The clients should have a facility to convert their matrix type into another type (e.g., create a mutable version from an immutable one).
- The clients must be able to define the content type of matrices (currently, it is `double`).
- The published API must be thoroughly documented.

All sorts of duplication and repetitive work must be avoided, i.e., the code base should revolve around the DRY (*Don't Repeat Yourself*) Agile principle.

All but the last item is about putting clients of the linear algebra library in a better position. The last item is mostly relevant for the developers of the library itself, as clients would be shielded from that mess. Now, there are two general ways to tackle this compound job:

- Using an API-centric method
- Performing the job in a sequential all-or-nothing fashion

The difference is profound as shown in Figs. 5.11 and 5.12. With the API-centric method, you can parallelize work and balance technical debt without incurring an interest rate. It is also beneficial from the viewpoint of communication, i.e., it

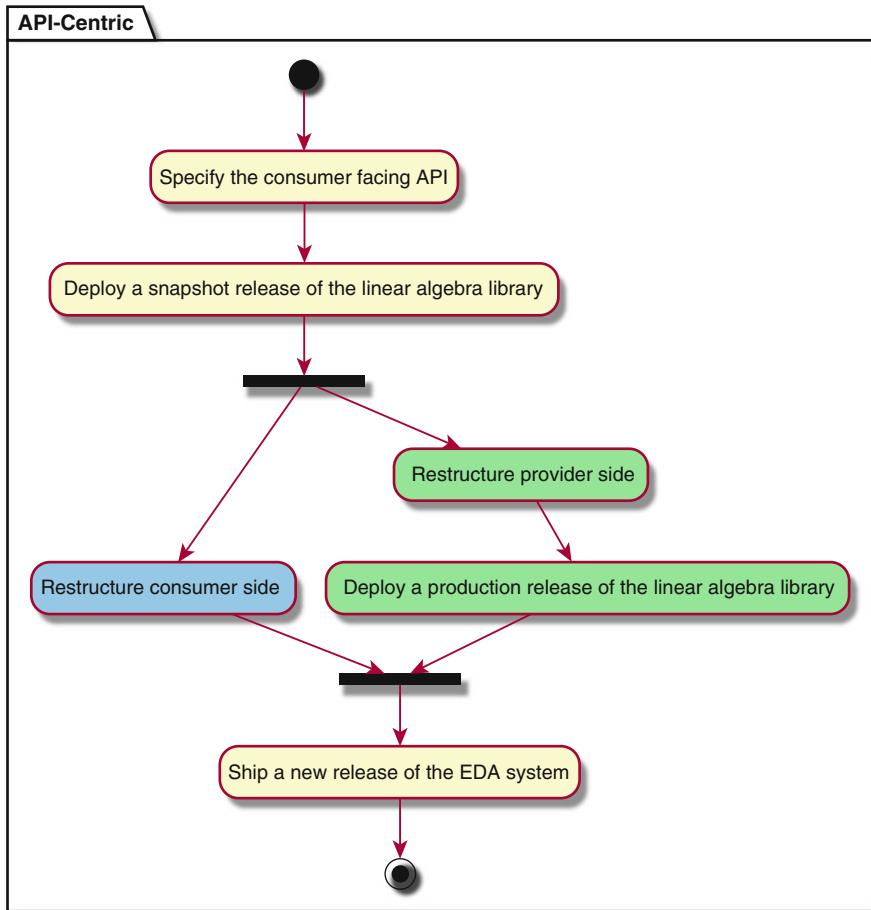


Fig. 5.11 The API-centric method has an important parallelized section containing the longest activities on both sides. The feedback loop is alive, while the consumer (client) is restructuring its code. The initial waiting time is only for acquiring the snapshot version of the library. The client may easily generate test doubles based upon the published API

maintains a more active feedback loop. The client could provide judgment regarding the API's usability while things are still in flux.

In the rest of this section, we will showcase the three major activities that have happened pertaining to the provider side of the linear algebra library. The first listing shows the common API for all matrix types. The second listing presents the class that implements the *Factory Method* design pattern. This is an entity that serves as an entry point for getting various instances of matrices based upon some characteristics. For now, the developers of the library have decided to accept key/value pairs as input. For example, to request a mutable matrix, the client would provide the (mutable, true) pair encoded as a string:

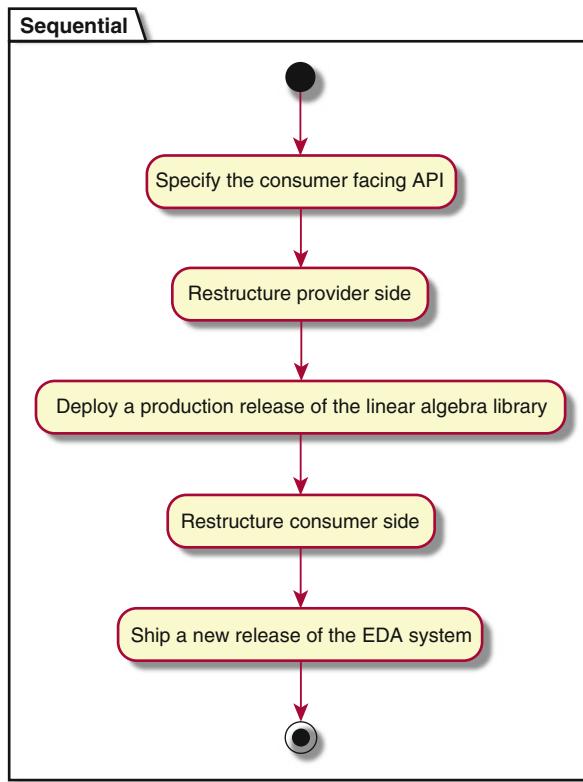


Fig. 5.12 In this method, all activities are linearly ordered. The client must wait until everything is finished by the provider. The moment to send feedback is postponed, and fixing errors is more expensive than in the API-centric case

Listing of the Client API that Is Now Shared by All Matrix Types

```

package com.example.eda.linalg;

import java.util.Properties;

/**
 * The client API of all matrix types (mutable, immutable,
 * dense, sparse, etc.). The standard operations inherited
 * from the {@link Object} class of converting a matrix to
 * string, comparing matrices and computing the hash code are
 * customized for each matrix type, but are not enrolled
 * here.
 *
 * @author ACME Developer

```

```
* @since 3.0
* @see Object#equals(Object)
* @see Object#hashCode()
* @see Object#toString()
* @param <E> the content type of this matrix
*         (for example, Double, Integer, etc.).
*/
public interface Matrix<E extends Number> {
    /**
     * Gets the number of rows.
     *
     * @return the maximum number of rows of this matrix.
     */
    int getRows();

    /**
     * Gets the number of columns.
     *
     * @return the maximum number of columns of this matrix.
     */
    int getColumns();

    /**
     * Gets the characteristics of this matrix. By default
     * the following properties are supported (the content
     * below isn't case sensitive):
     * <ul>
     * <li>mutable : true or false</li>
     * <li>matrix.type : dense or sparse</li>
     * <li>element.type : double, float, long, integer,
     *                     short or byte</li>
     * </ul>
     *
     * @return the characteristics of this matrix.
     */
    Properties getCharacteristics();

    /**
     * Retrieves the element at given coordinates.
     *
     * @return the value of the specified element.
     * @param row the row number of the element in the range
     *            of [0, max. rows - 1].
     * @param column the column number of the element in the
     *              range of [0, max. columns - 1].
     */
```

```
* @throws IllegalArgumentException if the coordinates
*         are out of range.
*/
E getElement(int row, int column);

/**
 * Sets the element at given coordinates.
 *
 * @return the reference to the updated matrix. It can
 *         be this matrix if it is mutable, or a new
 *         instance in case of an immutable one.
 * @param row the row number of the element in the range
 *            of [0, max. rows - 1].
 * @param column the column number of the element in the
 *               range of [0, max. columns - 1].
 * @param value the new value of the specified element.
 * @throws IllegalArgumentException if the coordinates
 *         are out of range.
 * @throws NullPointerException if the input reference
 *         is {@code null}.
*/
Matrix<E> setElement(int row, int column, E value);

/**
 * Performs matrix addition.
 *
 * @return the reference to the updated matrix. It can
 *         be this matrix if it is mutable, or a new
 *         instance in case of an immutable one.
 * @param other the matrix that needs to be added to
 *             this matrix.
 * @throws NullPointerException if the input reference
 *             is {@code null}.
 * @throws IllegalArgumentException if the dimensionality
 *             of matrices don't match.
*/
Matrix<E> add(Matrix<E> other);

/**
 * Performs matrix multiplication.
 *
 * @return the reference to the new matrix. As it is
 *         cumbersome to do in place multiplication,
 *         this operation never mutates the source
 *         matrix.
```

```
* @param other the matrix with which to multiply
*         this matrix.
* @throws NullPointerException if the input reference
*         is {@code null}.
* @throws IllegalArgumentException if the dimensionality
*         of matrices don't match.
*/
Matrix<E> multiply(Matrix<E> other);

/**
 * Performs matrix multiplication with a scalar.
 *
 * @return the reference to the updated matrix. It can
 *         be this matrix if it is mutable, or a new
 *         instance in case of an immutable one.
 * @param scalar the value with which to multiply all
 *         elements of this matrix.
 * @throws NullPointerException if the input reference
 *         is {@code null}.
*/
Matrix<E> multiply(E scalar);
}
```

The `Matrix` interface documents all operations of the client API. There is no need to look anywhere else to understand how to use matrices. The bolded sections deserve some additional comments, as listed below:

- The `@since` directive should only contain the major and minor versions. Don't put here the patch version, as it should never impact the API.
- The `equals`, `hashCode`, and `toString` methods are part of the client API that is automatically inherited by all classes in Java. The documentation just states that they are customized for matrices. If you put too much information about how these are implemented, then you can easily leak out implementation details. Whatever you publicly document is part of an API irrespectively of whether it is inside the Javadoc or code.
- The references via `@see` are convenient to draw users' attention to other related entities (in this case, customized methods from the `Object` class). This cross-linking is also useful to avoid repetition.
- The template (generics) parameter must be properly documented, as any other input parameter.
- The `getCharacteristics` Javadoc comment contains embedded HTML. Keep these stuff at minimum. HTML doesn't look nice with regular comments and Java code. If you need to, then you may mark some special text with a Javadoc tag, like `@code`.

Listing of the Factory for Creating Matrices

```
package com.example.eda.linalg;

import java.util.Objects;
import java.util.Properties;

/**
 * The single entry point for clients to ask for matrix
 * instances based upon the characteristics. This class
 * also provides an ability to convert between matrix types.
 *
 * @author ACME Developer
 * @since 3.0
 */
public final class MatrixFactory {
    /**
     * Creates a new instance with the desired
     * characteristics and dimensions.
     *
     * @return a new matrix instance.
     * @param rows the row dimension of the matrix.
     * @param columns the column dimension of the matrix.
     * @param characteristics the specification of the
     * target matrix type.
     * @param <E> the content type of this matrix
     * (for example, Double, Integer, etc.).
     * @throws NullPointerException if the input reference is
     * {@code null}.
     * @throws IllegalArgumentException if any dimension is
     * negative.
     */
    @SuppressWarnings("unchecked")
    public static <E extends Number> Matrix<E> newInstance(
        int rows, int columns,
        Properties characteristics) {
        Objects.requireNonNull(characteristics);
        if (rows < 0 || columns < 0) {
            throw new IllegalArgumentException(
                "The dimensions are negative.");
        }

        // TODO: see Exercise 13
        return (Matrix<E>) new MutableDenseRealMatrix(
```

```
        rows, columns, characteristics);
    }

    /**
     * Creates a new instance with the desired
     * characteristics from the source matrix.
     *
     * @return a new matrix instance.
     * @param source the source of the conversion.
     * @param characteristics the specification of the
     * target matrix type.
     * @param <E> the content type of this matrix
     * (for example, Double, Integer, etc.).
     * @throws NullPointerException if any input reference is
     * {@code null}.
     */
    public static <E extends Number> Matrix<E> newInstance(
        Matrix<E> source, Properties characteristics) {
        Objects.requireNonNull(source);

        @SuppressWarnings("unchecked")
        AbstractMatrix<E> target =
            (AbstractMatrix<E>) newInstance(
                source.getRows(), source.getColumns(),
                characteristics);
        for (int row = 0; row < target.getRows(); row++) {
            for (int column = 0;
                column < target.getColumns();
                column++) {
                target.set(
                    row, column,
                    source.getElement(row, column));
            }
        }
        return target;
    }
}
```

The previous interface and this factory class are all what the client sees from the linear algebra library. The second `newInstance` method is used to convert one matrix type into another. It relies on the first `newInstance` method to produce a brand-new instance with the given properties. That one isn't fully implemented, as noted in the comment. The `Objects` utility class contains very useful methods; among them is the one used above to check for null.

The advice to prefer factory methods over constructors isn't that simple and cannot be eagerly reduced as a syntactic sugar. A factory method has two major purposes: to give you control over how instances are created and to increase readability. The latter is especially important from the perspective of the client. In our case, we are using overloaded variants of the `newInstance` method. In both methods, the first parameters define the initial content of the new matrix (either blank or filled with elements from the source matrix). The second parameter is always the characteristics of the new matrix. All in all, it is intuitive what is going on. On the other hand, analyze the following contrived example of the `Circle` class:

```
class Circle {
    private Circle(Point center, Double radius)...
    public static getCircle(Point, Double)
    public static getCircle(Point)
    public static getCircle(Double)
}
```

It is hard to discern what is the difference between `getCircle(3.1)` and `getCircle(p2)`. If you want to increase readability by introducing a factory method pattern, then you should name each factory method separately when the context isn't clear. Compare the following refactored `Circle` class:

```
class Circle {
    private Circle(Point center, Double radius)...
    public static getCircle(Point, Double)
    public static getFixedSizeCircle(Point)
    public static getCenterCircle(Double)
}
```

Next is the integration test suite that the developer of the library has handed over to the client. It could serve as an excellent addendum to the Javadoc generated API documentation:

Listing of the Integration Test Suite for the Client API

```
public final class ClientMatrixAPIIntegrationTest {
    private static final Properties CHARACTERISTICS =
        new Properties();

    private final Matrix<Double> matrix4x4 =
        MatrixFactory.newInstance(4, 4, CHARACTERISTICS);
    private final Matrix<Double> matrix4x3 =
        MatrixFactory.newInstance(4, 3, CHARACTERISTICS);
```

```
private final Matrix<Double> matrix5x3 =
    MatrixFactory.newInstance(5, 3, CHARACTERISTICS);

private void fillMatrix(Matrix<Double> matrix,
    int rows, int columns) {
    for (int i = 0; i < rows; i++)
        for (int j = 0; j < columns; j++) {
            matrix.setElement(i, j, (double) (i + j));
        }
}

@BeforeClass
public static void setupCharacteristics() {
    CHARACTERISTICS.setProperty("mutable", "true");
    CHARACTERISTICS.setProperty("matrix.type", "dense");
    CHARACTERISTICS.setProperty("element.type", "double");
}

@Before
public void initializeMatrices() {
    fillMatrix(matrix4x4, 4, 4);
    fillMatrix(matrix4x3, 4, 3);
    fillMatrix(matrix5x3, 5, 3);
}

@Test
public void checkPrintoutOfMatrix() {
    assertEquals(
        "0.000      1.000      2.000      \n" +
        "1.000      2.000      3.000      \n" +
        "2.000      3.000      4.000      \n" +
        "3.000      4.000      5.000      \n",
        matrix4x3.toString());
}

private void validateContent(
    Matrix<Double> matrix, int rows, int columns,
    double tolerance, double... elements) {
    int idx = 0;

    for (int i = 0; i < rows; i++)
        for (int j = 0; j < columns; j++) {
            assertEquals(elements[idx++],
                matrix.getElement(i, j), tolerance);
        }
}
```

```
        }
```

```
}
```

```
@Test
```

```
public void multiplyMatrixWithScalar() {
```

```
    matrix4x3.multiply(10.0);
```

```
    validateContent(
```

```
        matrix4x3, 4, 3, 0.0,
```

```
        0.0, 10.0, 20.0,
```

```
        10.0, 20.0, 30.0,
```

```
        20.0, 30.0, 40.0,
```

```
        30.0, 40.0, 50.0);
```

```
}
```

```
@Test
```

```
public void multiplyMatrixWithItself() {
```

```
    final Matrix<Double> result =
```

```
        matrix4x4.multiply(matrix4x4);
```

```
    assertNotNull(result);
```

```
    validateContent(
```

```
        result, 4, 4, 0.0,
```

```
        14.0, 20.0, 26.0, 32.0,
```

```
        20.0, 30.0, 40.0, 50.0,
```

```
        26.0, 40.0, 54.0, 68.0,
```

```
        32.0, 50.0, 68.0, 86.0);
```

```
}
```

```
@Test
```

```
public void multiplyMatrix4x4With4x3() {
```

```
    final Matrix<Double> result =
```

```
        matrix4x4.multiply(matrix4x3);
```

```
    assertNotNull(result);
```

```
    validateContent(
```

```
        result, 4, 3, 0.0,
```

```
        14.0, 20.0, 26.0, 20.0,
```

```
        30.0, 40.0, 26.0, 40.0,
```

```
        54.0, 32.0, 50.0, 68.0);
```

```
}
```

```
@Test
```

```
public void addMatrixToItself() {
```

```
    matrix4x4.add(matrix4x4);
```

```
    validateContent(
```

```
        matrix4x4, 4, 4, 0.0,
```

```
        0.0, 2.0, 4.0, 6.0,
```

```
        2.0, 4.0, 6.0, 8.0,
        4.0, 6.0, 8.0, 10.0,
        6.0, 8.0, 10.0, 12.0);
    }

@Test
public void compareMatrices() {
    assertTrue(matrix5x3.equals(matrix5x3));
    assertFalse(matrix4x3.equals(matrix5x3));
}

@Test
public void getMatrixHashKey() {
    assertEquals(1401420256, matrix4x4.hashCode());
}

@Test
public void readingAndWritingMatrixElements() {
    assertEquals(4.0, matrix4x4.getElement(2, 2), 0.0);
    matrix4x4.setElement(2, 2, 111.0);
    assertEquals(111.0, matrix4x4.getElement(2, 2), 0.0);
}

@Test(expected = IllegalArgumentException.class)
public void addMatricesWithInvalidDimensions() {
    matrix4x4.add(matrix4x3);
}

@Test(expected = IllegalArgumentException.class)
public void multiplyMatricesWithInvalidDimensions() {
    matrix4x3.multiply(matrix5x3);
}

@Test
public void equalsShouldBeReflexive() {
    assertEquals(matrix4x3, matrix4x3);
}

@Test
public void equalsShouldBeFalseWithNull() {
    assertFalse(matrix4x3.equals(null));
}

@Test
public void equalsShouldBeFalseWhenContentDiffers() {
```

```
        assertFalse(matrix4x3.equals(matrix5x3));
    }

    @Test
    public void
        equalsShouldBeFalseWhenDimensionalityDiffers() {
        Matrix<Double> matrix3x3 =
            MatrixFactory.newInstance(3, 3, CHARACTERISTICS);
        Matrix<Double> matrix1x9 =
            MatrixFactory.newInstance(1, 9, CHARACTERISTICS);
        Matrix<Double> matrix9x1 =
            MatrixFactory.newInstance(9, 1, CHARACTERISTICS);

        assertFalse(matrix3x3.equals(matrix1x9));
        assertFalse(matrix3x3.equals(matrix9x1));
        assertFalse(matrix9x1.equals(matrix1x9));
    }

    @Test
    public void checkDimensionalityGetters() {
        assertEquals(4, matrix4x3.getRows());
        assertEquals(3, matrix4x3.getColumns());
    }

    @Test
    public void checkCharacteristics() {
        assertEquals(
            CHARACTERISTICS, matrix4x4.getCharacteristics());
    }

    @Test
    public void convertFromSource() {
        Matrix<Double> clone4x3 = MatrixFactory.newInstance(
            matrix4x3, CHARACTERISTICS);
        assertEquals(matrix4x3, clone4x3);
        assertEquals(
            CHARACTERISTICS, clone4x3.getCharacteristics());
    }

    @Test(expected = NullPointerException.class)
    public void newInstanceWithNullCharacteristics() {
        MatrixFactory.newInstance(3, 3, null);
    }
```

```
@Test(expected = IllegalArgumentException.class)
public void newInstanceWithNegativeRows() {
    MatrixFactory.newInstance(-3, 3, CHARACTERISTICS);
}

@Test(expected = IllegalArgumentException.class)
public void newInstanceWithNegativeColumnss() {
    MatrixFactory.newInstance(3, -3, CHARACTERISTICS);
}

@Test(expected = NullPointerException.class)
public void convertInstanceWithNullCharacteristics() {
    MatrixFactory.newInstance(matrix4x4, null);
}

@Test(expected = NullPointerException.class)
public void convertInstanceWithNullSource() {
    MatrixFactory.newInstance(null, CHARACTERISTICS);
}
}
```

The test suite is utilizing the linear algebra library only through the API without using any concrete classes. This is how the client side will look like once they upgrade to version 3.0.0. Notice that arithmetic operations accept a matrix represented solely through the interface. Instance creations are hidden behind the factory class. Below are the listings of the restructured classes on the provider side (see Exercises 13 and 14). None of these are accessible by the client, as they are all package visible:

Listing of the AbstractMatrix Base Class

```
/**
 * Base class that implements common functionality present
 * in all matrix types.
 *
 * @author ACME Developer
 * @since 3.0
 */
abstract class AbstractMatrix<E extends Number>
    implements Matrix<E>, Calculate<E> {
    private final int columns;
    private final int rows;
    private final Properties characteristics =
        new Properties();
```

```
AbstractMatrix(int rows, int columns,
              Properties characteristics) {
    assert characteristics != null :
        "Characteristics is missing";

    this.rows = rows;
    this.columns = columns;
    this.characteristics.putAll(characteristics);
}

/**
 * Checks the boundary conditions for element
 * coordinates.
 *
 * @param row the row number of the element in the range
 *            of [0, max. rows - 1].
 * @param column the column number of the element in the
 *            range of [0, max. columns - 1].
 * @throws IllegalArgumentException if the coordinates
 *            are out of range.
 */
protected void checkCoordinates(int row, int column) {
    if ((row < 0 || row >= rows) ||
        (column < 0 || column >= columns)) {
        throw new IllegalArgumentException(
            "The coordinates are out of range.");
    }
}

/**
 * Calculates the index of the element, as if the matrix
 * would be an array. This index may also be used as
 * a unique identifier in other data structures.
 *
 * @param row the row number of the element in the range
 *            of [0, max. rows - 1].
 * @param column the column number of the element in the
 *            range of [0, max. columns - 1].
 * @return the unique index of this element.
 */
protected int elementIndex(int row, int column) {
    assert 0 <= row && row < rows :
        "Row is out of range";
    assert 0 <= column && column < columns :
        "Column is out of range";
```

```
        return row * columns + column;
    }

    /**
     * Implements the raw set element method.
     * Each matrix type may use a different data structure to
     * store elements (array, map, etc.). For this reason,
     * this template method must be overridden by a concrete
     * matrix class.
     *
     * @param row the row number of the element in the range
     *            of [0, max. rows - 1].
     * @param column the column number of the element in the
     *            range of [0, max. columns - 1].
     * @param value the new value for the element.
     */
    protected abstract void set(
        int row, int column, E value);

    /**
     * Returns the reference to the target matrix.
     *
     * @return the reference to the target matrix. It can
     *         be this matrix if it is mutable, or a new
     *         instance in case of an immutable one. The
     *         dimensions and characteristics of the returned
     *         matrix will be the same as of the caller.
     */
    protected abstract AbstractMatrix<E> targetMatrix();

    /**
     * Returns a new matrix instance with the same
     * characteristics as of the caller.
     *
     * @return a new instance with the given dimensions.
     * @param rows the maximum number of rows of the matrix.
     * @param columns the maximum number of columns of the
     *            matrix.
     */
    protected abstract AbstractMatrix<E> newMatrix(
        int rows, int columns);

    @Override
    public Matrix<E> add(Matrix<E> other) {
        Objects.requireNonNull(other);
    }
}
```

```
if (other.getColumns() != columns ||  
    other.getRows() != rows) {  
    throw new IllegalArgumentException(  
        "The dimensions of matrices don't match");  
}  
  
final AbstractMatrix<E> result = targetMatrix();  
for (int row = 0; row < rows; row++) {  
    for (int column = 0; column < columns;  
        column++) {  
        result.set(  
            row, column,  
            add(  
                getElement(row, column),  
                other.getElement(row, column)));  
    }  
}  
return result;  
}  
  
@Override  
public Matrix<E> multiply(Matrix<E> other) {  
    Objects.requireNonNull(other);  
    if (other.getRows() != columns) {  
        throw new IllegalArgumentException(  
            "The dimensions of matrices don't match");  
    }  
  
final AbstractMatrix<E> result =  
    newMatrix(rows, other.getColumns());  
for (int row = 0; row < rows; row++) {  
    for (int column = 0; column < other.getColumns();  
        column++) {  
        for (int k = 0; k < columns; k++) {  
            result.set(  
                row, column,  
                add(  
                    result.getElement(row, column),  
                    multiply(  
                        getElement(row, k),  
                        other.getElement(k, column))));  
        }  
    }  
}
```

```
        return result;
    }

@Override
public Matrix<E> multiply(E scalar) {
    Objects.requireNonNull(scalar);

    final AbstractMatrix<E> result = targetMatrix();
    for (int row = 0; row < rows; row++) {
        for (int column = 0; column < columns;
             column++) {
            result.set(
                row, column,
                multiply(getElement(row, column), scalar));
        }
    }
    return result;
}

@Override
public String toString() {
    StringBuilder buff = new StringBuilder();
    for (int row = 0; row < rows; row++) {
        for (int column = 0; column < columns;
             column++) {
            buff.append(String.format(
                "%-15.3f", getElement(row, column)));
            if (column != columns - 1) {
                buff.append(' ');
            }
        }
        buff.append('\n');
    }
    return buff.toString();
}

// A very inefficient implementation of equals.
// Children classes may want to override this,
// and provide a much faster variant.
@Override
public boolean equals(Object other) {
    if (this == other) {
        return true;
    } else if (other instanceof Matrix) {
        @SuppressWarnings("rawtypes")

```

```
final Matrix otherMatrix = (Matrix) other;
boolean isEqual =
    rows == otherMatrix.getRows() &&
    columns == otherMatrix.getColumns();
for (int row = 0; isEqual && (row < rows);
     row++) {
    for (int column = 0;
         isEqual && (column < columns);
         column++) {
        isEqual =
            getElement(row, column)
            .equals(
                otherMatrix.getElement(
                    row, column));
    }
}
return isEqual;
}

@Override
public int hashCode() {
    return Objects.hashCode(new Object[] {this});
}

@Override
public int getColumns() {
    return columns;
}

@Override
public int getRows() {
    return rows;
}

@Override
public Properties getCharacteristics() {
    return characteristics;
}
}
```

Listing of the Calculate Interface

```
interface Calculate<E extends Number> {
    E add(E x, E y);
    E multiply(E x, E y);
}
```

Listing of the RealAbstractMatrix Class

```
/**
 * Implements the basic arithmetic operations for real
 * numbers.
 *
 * @author ACME Developer
 * @since 3.0
 */
abstract class RealAbstractMatrix
    extends AbstractMatrix<Double> {
    RealAbstractMatrix(int rows, int columns,
                      Properties characteristics) {
        super(rows, columns, characteristics);
    }

    @Override
    public Double add(Double x, Double y) {
        assert x != null && y != null;
        return x + y;
    }

    @Override
    public Double multiply(Double x, Double y) {
        assert x != null && y != null;
        return x * y;
    }
}
```

Listing of the DenseRealAbstractMatrix Class

```
/**
 * Class that implements a dense matrix with real numbers
 * as elements.
 *
 * @author ACME Developer
 * @since 3.0
 */
```

```

abstract class DenseRealAbstractMatrix
    extends RealAbstractMatrix {
    private final double[] store;

    DenseRealAbstractMatrix(int rows, int columns,
                           Properties characteristics) {
        super(rows, columns, characteristics);
        store = new double[rows * columns];
    }

    @Override
    public Double getElement(int row, int column) {
        checkCoordinates(row, column);
        return store[elementIndex(row, column)];
    }

    @Override
    protected void set(int row, int column, Double value) {
        assert value != null;
        store[elementIndex(row, column)] = value;
    }

    /**
     * Returns the raw store where elements are saved.
     *
     * @return the array in which elements are stored.
     */
    protected double[] getStore() {
        return store;
    }
}

```

Listing of the MutableDenseRealMatrix Class

```

final class MutableDenseRealMatrix extends DenseRealAbstractMatrix
{
    MutableDenseRealMatrix(int rows, int columns,
                           Properties characteristics) {
        super(rows, columns, characteristics);
    }

    @Override
    public Matrix<Double> setElement(int row, int column,
                                      Double value) {
        Objects.requireNonNull(value);

```

```
        checkCoordinates(row, column);
        set(row, column, value);
        return this;
    }

    @Override
    protected AbstractMatrix<Double> targetMatrix() {
        return this;
    }

    @Override
    protected AbstractMatrix<Double> newMatrix(
        int rows, int columns) {
        return new MutableDenseRealMatrix(
            rows, columns, getCharacteristics());
    }
}
```

The lineage of four classes (`AbstractMatrix`, `RealAbstractMatrix`, `DenseRealAbstractMatrix`, and `MutableDenseRealMatrix`) eliminate duplication. The `AbstractMatrix` is the base class from which all the others are derived. It implements the basic version of arithmetic operations using element accessor template methods (the usage of the *Template* design pattern). The auxiliary methods are also implemented by this class. These implementations can be overridden if a more advanced kind is required (e.g., a highly specialized matrix multiplication utilizing GPUs).

Java doesn't support operator overloading, so the following will not compile (see Exercise 15):

```
public final class OperatorOverloadingCase<E extends Number>
    implements Calculate<E> {
    @Override
    public E add(E x, E y) {
        return x + y;
    }

    @Override
    public E multiply(E x, E y) {
        return x * y;
    }
}
```

You would get the following compiler errors, despite E being a kind of a Number:

```
The operator + is undefined for the argument type(s) E, E  
The operator * is undefined for the argument type(s) E, E
```

This is the motive for the Calculate interface and the intermediate RealAbstractMatrix class. It lowers the abstraction level from E extends Number to Double. The approach here is a simplified variant of the more general way to denote matrix elements (e.g., through E extends FieldElement<E>, as is done in the Apache Commons Math library).

The DenseRealAbstractMatrix class introduces an array store for matrix elements and implements the element accessors. Here, the store is embedded inside the class and is a fine example of the *Composition over Inheritance* principle (this is especially pronounced in the case of the sparse matrix with an inner hash map object). The leaf MutableDenseRealMatrix class adds the logic to support a mutable matrix version. The new class diagram is presented in Fig. 5.13.

At any rate, the developer of the library has published a new 3.0.0 version, and everybody was happy with the endeavor. The team was aware that the class hierarchy is a bit deep. They have devised a remedy for this but needed to wait for the next restructuring opportunity.

5.25 Matrix Transposition: Problem

The 3.0.0 version was more important inside the company than to the users of the EDA system. However, the company had announced its ability to quickly address now new feature requests from users. One such feature had demanded a new matrix operation from the linear algebra library, namely the capability to transpose a matrix.

5.26 Matrix Transposition: Resolution

The developer to whom this change request was assigned confidently started the job. He has increased the version number to 3.1.0 and has extended the client API to include this new operator (see Exercise 17). The implementation was easy, and thanks to the previous restructuring effort, there was no duplication of code. Here are the patches to the corresponding entities of the linear algebra library:

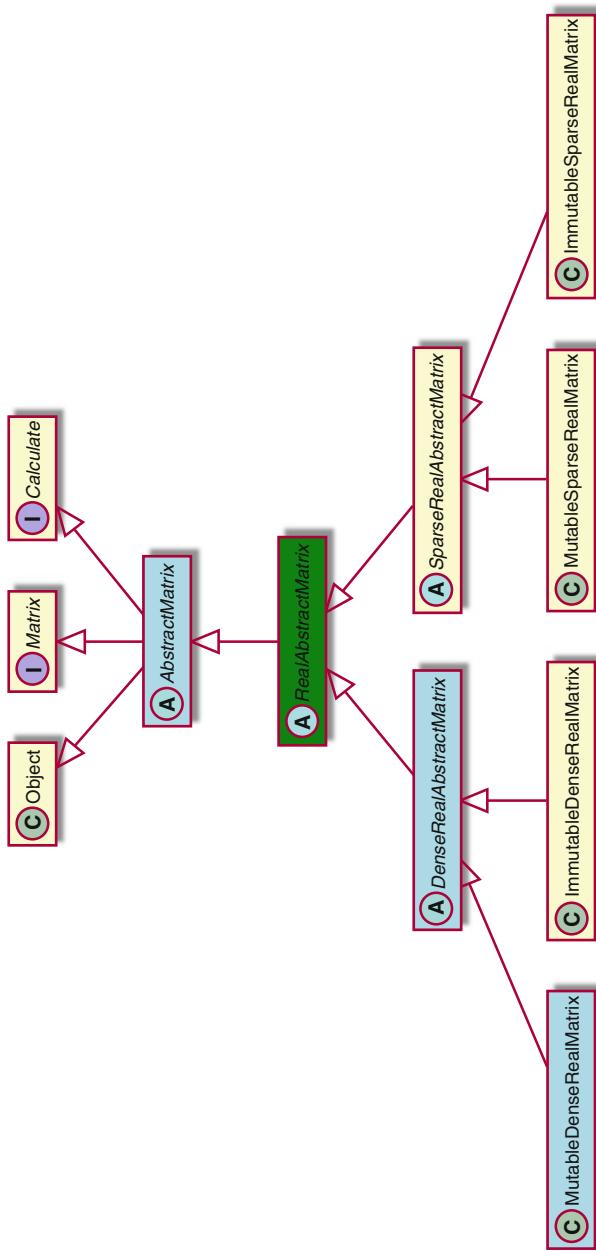


Fig. 5.13 The new class hierarchy (the `IntegerAbstractMatrix` class would be a sibling of the `RealAbstractMatrix` class). At any rate, we see that all of them descend from the `AbstractMatrix` class, so there is a clear message that they are matrices

Listing of the Implementation of Matrix Transpose in the AbstractMatrix Class

```

@Override
public Matrix<E> transpose() {
    final AbstractMatrix<E> result = newMatrix(columns, rows);
    for (int row = 0; row < rows; row++) {
        for (int column = 0; column < columns; column++) {
            result.set(column, row, getElement(row, column));
        }
    }
    return result;
}

```

Listing of the Test Case for this New Operation

```

@Test
public void transposeTheMatrix() {
    Matrix<Double> transpMatrix3x4 = matrix4x3.transpose();
    assertEquals(3, transpMatrix3x4.getRows());
    assertEquals(4, transpMatrix3x4.getColumns());
    for (int row = 0; row < 3; row++) {
        for (int column = 0; column < 4; column++) {
            assertEquals(
                matrix4x3.getElement(column, row),
                transpMatrix3x4.getElement(row, column));
        }
    }
}

```

If you have accomplished Exercise 13, then you should appreciate that all the other matrix types would immediately support this transpose operation. The developer was excited that this has been done without a single duplication of effort. Since he had lot of extra time, this was a perfect opportunity to spend it on refactoring the class hierarchy. Such a slack time was inconceivable in the past! It was crucial to keep the previous client API test intact.

The idea revolves around exchanging the Template design pattern for deciding whether to use the current instance or not with the Strategy design pattern. The Template pattern forces you to use inheritance to customize behavior. The Strategy pattern motivates composition. With this move, you may decrease the inheritance tree's depth (in our case by one). The API for the mutability strategy contains only that template method `targetMatrix`. Therefore, the developer has first used the *Extract Interface* refactoring (see [11] for an extensive treatment on how to

refactor code). This refactoring has resulted in the new interface called `MutabilityStrategy`. Here is the listing of this interface followed by two concrete strategies:

Listing of the `MutabilityStrategy` Interface

```
/**  
 * Defines the API to control mutability of matrices.  
 *  
 * @author ACME Developer  
 * @since 3.1  
 * @param <E> the content type of this matrix  
 *         (for example, Double, Integer, etc.).  
 */  
interface MutabilityStrategy<E extends Number> {  
    /**  
     * Returns the reference to the target matrix.  
     *  
     * @return the reference to the target matrix. It can  
     *         be this matrix if it is mutable, or a new  
     *         instance in case of an immutable one. The  
     *         dimensions and characteristics of the returned  
     *         matrix will be the same as of the caller.  
     */  
    AbstractMatrix<E> targetMatrix();  
}
```

Listing of the Mutable Strategy

```
/**  
 * Implements the mutable strategy.  
 *  
 * @author ACME Developer  
 * @since 3.1  
 */  
final class Mutable<E extends Number>  
    implements MutabilityStrategy<E> {  
    private final AbstractMatrix<E> matrix;  
  
    Mutable(AbstractMatrix<E> matrix) {  
        this.matrix = matrix;  
    }  
  
    @Override  
    public AbstractMatrix<E> targetMatrix() {
```

```

        return matrix;
    }
}

```

Listing of the Immutable Strategy

```

/**
 * Implements the mutable strategy.
 *
 * @author ACME Developer
 * @since 3.1
 */
final class Immutable<E extends Number>
    implements MutabilityStrategy<E> {
    private final AbstractMatrix<E> matrix;

    Immutable(AbstractMatrix<E> matrix) {
        this.matrix = matrix;
    }

    @SuppressWarnings("unchecked")
    @Override
    public AbstractMatrix<E> targetMatrix() {
        return (AbstractMatrix<E>) matrix.clone();
    }
}

```

The mutable strategy always returns the referenced matrix, while the immutable strategy returns the deep clone of it. In this way, the returned value is either the current matrix or a copy. The `AbstractMatrix` class also implements the `MutabilityStrategy` interface as well as the `Cloneable` markup interface. It holds a reference to the concrete mutability strategy that defines whether we have a mutable or immutable matrix. The following listing shows the new and modified parts of this class compared to the previous version. You should take a note how cloning of the mutability strategy is done in the code (it is a bit tricky, but nothing serious):

Listing of the Patch for the `AbstractMatrix` Class

```

abstract class AbstractMatrix<E extends Number>
    implements Matrix<E>, Calculate<E>,
               MutabilityStrategy<E>, Cloneable {
    private final int columns;
    private final int rows;
    private volatile Properties characteristics =

```

```
        new Properties();
private volatile MutabilityStrategy<E>
    mutabilityStrategy;

AbstractMatrix(int rows, int columns,
    Properties characteristics) {
    assert characteristics != null :
        "Characteristics is missing";

    this.rows = rows;
    this.columns = columns;
    this.characteristics.putAll(characteristics);
}

/**
 * Returns the reference to the target matrix.
 *
 * @return the reference to the target matrix. It can
 *         be this matrix if it is mutable, or a new
 *         instance in case of an immutable one. The
 *         dimensions and characteristics of the returned
 *         matrix will be the same as of the caller.
 */
@Override
public final AbstractMatrix<E> targetMatrix() {
    return mutabilityStrategy.targetMatrix();
}

/**
 * Returns back the mutability strategy associated
 * with this matrix.
 *
 * @return the strategy that controls mutability.
 */
protected MutabilityStrategy<E> getMutabilityStrategy() {
    return mutabilityStrategy;
}

/**
 * Sets the mutability strategy for this matrix. This
 * method is supposed to be called by the factory
 * during initialization.
 *
 * @param mutabilityStrategy the new mutability strategy.
 */
```

```
protected void setMutabilityStrategy(
    MutabilityStrategy<E> mutabilityStrategy) {
    assert mutabilityStrategy != null;
    this.mutabilityStrategy = mutabilityStrategy;
}

/**
 * Clones the given mutability strategy and assign it
 * to the caller.
 *
 * @return the cloned mutability strategy.
 * @param origMutabilityStrategy the strategy to clone.
 */
protected void cloneMutabilityStrategy(
    MutabilityStrategy<E> origMutabilityStrategy) {
    MutabilityStrategy<E> clonedMutabilityStrategy;
    if (origMutabilityStrategy instanceof Mutable) {
        clonedMutabilityStrategy = new Mutable<>(this);
    } else {
        clonedMutabilityStrategy = new Immutable<>(this);
    }
    setMutabilityStrategy(clonedMutabilityStrategy);
}

@SuppressWarnings("unchecked")
@Override
protected Object clone() {
    AbstractMatrix<E> clonedMatrix = null;
    try {
        clonedMatrix = (AbstractMatrix<E>) super.clone();
        clonedMatrix.characteristics =
            (Properties) characteristics.clone();
        clonedMatrix.cloneMutabilityStrategy(
            mutabilityStrategy);
    } catch (CloneNotSupportedException e) {
        e.printStackTrace();
    }
    return clonedMatrix;
}
}
```

The old `MutableDenseRealMatrix` class has gone and is melted into the `DenseRealMatrix` class. By pruning away, the previous leaf class that has controlled the mutability aspect via inheritance, we have managed to lessen by one the depth of our class hierarchy. The intermediate `RealAbstractMatrix` class remained as is. Nonetheless, keep in mind that by implementing Exercise 15, you can further reduce the previously mentioned inheritance depth. In object-oriented development, a multitude of tiny classes with convoluted inheritance graph considerably hinders maintainability. When functionality is spread over dozens of classes, it isn't easy to collect all the chunks together to grasp the global picture. Miller's law states that on average, we may deal with 7 ± 2 objects at a time, i.e., this is the number of details we can store in our working memory.

Next is the listing of the restructured `DenseRealMatrix` class:

Listing of the Restructured `DenseRealMatrix` Class

```
/**  
 * Class that implements a dense matrix with real numbers  
 * as elements.  
 *  
 * @author ACME Developer  
 * @since 3.0  
 */  
final class DenseRealMatrix extends RealAbstractMatrix {  
    private volatile double[] store;  
  
    DenseRealMatrix(int rows, int columns,  
                    Properties characteristics) {  
        super(rows, columns, characteristics);  
        store = new double[rows * columns];  
    }  
  
    @Override  
    public Object clone() {  
        DenseRealMatrix clonedMatrix =  
            (DenseRealMatrix) super.clone();  
        clonedMatrix.store = store.clone();  
        return clonedMatrix;  
    }  
  
    @Override  
    public Double getElement(int row, int column) {  
        checkCoordinates(row, column);  
        return store[elementIndex(row, column)];  
    }
```

```

@Override
public Matrix<Double> setElement(int row, int column,
        Double value) {
    Objects.requireNonNull(value);
    checkCoordinates(row, column);
    AbstractMatrix<Double> targetMatrix =
        getMutabilityStrategy().targetMatrix();
    targetMatrix.set(row, column, value);
    return targetMatrix;
}

@Override
protected void set(int row, int column, Double value) {
    assert value != null;
    store[elementIndex(row, column)] = value;
}

@Override
protected AbstractMatrix<Double> newMatrix(
        int rows, int columns) {
    DenseRealMatrix newMatrix = new DenseRealMatrix(
        rows, columns, getCharacteristics());
    newMatrix.cloneMutabilityStrategy(
        getMutabilityStrategy());
    return newMatrix;
}

/**
 * Returns the raw store where elements are saved.
 *
 * @return the array in which elements are stored.
 */
protected double[] getStore() {
    return store;
}
}

```

The final touch is pertaining to the factory class (see the accompanying source code for the new version). The instantiation of a matrix is now a two-step process: creating a new instance and setting the proper mutability strategy. This is a fine example why factories are preferable over direct constructor calls. Inside a factory, you have bigger control how instances are created and there is less danger that a client will work with incomplete objects. The final class hierarchy is shown in Fig. 5.14.

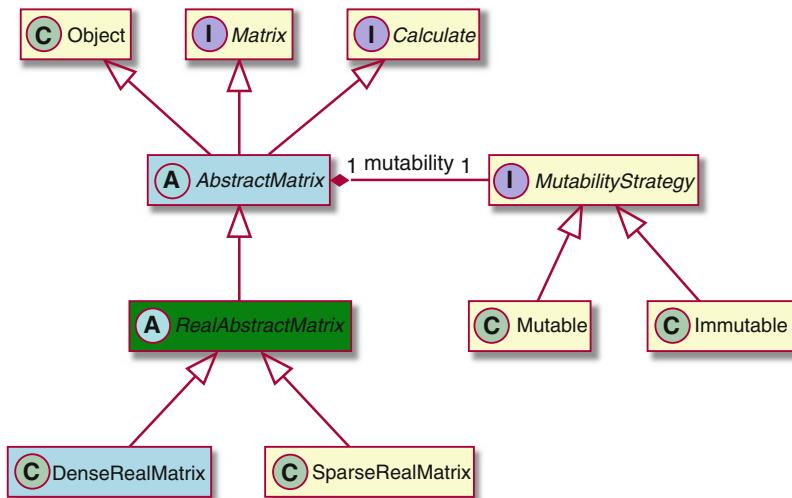


Fig. 5.14 The class hierarchy is now shallower and contains fewer classes. To reduce clutter, the UML class diagram doesn't show that the `AbstractMatrix` also implements the `MutabilityStrategy` interface

After all these changes, the developer has published the 3.1.0 version of the linear algebra library (see Exercises 20 and 21). The EDA software has managed to thrive on the market due to the last-minute reaction of the management to improve quality and enable further evolution.

5.27 Summary

Without proper preliminary work and focus on quality, there is no smooth evolution. In this aspect, the software development and maintenance/evolution is reminiscent of a chess game (I was a former chess player). In professional chess, you have different phases of a game; that begins with preparation. This is where you analyze the options against your next opponent (using experience if possible), judge your current standing at the tournament, and based upon all this, devise the strategy. Opening, middle game, and end game are reflecting what you have decided from the very beginning. During the game, you create plans, depending on what you wish to achieve and what your opponent is doing (like change requests during evolution). At any rate, you cannot play tournament chess without a plan, as your position wouldn't be evolvable. A professional chess game isn't just a sequence of context-free moves, where you evaluate the next move solely based upon the current position. The ability to plan is tightly related to theory, knowledge, and experience. In amateur chess, where planning as a notion doesn't even exist, theory doesn't matter neither. The same is true for software engineering. A professional software

engineer cannot behave like an amateur, where the whole endeavor revolves around trying something out, fixing broken things, and hoping for the best.

You have witnessed what happens in an ad hoc software engineering. Early bad decisions follow you unwaveringly along the evolutionary path and create obstacles. At one point you must stop and perform a major overhaul. You cannot usually fix or eliminate all consequences of previous mistakes at once, but you may gradually improve things. In chess, if you make a mistake in the opening, the rest of the game is a struggle for a draw, unless your opponent also makes an equivalent mistake and gives you another chance to gain the advantage. Unfortunately, the latter isn't what professional players count on since they don't base their destiny on pure luck. Similarly, you cannot expect that a bungled software will simply run while customers and competitors stay quiescent. These groups will actively force you to alter your solution (besides influences from the environment itself), and if your software isn't ready for evolution, then it is game over for you.

Software engineers must be responsible to drive their profession. They cannot passively watch and react to whishes from outsiders of the profession, for example, to irresponsibly accept impossible missions and produce a low-quality product. Software engineers must also be proficient to fight all sorts of negative hypes that may endanger their profession. You have probably heard the mantra "*We don't require plans, since we are 100% Agile*" many times in companies. This empty catchphrase bears two detrimental upshots: it blows up the company and makes dishonest service to the term Agile. Try to "impress" a professional chess player by starting the game with 1. b4 and telling him that you're following the latest Martian opening, which is nowadays so popular on blogs. Will his response be equally crazy, like 1. . . . b5? Of course not. He would just play 1. . . . e5 and immediately start controlling the center squares. You would need to defend your pawn on b4 and further postpone the battle to control the center of the table.

The key takeaways from the chapter are:

- The development cannot be disconnected from maintenance and evolution. You must invest in requirements and architecture and properly estimate the crucial vector of changes.
- Quality cannot be treated as an aftermath that can be built in as an additional feature of the product. The pertinent quality properties of the system must be reflected in the architecture.
- The difference between Agile and plan-driven approaches isn't that the former bans plans while the latter is only about plans. Both must judiciously gauge how much upfront design is necessary in a given context. Fortune cannot scale nor is repeatable.

5.28 Exercises

1. Try to find in [3] what principles were broken at the beginning of our story.

2. Is it a good decision to call the `getElement` and `setElement` public methods from other public methods in version 1.0.0? What is the result of executing the following code (does the `addAll` rely on `add`)?

```
class FunnyArray<E> extends ArrayList<E> {  
    @Override  
    public boolean add(E e) {  
        return false;  
    }  
  
    @Override  
    public void add(int index, E element) {}  
};  
  
FunnyArray<Double> myArray = new FunnyArray<>();  
myArray.addAll(<some collection>);  
System.out.println(myArray);
```

3. Can you untangle solely by looking at the API of the `Matrix` class (version 1.0.0) whether elements are indexed from zero or one? If not, then what would you do? Would you bet on your assumption to be always true in the future?
4. Does the `toString` method of the `Matrix` class in version 1.0.0 produce an intuitive output? How would you differentiate between the `toString` output from an `ArrayList` object and a `Matrix` instance?
5. The code is in version 1.0.1 tested purely with nonsquare matrices. Is this a step forward? Is it valuable to also have tests for special cases (like working with square matrices)? (Hint: Recall the tenets of the *equivalence partitioning* technique.)
6. Suppose that one of the unit test in version 1.0.2 was implemented in the following way:

```
@Test  
public void multiplyMatrixWithScalar() {  
    assertEquals(  
        "[0.0, 10.0, 20.0, " +  
        "10.0, 20.0, 30.0, " +  
        "20.0, 30.0, 40.0, " +  
        "30.0, 40.0, 50.0]",  
        matrix4x3.multiplyWithScalar(10.0).toString());  
}
```

Why should you avoid such a unit test? (Hint: Think about the independence of tests and the fact that this relies on the proper functioning of the `toString` method.)

7. The unit tests in version 1.0.2 are still working with integers converted to double. Moreover, it is possible to create a matrix with even negative dimensions. Enhance the tests to reflect these edge cases (you will also need to alter the constructor). After you have finalized this part, think about other error scenarios. What would happen if the input to `addMatrix` or `multplyMatrix` is `null`? Should you also protect the `getElement` and `setElement` methods?
8. Does this prodigious resolution to add meta-data to the collection, backing the version 1.0.2 of the `Matrix` class, hide a secret behavior? (hint: remember that the matrix is initialized with zeros).
9. How usable are the tests in version 1.1.0, aimed to exercise a sequential code, for a multithreaded variant of the matrix class? Can you rely on automated tests to say anything about the correctness of a concurrent program?
10. The implementation of the `toString` method in version 2.1.0 contains the following snippet:

```
if (column != columns - 1) {
    buff.append(' ');
}
```

How would you improve this to be more expressive and readable? Think about what that condition represents. And would it help if you could refactor the code to look like this?

```
boolean <<YouFigureOutWhat>> = column != columns - 1;
if (<<YouFigureOutWhat>>) { ... }
```

11. Repeat the same process for the `MutableMatrixUnitTest` and `ImmutableMatrixUnitTest` classes in version 2.1.0 of the linear algebra library.
12. Enhance the tests in version 2.2.0 to use floating-point numbers. You may want to change the `fillMatrix` method to the following:

```
private void fillMatrix(
    MutableSparseMatrix matrix,
    int rows, int columns) {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < columns; j++) {
            matrix.setElement(
                i, j,
                Math.pow(-1.0, i + j) * (i + j) / 3.0);
        }
    }
}
```

13. (Challenging project assignment) Implement the missing classes and test suites from the linear algebra library. You should follow the existing design. If you are stuck, then you may peek into the next section, where deep cloning is implemented for matrices. Try to introduce an integer matrix type too. Observe how much of such an inclusion increases the complexity and causes considerable proliferation of new classes. The Curry-Howard correspondence or isomorphism establishes a formal relationship between type theory and proof theory (systems of formal logic). How evident is this connection if you try to ascertain guarantees about execution looking at the resulting Java code (after you're done with the previous extension). The Java Generics facility is rather awkward since it was introduced in Java SE 5 in a backward-compatible manner using a technique called Type Erasure. However, that same procedure could have been named Logic Erasure too (see [10] for interesting viewpoints on these and other aspects of software engineering).
14. In version 3.0.0, the client still needs to know the exact factory class. This may complicate the realization of the API-centric path using an intermediary snapshot release. Restructure the design to leverage the *Abstract Factory* design pattern. You would need to introduce a factory API in the form of an interface and possibly use dependency injection (see Chap. 4) to dynamically provide the client with the development or production factory class.
15. You may want to try out the *Polyglot Programming* principle and implement version 3.0.0 provider side in some JVM language that supports operator overloading (e.g., Groovy or Scala). Also, try to rewrite the client integration test suite using the Spock framework (visit spockframework.org). This is a testing DSL built on Groovy that allows you to write more readable and powerful test cases. Another viable option is to rewrite the client integration test suite using the Hamcrest matchers (visit hamcrest.org). All in all, you should always strive to try out many design and implementation options before settling down with the final one. In a book, I cannot perform such extensive explorations, but you can and should in your daily work. The best designers never consider only one or two choices, though.
16. In the implementation of the `equals` method in the `AbstractMatrix` class in version 3.0.0, you can find the following snippet:

```
isEquals = getElement(row, column)
    .equals(
        otherMatrix.getElement(row, column));
```

Is there any reason to use `equals` instead of `==`? Will a check `assertEquals(myMatrix, myMatrix)` reveal anything? What about `assertEquals(myMatrix, cloneOfMyMatrix)`?

17. In version 3.1.0, the transpose operation similar to the matrix multiplication returns a new matrix instance. The documentation says that it is cumbersome to perform it in place. Unfortunately, the matrix addition and multiplication with

scalar are in place operators, while matrix multiplication and transpose are not. Such an inconsistency in the API is problematic. We experience here the consequences of earlier poor decisions to buttress single-threaded and botched concurrent variants that had transformed into mutable and immutable types. Code restructuring may help us to some level to correct past mistakes, but you cannot expect miracles. Sometimes, when cruft uncontrollably pile up, maybe a full rewrite is a better option. Of course, such a decision must be aligned with the business.

18. Restructure your classes from Exercise 13 to follow the design established in version 3.1.0.
19. (Challenging) Expand the conversion capability of version 3.1.0 of the system to element types too. In other words, create a method in the factory class to transform, for example, a real matrix into integer, and vice versa.
20. The `equals` method in the `AbstractMatrix` class has the following comment:

```
// A very inefficient implementation of equals.  
// Children classes may want to override this,  
// and provide a much faster variant.
```

How useful is this comment? Is it only about speed? (Hint: Think about the possible substitutions for the generic parameter `E` and how they are compared).

21. (Challenging) There is one hideous instability in one of the methods in the current code base. Can you spot which one? (Hint: Consider the source code of the 3.1.1 version of the matrix library in Chap. 7, and try to find the difference).
-

References

Further Reading

1. Guttag J, Liskov B (2000) Program development in Java: abstraction, specification, and object-oriented design. Addison-Wesley Professional, Boston, MA. This book is the foundation for learning how to produce high quality products using Java. It fully covers the systems development life-cycle, and contains explanations for doing object-oriented program design and component-based software development

Regular Bibliographic References

2. Kuipers T (2016) Why you need to know about code maintainability. www.oreilly.com/ideas/why-you-need-to-know-about-code-maintainability. Accessed 25 Aug 2017
3. IEEE/ACM (2015) Software engineering code of ethics and professional practice. www.acm.org/about/se-code. Accessed 26 Aug 2017

4. Robinson I (2008) Service-oriented development with consumer-driven contracts. www.infoq.com/articles/consumer-driven-contracts. Accessed 26 Aug 2017
5. Mili H, Mili A, Yacoub S, Addy E (2001) Reuse-based software engineering: techniques, organizations, and controls. Wiley Inter-Science, Chichester
6. Mind Tools Editorial Team (2017) 5 whys: getting to the root of a problem quickly. www.mindtools.com/pages/article/newTMC_5W.htm. Accessed 28 Aug 2017
7. Kamp HP (2012) A generation lost in the bazaar. queue.acm.org/detail.cfm?id=2349257. Accessed 29 Aug 2017
8. Sommerville I (2014) Warsaw airbus accident. www.slideshare.net/sommerville-videos/warsaw-airbus-accident. Accessed 30 Aug 2017
9. Bloch J (2008) Effective Java, 2nd edn. Addison-Wesley, Upper Saddle River, NJ
10. Seibel P (2009) Coders at work: reflections on the craft of programming. Apress, New York
11. Fowler M, Beck K, Brant J, Opdyke W, Roberts D (1999) Refactoring: improving the design of existing code. Addison Wesley, Upper Saddle River, NJ

In this chapter, we will see the opposite to what we have observed with ad hoc development. This chapter highlights the most important ingredients of a successful product and systematically develops a brand new microservice that solves infrastructural problems in the existing services of our imaginary corporation. Afterward, we will add a couple of extensions to the developed service, showing how evolution is smooth if things are properly done during initial development.

There are many ingredients of a successful product,¹ and it would be impossible to detail them all (see [1] for an extensive treatment of quality as the foremost concern). However, I will emphasize the following duties that every endeavor should encompass:

- **Understand the core problem (work) and describe the user requirements.** This is fundamental; otherwise, you will never succeed. You must have a clear business case for your software and know exactly how it will solve the customer's problem. In the Volere requirements technique (visit www.volere.co.uk), there is a transitioning flow from extracting the problem to delivering the solution called the *Brown Cow* model [2]. In Chap. 8, I will give some examples of how things may go wrong when you disregard the primary problem and purely focus on technology. When you know the problem, the next step is to establish the scope of the product. This is the territory that you want to support with software and alter the current state of affair. The user requirements are talking in the realm of the problem domain, telling what the product should do to solve the core problem.
- **Specify the system requirements and architecture.** We have talked about the role of the architecture in Chap. 4, but I am again stressing the importance of the

¹Naturally, for us a successful product is evolvable and long lasting. An experimental prototype could be extremely successful, but it has a totally different aim. Usually, it is used as a risk reduction technique and supports the development of the main product. Many Agile methods use the term *spike* for such tryouts.

upfront design work including requirements. The system requirements must detail in a testable fashion how the software should behave to address the user requirements. These requirements will drive the design, construction, and furthermore, evolution of the system. The ability to trace back construction to system requirements and finally to user requirements is crucial during evolution. Remember from Chap. 5 the arbitrary decision of the developer to initialize the matrix with zeros. That was an isolated vacuum-filling deed. Any such vacuum can disappear from or needlessly hang around in the code base.

- **Plan in quality and risk management work.** Projects that neglect quality or postpone risk management in the name of speeding up things will just experience the opposite. The amount of rework will swamp the progress, besides overrunning the schedule. Chapter 5 should serve as a good reference. Firefighting is not risk reduction, as it combats a problem instead of a risk.
- **Practice TDD.** Covering everything with tests and thinking about testability is paramount from the viewpoint of maintenance and evolution. Test-Driven Development (TDD) should start with requirements and continue all along. It may be named differently, like acceptance tests for requirements, BDD for system tests, and so on, but the idea is the same. You must express all decisions in a testable way (a line in your program is also kind of a decision).
- **Avoid duplication of all sorts.** If you must duplicate, then you should have good arguments for it. Many times, a need for duplication is an indicator that something is wrong in the system or that you are too lazy to think enough how to accomplish something without redundancy. Every line of code must be maintained, so less code means less maintenance burden.
- **Attain license in software engineering from a recognized institution.** Every member of a mature profession is licensed. Doctors, lawyers, accountants, civil engineers, and so on must be licensed (or take care not to lose the license) or otherwise cannot practice their profession. This must become true for software engineering too. Just to be correct to readers, I possess the following certificates: Sun Certified Software Developer for Java 2, IEEE Professional Software Engineering Master Certification (I've converted my earlier IEEE Certified Software Development Professional into this), and IEEE Software Engineering Certified Instructor. Google, for example, mandates certification for developers before allowing them to check in code into the source code repository. Certification is in some way quality assurance for people. Some argue that certification cannot guarantee a bug-free code base or some similar peculiarity. This is true; yet more so, this is a universal truth. Despite knowing that there is no guarantee that a doctor will prescribe a proper treatment, most patients still prefer a licensed doctor than a shaman.

In the rest of this chapter, we will develop an initial version of our microservice and afterward, add three enhancements. We will follow the above recommendations and start with the problem description. The service itself isn't that much more important than the principles and techniques showcased in the process. Greenfield and brownfield projects have many commonalities, and what

we will describe next is in general applicable to both. You must understand the problem, manage requirements, implement the solution, and deliver the new software. Certainly, there are also differences but not that much as one would think. As the case study below shows, sometimes a greenfield project is initiated to resolve an issue in the existing system. Producing technology wrappers around legacy systems is another good example.

6.1 Initial Version: Problem

Most programmatic Web services out there belong to the category named as Resource API services (most of the time, these are also titled as RESTful, but that is an abomination). Using the Richardson maturity model [3] they would be classified as level 2 REpresentational State Transfer (REST) services. Most of them do have service descriptors that principally act as API documentation. They are also used to autogenerate both client and server code. Many service descriptor standards exist, but we will focus on OpenAPI specification 2.0² (visit swagger.io/specification/ for the full description of the latest version as well as for the link to version 2.0). Swagger is the API tooling (visit swagger.io) built around the OpenAPI specification.

The ACME Corporation has lots of services described in this specification, and each of them exposes a Swagger User Interface (UI)³ endpoint. Hitting this URL renders the matching service descriptor in HTML. Thus, the user may read the API documentation and even experiment with the service by sending HTTP requests. The HTML page also gives hints what are the mandatory fields and automatically populate an HTTP request's body with a template. The benefits are manifold:

- The API documentation is in standard format (both the content as well as the serialization mode, which is YAML and JavaScript Object Notation (JSON)).
- All necessary information about the service is available from a single place. The OpenAPI service descriptor can contain a link to external documentation that could point the user to another piece of content.
- The documentation “travels” with the service.
- The documentation is active, as it enables the user to try out the service with assistance from the tool. The Swagger UI reports an error if the documentation is ill formed, which is another benefit of keeping it active.

²The latest version 3.0 just came out at the time of this writing and isn't yet fully supported by tools.

³The *Swagger Editor* is the primary tool to work with an OpenAPI service descriptor. The generated specification can be rendered in HTML by the *Swagger UI* embeddable engine. Swagger UI allows someone to invoke the target service with data entered via the autogenerated HTML page. To see what all this looks like, visit editor.swagger.io, and it will open the Swagger Editor with the sample Petstore server's API.

Nonetheless, the QA team who mostly uses this Swagger UI facility is complaining about all sorts of issues with services. In one group of services, the Swagger UI reports an error, while invoking the services directly via curl executes without problems. In another case, the requests properly go through the latest version of the Swagger Editor but fail when called via the embedded Swagger UI engines provided by the services. The team also grumbles that downloading the images of services from the corporate repository is taking too much time. The ACME corporation is interested in remedying this situation.

6.2 Initial Version: Resolution

Our first job is to understand what is going on. The fact that the QA team is cranky doesn't mean that they are not happy overall with this Swagger UI choice. So, hastily proposing something else instead of Swagger and trying to increase the capacity of the corporate network are ineffectual attempts to address their needs. A deeper analysis is required.

6.2.1 Understanding the Problem

After scrutiny, the conclusion is that the current approach of embedding the Swagger UI in each service has many drawbacks⁴:

- The embedded Swagger UI considerably bloats the footprint of the service.
- There is tight coupling between the service and the Swagger UI. To update the latter, the service must be upgraded and deployed with a new version.
- Developers must know the details of deploying and configuring Swagger UI to deliver their own service. This slows down progress and introduces a chance of error (the same effort is repeated multiple times).
- It is hard to keep all services up to date regarding the Swagger UI, i.e., to ensure that they all use the same version.
- The Swagger UI engine is running inside the service, which negatively impacts its performance.
- There is no way to share common service descriptors. For example, every service would need to separately provide a service descriptor for common company-wide established endpoints (a classical example of duplication).
- It isn't possible to peruse a service via Swagger UI that doesn't expose a Swagger UI endpoint.

⁴The lack of a central registry with those Swagger UI endpoints isn't directly connected with the core problem. Let us assume that the relative URL for a Swagger UI rendered page in a service is standardized in ACME. In this case, any central registry for services would do.

The above elaboration decently describes the problem, as each bullet point indeed speaks in terms of problem space. The arguments for acting are sound, and they are based on facts. There is a clear connection between symptoms (the complaints from the QA team) and discovered shortcomings. For example, due to currently tedious update mechanism of the Swagger UI engine, it isn't surprising that some services may use an old version containing defects. Keep in mind that the previous list is the result of apprehending the problem and requires a lot of time and hard labor. Don't expect users to approach you with such a distilled report.

The next step is to map the collected items onto the *Kano Model* (visit www.kanomodel.com for more details) shown in Fig. 6.1. As we invest more effort and implement features, the satisfaction level changes as a weighted sum of satisfactions per each axis. The linear performance line is intuitive since as we deliver more, the satisfaction rises equally. The excitement quickly rises, and there is no sense in investing too much in it. Even a simple push makes an effect. However, a zero investment doesn't lower the satisfaction. Contrary to this is the curve representing the basic needs. If we don't deliver what already works then the satisfaction drops swiftly. On the other hand, we cannot impress by giving what the user already has.

I've witnessed many improvement tries in my career, where a solution for one problem also created new problems. What usually happens is that the extreme focus on issues creates blind spots that hide excellent stuff in the current system. Omitting these is fatal. The Kano diagram is superb in reminding us what must not be forgotten. I must admit that this isn't a typical usage of the Kano model;

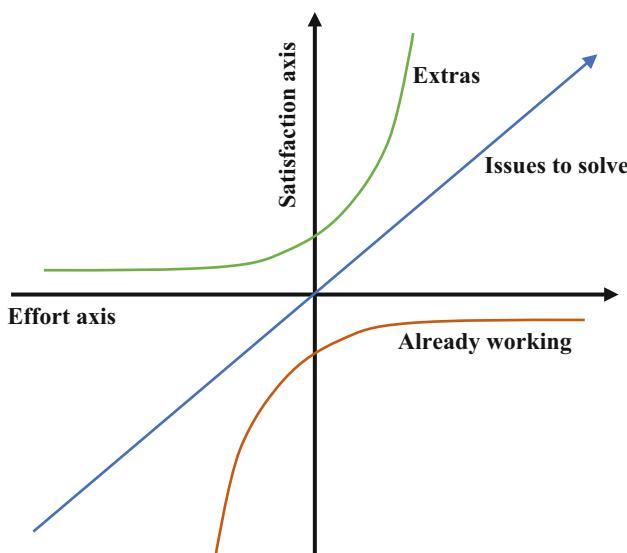


Fig. 6.1 The three principal directions in the Kano model: excitement (extras), performance (issues to solve), and basic (stuff that already works). The effort axis could be expressed in various units, like time, number of features, etc.

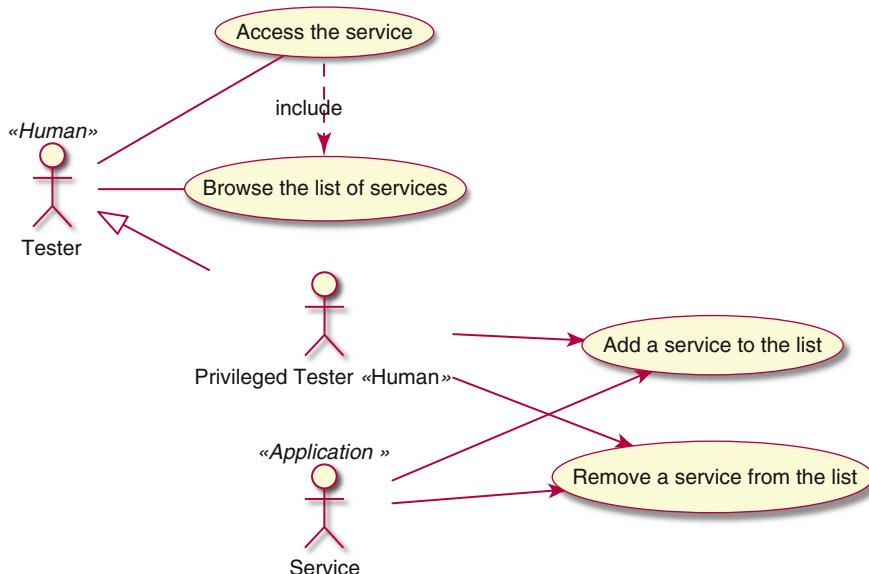


Fig. 6.2 The UML use case diagram portrays the major goals that users expect from the product. The diagram also shows various actors and their interrelationships. The communication between an actor and a use case is usually bidirectional. However, some are typically directed as indicated by arrows

nevertheless, it works for me in practice. At any rate, the extras would be the last two items from our list above and centralizing the Swagger UI endpoints. The issues to solve would be the other drawbacks from our list. The basic needs are the stuff that already works nicely. By putting these items directly onto the diagram, we can prioritize them based upon their impact on the total satisfaction.

To depict the user requirements on a high level, we will utilize the UML use case diagram as shown in Fig. 6.2. It includes the user's major goals and delineates the product's scope (see [4] for details how to write down use cases in textual form). Everything that isn't encompassed with this set of use cases is outside of the business bounds. It is very important to stress the notion of a business bound since such a clear border line is essential to successfully apply the microservices architectural style (that we intend to use here). Of course, we haven't touched yet the solution space, and scoping the product is a precondition for any solution, though.

The use case diagram isn't a design tool! It is a mechanism to express pertinent business requirements rather than model the solution. The diagram in Fig. 6.2 contains an include relationship between "Access the service" and "Browse the list of services" use cases. This emphasizes the fact that the user should first select the available Swagger UI endpoints from the list before being able to access a service. At any rate, we are simply structuring the requirements here, as both use cases exhibit a clear goal. For example, the user may only want to see the list of

registered services without accessing them. On the other hand, if you have a use case that doesn't represent a standalone goal at the same level as the others, then you should stop. This might be an indicator that you've started designing the solution.

6.2.2 Specifying the Architecture

Our product will be a level 2 REST microservice called *Swaggerize* that would wrap other services and provide access to them via the Swagger UI engine (read [5] for a thorough explanation about the microservices architectural style). It will also utilize a relational database to store the list of services and their service descriptors. It is important to achieve portability in respect to databases, as we don't want to create an additional burden on the QA team. They would just point our service to their database, and all should work without any additional setup. This is a precondition for us to deliver a self-contained and stateless service (part of the microservices tenet). The solution should also be maintainable, as we do intend to deliver the service in stages. Figure 6.3 shows the deployment view of our architecture.

We will use the Dropwizard framework (visit www.dropwizard.io) to develop our service. Dropwizard is an umbrella Java framework that integrates other frameworks. Therefore, it constitutes a full stack of carefully selected technologies to implement a level 2 REST microservice using a relational database in the

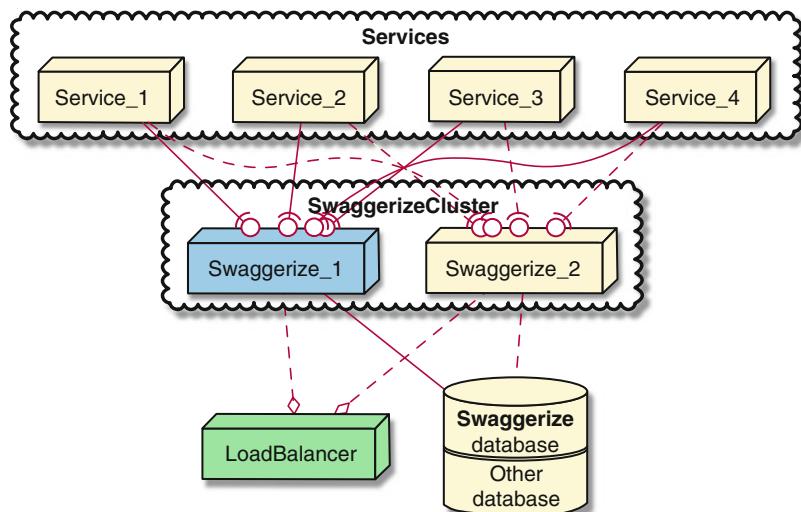


Fig. 6.3 The deployment view of our architecture. The Swaggerize microservice may be a member of the cluster controlled by the front-end load balancer. As our service is stateless, instances may come and go, which enables easy autoscaling of the cluster. However, clusterization is just an additional possibility that may be realized later

background. It has an extensive support for an automated unit, integration, and system testing. It supports database schema evolution via the Migrations module. All in all, by relying on this framework, we are merely left with stating our Resource API (using the OpenAPI 2.0 specification) to fully define the architecture. The API must be part of the architecture and cannot be left to emerge by chance.

An important aspect that even precedes the architecture is the feasibility study. We must analyze the prospect of achieving our vision by considering the constraints imposed by browsers. Cross-origin resource sharing (CORS) is a policy governing the interaction between a browser and a server. It tells whether and how a browser may issue cross-origin requests. Suppose that your browser loads the Swagger UI engine from our service. If you want to fetch some service descriptor delivered by another service (from another domain) through the Swagger UI page, then this is a cross-origin request. It is forbidden by default for security reasons. There are many ways to overcome this limitation. One option is to install a browser plug-in that would setup the necessary CORS headers so that the browser would think that these are set by the server. This is ok if you control your clients, and such plug-in exists for some of the supported browsers by the Swagger UI engine. In our case, both these conditions are met. For example, Google Chrome is one viable choice. In the rest of the text, we will assume that the QA team is willing to accept this route. Otherwise, you would need to devote extra time to deal with this topic. There is an extensive treatment inside the Swagger UI documentation (see the section *CORS support* at swagger.io/docs/swagger-tools/#swagger-ui-documentation).

Creating the Swaggerize Maven Project As this is a Java project, we will use Maven (visit maven.apache.org) as our build tool. We must first create the skeleton of our project in order to incorporate the API description (it will go into `src/main/resources/assets` folder). Inside the parent folder of the project, execute the following shell command⁵:

```
mvn archetype:generate \
-DarchetypeGroupId=io.dropwizard.archetypes \
-DarchetypeArtifactId=java-simple -DarchetypeVersion=1.1.4
```

During the execution of the command, you should answer the questions in the following manner or just accept the default:

⁵At the time of this writing (I've also created a GitHub Issue at github.com/dropwizard/dropwizard/issues/2146), the Getting Started guide has an error. The `archetypeVersion` is set to `1.0.0` instead of `1.1.4`. There is also a link to the `dropwizard/dropwizard-archetypes` GitHub project, where it is properly documented that `archetypeVersion` should be set to the valid Dropwizard version. This is a fine example that you must check the documentation for errors by seeking the source of truth. In this case, it is the `dropwizard/dropwizard-archetypes` project.

```
Define value for property 'groupId': com.example
Define value for property 'artifactId': swaggerize
Define value for property 'version' 1.0-SNAPSHOT: :
Define value for property 'package' com.example: :
com.example.swaggerize
Define value for property 'description': The micro-service to enable
access to other REST services via the Swagger UI engine.
Define value for property 'name': Swaggerize
Define value for property 'shaded' true: :
```

If everything goes well, then you should have your project created inside the `swaggerize` folder. The nice thing is that Maven will autogenerated the standard package structure for various parts of our service (see www.dropwizard.io/1.1.4/docs/manual/core.html). Such standardization of the source code is very important, as we don't need to provide additional explanation. Moreover, our code is now following the Dropwizard convention, so the accidental complexity regarding the code structure is substantially reduced. For simplicity, we will not split our project into multiple Maven modules, though. Now, you may import this newly generated project into your favorite Integrated Development Environment (IDE). I'm going to use Eclipse.

At this point, it is useful to analyze the module decomposition view of the architecture (in this case, modules are various packages) as shown in Fig. 6.4. Based upon this view, we can coordinate our construction activities. During evolution, such a view is indispensable, as it is the basis for impact analysis. It also allows parallelization of work since concurrent changes would not collide if the work breakdown structure (WBS) follows the modular decomposition of the system.

Defining the Swaggerize Resource API The API-centric style empowers the users of our service to review what we intend to provide. In this manner, they can send back valuable feedback early on. They don't need to wait until the service is fully ready, nor should we waste time on rework later in the project. It is also easy to produce a stub of the service based upon the service descriptor. Here is the listing of the service descriptor⁶ created with the Swagger Editor tool:

⁶It is easier to send immediately the service descriptor than to deliver first the so-called *Service Table*. It is a table with rows containing endpoints. The attributes for each endpoint are listed in columns. One possible arrangement can be as follows: service name, HTTP method, URI template (see RFC 6570 at tools.ietf.org/html/rfc6570), request/response content media type (usually JSON or XML based payload), and HTTP response status codes. After all, you can easily autogenerated such a table in HTML format using the Swagger code generator.

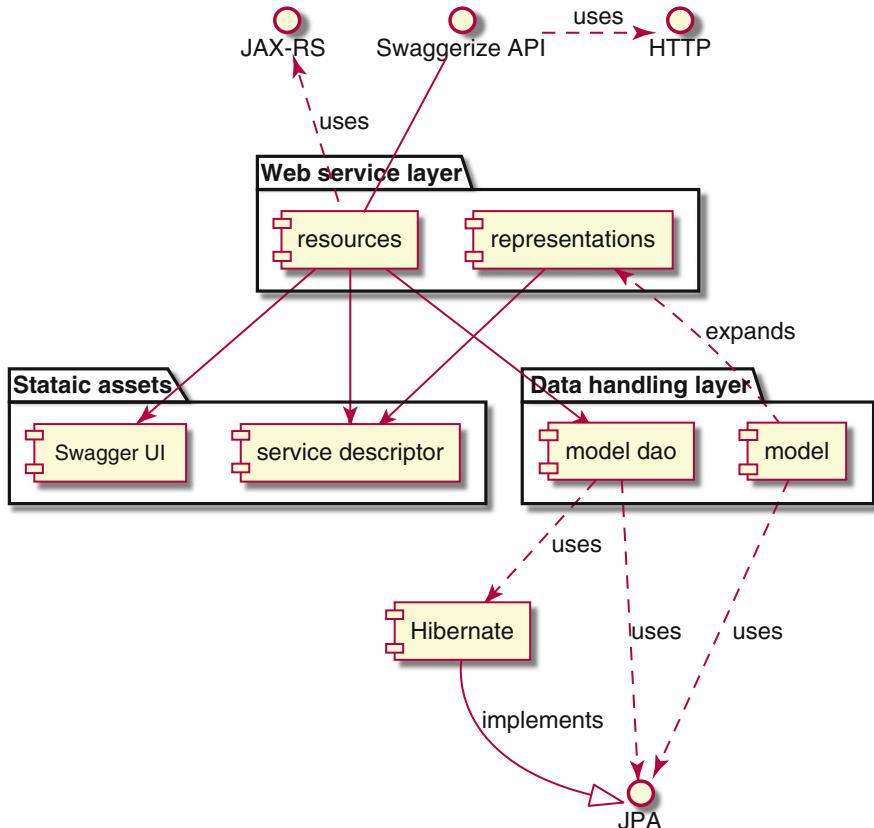


Fig. 6.4 The package structure of our service, together with the dependencies between elements, will be the basis for an organized construction process. The model DAO obviously depends on the model, as its name implies. Moreover, the resources do use representations, but both are initially autogenerated from the service descriptor. The diagram also shows which important technology APIs are exploited by the system

Listing of the Swaggerize API Description

```

swagger: '2.0'
info:
  description: >-
    The micro-service to enable access to other REST services
    via the Swagger UI engine. It supports the following
    major use cases:
    - Accessing the remote service via Swagger UI.
    - Browsing the list of available services in the
      registry.
    - Registering the service.
  
```

```
- Removing the service from the registry.

version: 1.0.0
title: Swaggerize
contact:
  name: Ervin Varga
  email: ervin.varga@exploit.rs
license:
  name: MIT
  url: https://opensource.org/licenses/MIT
basePath: /swaggerize
tags:
  - name: Tester
    description: End-points used by testers.
  - name: Admin
    description: End-points used by privileged users or other
      services.
  - http
consumes:
  - application/json
produces:
  - application/json
externalDocs:
  description: Click here for details about the Swagger UI
    engine.
  url: https://swagger.io/swagger-ui/
paths:
  /services:
    get:
      tags:
        - Tester
      summary: Gets the available services from the registry.
      description: Retrieves the list of registered services.
        This end-point is used to browse the services, and
        select the one to access via Swagger UI.
      operationId: getServices
      responses:
        200:
          description: The list of services, or an empty
            response.
          schema:
            $ref: '#/definitions/Services'
        default:
          $ref: '#/responses/UnexpectedServiceError'
  /services/{serviceName}:
    parameters:
```

```
- name: serviceName
  description: The unique name of the service.
  in: path
  type: string
  required: true
put:
  security:
    - swaggerize-realm: []
  tags:
    - Admin
  summary: Registers the new service, or updates the
    existing one.
  description: Creates a new registration entry, or
    updates the existing record for the given service.
    This end-point is idempotent. It is the caller's
    responsibility to properly name the service.
  operationId: putService
  parameters:
    - name: payload
      description: The URL from where to retrieve the
        service descriptor for the specified service. It
        is assumed that services will provide such an
        end-point.
      in: body
      schema:
        $ref: '#/definitions/ServiceDescriptorURL'
      required: true
  responses:
    201:
      description: The service is properly registered.
      default:
        $ref: '#/responses/UnexpectedServiceError'
delete:
  security:
    - swaggerize-realm: []
  tags:
    - Admin
  summary: Deletes the given service registration entry
    from the system.
  operationId: deleteService
  responses:
    204:
      description: This service is removed from the
        system.
      default:
```

```
$ref: '#/responses/UnexpectedServiceError'  
/services/{serviceName}/access:  
parameters:  
- name: serviceName  
  description: The unique name of the service.  
  in: path  
  type: string  
  required: true  
post:  
summary: Accesses the specified service via the Swagger  
UI engine. The service with the given name must be  
previously registered.  
operationId: accessService  
responses:  
303:  
  description: This service is accessed via Swagger  
  UI.  
default:  
  $ref: '#/responses/UnexpectedServiceError'  
securityDefinitions:  
swaggerize-realm:  
description: The service should be protected by a minimal  
basic authentication mechanism to prevent inadvertent  
usage (the assumption is that this service  
will run behind a corporate firewall).  
type: basic  
responses:  
UnexpectedServiceError:  
description: Unexpected service error (4x or 5x type of  
error).  
schema:  
$ref: '#/definitions/ErrorReport'  
definitions:  
Services:  
description: List of registered services.  
type: array  
uniqueItems: true  
items:  
$ref: '#/definitions/ServiceEntry'  
ServiceEntry:  
description: The record per service in the registry.  
allOf:  
- $ref: '#/definitions/ServiceDescriptorURL'  
- type: object  
  description: The unique name of the service.
```

```

required:
  - name
properties:
  name:
    type: string
ServiceDescriptorURL:
  description: The URL from where to retrieve the service
  descriptor.
type: object
required:
  - url
properties:
  url:
    description: The URL of the service descriptor.
    type: string
ErrorReport:
  description: The application specific error report.
  type: object
required:
  - title
  - status
  - code
properties:
  title:
    type: string
    description: The short-summary of this error in human
    readable form.
  status:
    type: integer
    format: int32
    minimum: 400
    exclusiveMinimum: true
    maximum: 600
    description: The status code associated with this
    error (this is always the HTTP status code as
    returned in the response).
  code:
    type: string
    description: Any application-specific error code.
    pattern: Swaggerize-[0-9]+
example: Swaggerize-5

```

This is the 1.0.0 version of the API, as is indicated in the description itself. I recommend you use `tags` to group endpoints. The grouping in our case follows the modeling of actors in Fig. 6.2, which nicely connects the use case model with the

API. All elements of the API should be properly documented using the `description` and `summary` properties. Such a commentary is especially handy when working with the Swagger UI facility. The `externalDocs` is a good way to provide a link to an external source (in our case, it points to the Web site of the Swagger UI engine). The security aspect is also defined in the API, and it is very simple for our service. The Swaggerize service is solely used by the QA team and isn't going to be exposed outside of the corporate network. For more advanced security details, consult the official OpenAPI specification. Some parts of the API may deserve even a concrete example (like the `code` field of the `ErrorReport` object). Finally, you should take care to avoid duplication. We have achieved this by using the `allOf` construct. Otherwise, the `ServiceDescriptorURL` definition would be repeated twice. Figure 6.5 shows the overall structure of our service descriptor.

The unified error handling is an important detail of the architecture. The API specification introduces an `ErrorReport` entity to hold application-specific errors. The service would therefore need to catch application exceptions while processing requests and turn them into this report. The other errors may be directly passed back to clients by the underlying Web server. In both cases, the HTTP status code will properly reflect the outcome of the request.

The final touch pertaining to API is the setup of the code generation facility. Our ability to speed things up in this way emanates from the API-centric method. We will use the Swagger's code generator to provide us with the initial code base. However, we will not directly use the produced source code (at the time of this

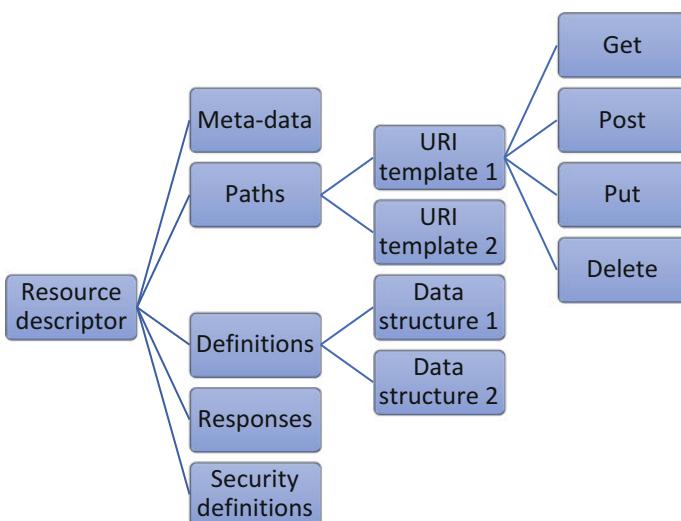


Fig. 6.5 The main building blocks of an OpenAPI specification 2.0 (version 3.0 has a different structure). This structuring is intended to reduce clutter, eliminate duplication, and increase maintainability. It is even possible to share common definitions across multiple descriptors by putting them into a separate file. Therefore, the structuring may continue at the folder level

writing, there is no direct Dropwizard support). The source code will be saved inside the `target` folder (where other compiled artifacts are put), and we will simply pick up what we need from there. Dropwizard relies on the Jersey framework for building Web services that fully implements Java API for RESTful Web Services (JAX-RS) API specification. This will be the target for the code generator. To eschew needless code generation, each time we invoke the Maven `compile` target, we will use the existing Maven property `codegen.skip`. We will set it to `true` inside our build file (`pom.xml`) as a default. It can be easily overridden at the command line by saying `-Dcodegen.skip=false`. All in all, the following Maven plug-in must be added to the build file to activate code generation:

```
<plugin>
    <groupId>io.swagger</groupId>
    <artifactId>swagger-codegen-maven-plugin</artifactId>
    <version>2.2.3</version>
    <executions>
        <execution>
            <goals>
                <goal>generate</goal>
            </goals>
            <configuration>
                <inputSpec>
${project.basedir}/src/main/resources/assets/swagger.json
                    </inputSpec>
                    <language>jaxrs</language>
                    <output>
${project.build.directory}/generated-sources/swagger
                    </output>
                    <modelPackage>
                        com.example.swaggerize.api
                    </modelPackage>
                    <apiPackage>
                        com.example.swaggerize.resources
                    </apiPackage>
                    <invokerPackage>
                        com.example.swaggerize.resources
                    </invokerPackage>
                    <generateApiDocumentation>
                        false
                    </generateApiDocumentation>
                    <generateModelDocumentation>
                        false
                    </generateModelDocumentation>
                    <generateSupportingFiles>
                        false
                    </generateSupportingFiles>
                </configuration>
            </execution>
        </executions>
    </plugin>
```

```
</generateSupportingFiles>
<library>jersey2</library>
</configuration>
</execution>
</executions>
</plugin>
```

If you call `mvn generate-sources -Dcodegen.skip=false` at the command line, then you will get a bunch of autogenerated stuff inside the given output folder (see above). Among them are the representations (HTTP request/response payloads) specified in our API. These classes (`ErrorReport`, `ServiceDescriptorURL`, `Services`, and `ServiceEntry`) come with the predefined `equals` method, nicely formatted string output via the overridden `toString` method, etc. We don't need to alter these classes too much. Also, the generator produces skeletons for resource classes (representing HTTP endpoints) that are an excellent starting point. We just need to move them over into our main code base and modify them accordingly. It is also possible to redirect the output folder into the main code base and prevent inadvertent overwrites with the `.swagger-codegen-ignore` file. I think it is more comfortable to keep generated stuff inside the `target` folder, as the Maven's `clean` target⁷ would properly prune them away (the parts that you don't need). After all, you will anyhow try to avoid changing your API often and substantially so that manually selecting the interesting parts from the autogenerated source code is applicable.

6.2.3 Constructing the Service with TDD

After cleaning up the generated source code (e.g., we don't need Swagger annotations, as we will never produce a specification based upon the annotated code base), we are ready to write our first tests. As all the stuff is autogenerated, you may be tempted to skip unit testing it. However, you should write unit tests. It might happen that some newer version of the code generator has a bug, and you will not notice it without tests (see Exercise 1). Besides unit tests, we also need to perform integration tests with the Jackson JSON parser to ensure that the representations are suitably serialized/deserialized to/from JSON. Thanks to Dropwizard's excellent support for testing and modular structure, everything is ready for us. The **dropwizard-testing** module contains lots of useful artifacts to make Dropwizard-based services easy to test. The **dropwizard-validation** module is used to validate representations and resources and is very beneficial. We first need to boost our build file with the next dependencies:

⁷Sure, it is possible to customize the `clean` target to remove additional artifacts from nonstandard locations, but this would complicate the build file. Imagine having both the `.swagger-codegen-ignore` file and your proprietary cleanup behavior. Nobody would understand what is going on without spending time analyzing your build file.

```

<dependency>
    <groupId>io.dropwizard</groupId>
    <artifactId>dropwizard-validation</artifactId>
</dependency>
<dependency>
    <groupId>io.dropwizard</groupId>
    <artifactId>dropwizard-testing</artifactId>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <scope>test</scope>
</dependency>

```

Implementing and Testing Representations Representations are already autogenerated, so we should continue by testing them. We will start with *fixtures*. The idea is to create a JSON sample in the `src/test/resources/fixtures` folder and use it as a reference while checking the JSON serialization/deserialization mechanism. The sample for the `ServiceEntry` representation looks like as follows (it is named as `ServiceEntry.json`):

```
{
    "name" : "Swaggerize",
    "url" : "http://localhost:8080/swagger.json"
}
```

The cleaned up `ServiceEntry` class (no extra line is added compared to the original) is shown next as well as the matching integration test with Jackson:

Listing of the Tidied ServiceEntry Class

```

/**
 * The record per service in the registry.
 */
@javax.annotation.Generated(
        value      =      "io.swagger.codegen.languages.
JavaJerseyServerCodegen",
        date = "2017-07-07T14:25:49.847+02:00")
public class ServiceEntry {
    @NotEmpty
    private String url;
    @NotEmpty
    private String name;
}

```

```
public String getUrl() {
    return url;
}

public void setUrl(String url) {
    this.url = url;
}

public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}

@Override
public boolean equals(java.lang.Object o) {
    if (this == o) {
        return true;
    }
    if (o == null || getClass() != o.getClass()) {
        return false;
    }
    ServiceEntry serviceEntry = (ServiceEntry) o;
    return Objects.equals(this.url, serviceEntry.url)
        && Objects.equals(
            this.name, serviceEntry.name);
}

@Override
public int hashCode() {
    return Objects.hash(url, name);
}

@Override
public String toString() {
    StringBuilder sb = new StringBuilder();
    sb.append("class ServiceEntry {\n");
    sb.append("    url: ")
        .append(toIndentedString(url)).append("\n");
    sb.append("    name: ")
        .append(toIndentedString(name)).append("\n");
    sb.append("}\n");
}
```

```
        return sb.toString();
    }

    /**
     * Convert the given object to string with each line
     * indented by 4 spaces
     * (except the first line).
     */
    private String toIndentedString(java.lang.Object o) {
        if (o == null) {
            return "null";
        }
        return o.toString().replace("\n", "\n    ");
    }
}
```

Listing of the Integration Test for the ServiceEntry Class with Jackson

```
public class ServiceEntryIntegrationTest {
    private static final ObjectMapper MAPPER =
        Jackson.newObjectMapper();
    private final ServiceEntry serviceEntry =
        new ServiceEntry();

    @Before
    public void setupExampleServiceEntries() {
        serviceEntry.setName("Swaggerize");
        serviceEntry.setUrl(
            "http://localhost:8080/swagger.json");
    }

    @Test
    public void serializeToJson() throws Exception {
        final String expectedServiceEntry =
            MAPPER.writeValueAsString(
                MAPPER.readValue(
                    fixture("fixtures/ServiceEntry.json"),
                    ServiceEntry.class));
        assertThat(MAPPER.writeValueAsString(serviceEntry))
            .isEqualTo(expectedServiceEntry);
    }

    @Test
    public void deserializeFromJson() throws Exception {
```

```
        assertEquals(
            MAPPER.readValue(
                fixture("fixtures/ServiceEntry.json"),
                ServiceEntry.class)
            .isEqualTo(serviceEntry);
    }
}
```

If you execute `mvn clean test`, then all tests should pass (including those that were added as part of Exercise 1). The integration test above is following the advice from the Dropwizard Testing guide (available at www.dropwizard.io/1.1.4/docs/manual/testing.html). However, this isn't quite appropriate, as serialization assumes that deserialization works properly (see Exercise 2 as a better alternative). The `@NotNull` annotation is part of the Bean Validation API and isn't handled by Jackson. We will revisit this later (see also Exercise 3).

Implementing the Database Layer The next job is the realization of the data handling part with the database. Dropwizard integrates Hibernate, which is a powerful object-relational mapping (ORM) framework. Our first step is to add the corresponding Dropwizard module to our build file:

```
<dependency>
    <groupId>io.dropwizard</groupId>
    <artifactId>dropwizard-hibernate</artifactId>
</dependency>
```

There are two principal components pertaining to persistency: the database model class and the data access object (DAO) class. The former is the template upon which Hibernate may produce the necessary data definition language (DDL) statements. It is annotated with Java Persistence API (JPA) annotations. This allows us to create the necessary tables and indexes inside the target database automatically on startup (of course, this step is skipped if they already exist). The target schema for these DDL instructions is database dependent (for MariaDB and MySQL, this can be provided in the Java DataBase Connectivity (JDBC) connection string). At any rate, using the JPA-annotated model class makes our service portable across a broad range of relational databases (depending on the availability of a suitable driver as well as Hibernate support). Most major databases are fully supported by Hibernate.

The DAO class is the primary entry point toward the database regarding structured query language (SQL) commands. It contains typical create, read, update, and delete (CRUD) methods. Each model class should be backed by the matching DAO class. Again, we will use the JPA API to formulate our queries and requests. All in all, the whole database layer on our side will not contain a single DDL nor SQL statement.

The choice to utilize Hibernate is directly dictated by our architecture and the desire to achieve portability. Otherwise, we could have fixed our solution toward a

concrete database (like Oracle) and use Oracle-specific DDL and SQL via the corresponding JDBC driver.

The ServiceEntry class encompasses the data about a service. This means that we should save the ServiceEntry instance as a database row in the Services table. The primary key will be the unique service name. If this wouldn't be a good candidate, then we would need to devise an artificial identification, probably by using a sequencer of some sort. Here is the listing of the ServiceEntry class again but embellished with JPA annotations (we will expand our model in later stages, but for now this is all that we need):

Partial Listing of the ServiceEntry Class (The Unchanged Details Are Omitted)

```
@Table(name = "SERVICES",
        indexes = {
            @Index(name = "IDX_SERVICES_PK", columnList = "Name")
        })
@Entity
public class ServiceEntry {
    @Column(name = "URL", unique = false, nullable = false)
    @NotEmpty
    private String url;

    @Id
    @Column(name = "Name", unique = true, nullable = false)
    @NotEmpty
    private String name;
```

Next is the listing of our DAO for the ServiceEntry model class. The code is well commented, as it is central for our service:

Listing of the ServiceEntryDAO Class

```
/**
 * This is a Data Access Object (DAO), that contains business
 * methods to manage service entries in the database.
 *
 * @author Ervin Varga
 * @since 1.0
 * @see ServiceEntry
 */
public class ServiceEntryDAO
        extends AbstractDAO<ServiceEntry> {
    private static final String CLASS_NAME =
```

```
        ServiceEntryDAO.class.getSimpleName() ;  
private static final Logger LOG =  
    Logger.getLogger(ServiceEntryDAO.class.getName()) ;  
  
/**  
 * Creates an instance of this class and associates it  
 * with the given session factory. Operations are  
 * executed inside sessions.  
 *  
 * @param sessionFactory the session factory that is used  
 * by the parent class to create proper sessions.  
 */  
public ServiceEntryDAO(SessionFactory sessionFactory) {  
    super(sessionFactory);  
}  
  
/**  
 * Finds the matching service record.  
 *  
 * @param name the unique name of the service. This is  
 * used as the key to locate the service entry.  
 * @return the requested service entry instance.  
 * @throws NullPointerException if the name is  
 * {@code null}.  
 * @throws IllegalArgumentException if a record  
 * with the given name doesn't exist.  
 */  
public ServiceEntry findByName(String name) {  
    Objects.requireNonNull(name);  
    LOG.entering(CLASS_NAME, "findByName",  
        new Object[] { name });  
  
    final ServiceEntry serviceEntry = get(name);  
    if (serviceEntry != null) {  
        if (LOG.isLoggable(Level.FINE)) {  
            LOG.fine("Found: " + serviceEntry);  
        }  
        LOG.exiting(CLASS_NAME, "findByName",  
            new Object[] { name });  
        return serviceEntry;  
    } else {  
        throw new IllegalArgumentException(  
            "Bad service identifier");  
    }  
}
```

```
/**  
 * Deletes a service entry from the database.  
 *  
 * @param name the unique name of the service. This is  
 * used as the key to locate the service entry.  
 * @throws NullPointerException if the name is  
 * {@code null}.  
 */  
public void deleteByName(String name) {  
    Objects.requireNonNull(name);  
    LOG.entering(CLASS_NAME, "deleteByName",  
        new Object[] { name });  
  
    final ServiceEntry serviceEntry = get(name);  
    if (serviceEntry != null) {  
        currentSession().delete(serviceEntry);  
  
        if (LOG.isLoggable(Level.FINE)) {  
            LOG.fine("Deleted: " + serviceEntry);  
        }  
    }  
    LOG.exiting(CLASS_NAME, "deleteByName",  
        new Object[] { name });  
}  
  
/**  
 * Creates a new service entry or updates an existing  
 * service entry in the database.  
 *  
 * @param name the unique name of the service. This is  
 * used as the key to locate the service entry.  
 * @param sdUrl the service descriptor URL.  
 * @return the created or updated service entry instance.  
 * @throws NullPointerException if any input is  
 * {@code null}.  
 */  
public ServiceEntry createOrUpdateEntry(  
    String name, ServiceDescriptorURL sdUrl) {  
    Objects.requireNonNull(name);  
    Objects.requireNonNull(sdUrl);  
    LOG.entering(CLASS_NAME, "createOrUpdateEntry",  
        new Object[] { name, sdUrl });  
  
    ServiceEntry serviceEntry = get(name);  
    if (serviceEntry != null) {
```

```
        serviceEntry.setUrl(sdUrl.getUrl());
        currentSession().update(serviceEntry);
    } else {
        serviceEntry = new ServiceEntry();
        serviceEntry.setName(name);
        serviceEntry.setUrl(sdUrl.getUrl());
        persist(serviceEntry);
    }

    if (LOG.isLoggable(Level.FINE)) {
        LOG.fine("Created/updated: " + serviceEntry);
    }
    LOG.exiting(CLASS_NAME, "createOrUpdateEntry",
                new Object[] { name, serviceEntry });
    return serviceEntry;
}

/**
 * Retrieves all service entries from the database.
 *
 * @return the service entries packed inside the
 * Services entity.
 * @see Services
 */
public Services findAll() {
    LOG.entering(CLASS_NAME, "findAll");

    final CriteriaQuery<ServiceEntry> criteria =
        constructCriteria();
    final Services services = new Services();
    services.addAll(
        currentSession()
            .createQuery(criteria)
            .getResultList());

    if (LOG.isLoggable(Level.FINE)) {
        LOG.fine(
            "Found: " + services.size() + " services.");
    }
    LOG.exiting(CLASS_NAME, "findAll",
                new Object[] { services });
    return services;
}
```

```
/*
 * This method is used by tests only to delete all data
 * from the database.
 */
void eraseData() {
    // Clear the cache.
    currentSession().clear();

    // Delete one by one, since Hibernate doesn't cascade
    // over dependent tables.
    final CriteriaQuery<ServiceEntry> criteria =
        constructCriteria();
    final Services services = new Services();
    services.addAll(
        currentSession()
            .createQuery(criteria)
            .getResultList());
    for (ServiceEntry service : services) {
        currentSession().delete(service);
    }
}

// Builds the proper JPA criteria for selecting all
// records.
private CriteriaQuery<ServiceEntry> constructCriteria() {
    final CriteriaBuilder cb =
        currentSession().getCriteriaBuilder();
    final CriteriaQuery<ServiceEntry> criteria =
        cb.createQuery(ServiceEntry.class);
    final Root<ServiceEntry> serviceEntryRoot =
        criteria.from(ServiceEntry.class);

    criteria.select(serviceEntryRoot);
    return criteria;
}
}
```

Most of the functionality is inherited from the `AbstractDAO` base class that is part of the Dropwizard Hibernate module. The `constructCriteria` method encapsulates the criteria-building procedure with JPA. We will expand this method in later stages using filtering parameters. The `eraseData` is a handy utility method to clear the test database. Notice the log statements and why it is useful to redefine the `toString` method (it was automatically generated for us by the Swagger code generator). Here is the corresponding integration test that uses the test double toward the database layer:

Listing of the ServiceEntryDAOIntegrationTest Class

```
public class ServiceEntryDAOIntegrationTest {  
    @ClassRule  
    public static final DAOTestRule DATABASE =  
        DAOTestRule  
            .newBuilder()  
            .setShowSql(true)  
            .addEntityClass(ServiceEntry.class)  
            .build();  
  
    private static ServiceEntryDAO serviceEntryDAO;  
  
    @BeforeClass  
    public static void setupServiceEntryDAO() {  
        serviceEntryDAO = new ServiceEntryDAO(  
            DATABASE.getSessionFactory());  
    }  
  
    @After  
    public void cleanupDatabase() {  
        DATABASE.inTransaction(() -> {  
            serviceEntryDAO.eraseData();  
        });  
    }  
  
    private ServiceDescriptorURL createSampleSDUrl(  
        String url) {  
        final ServiceDescriptorURL sdUrl =  
            new ServiceDescriptorURL();  
        sdUrl.setUrl(url);  
        return sdUrl;  
    }  
  
    @Test  
    public void createServiceEntry() {  
        final ServiceDescriptorURL sdUrl = createSampleSDUrl(  
            "http://localhost:8080/swagger.json");  
        final ServiceEntry savedServiceEntry =  
            DATABASE.inTransaction(() -> {  
                return serviceEntryDAO.createOrUpdateEntry(  
                    "Swaggerize1", sdUrl);  
            });  
    };
```

```
        assertEquals(savedServiceEntry);
        assertEquals(
            "Swaggerizer1",
            savedServiceEntry.getName());
    assertEquals(
        sdUrl.getUrl(),
        savedServiceEntry.getUrl());
}

@Test
public void updateServiceEntry() {
    createServiceEntry();
    final ServiceDescriptorURL sdUrl = createSampleSDUrl(
        "http://localhost:8081/admin");
    final ServiceEntry updatedServiceEntry =
        DATABASE.inTransaction(() -> {
            return serviceEntryDAO.createOrUpdateEntry(
                "Swaggerizer1", sdUrl);
        });
    assertNotNull(updatedServiceEntry);
    assertEquals(
        "Swaggerizer1",
        updatedServiceEntry.getName());
    assertEquals(
        sdUrl.getUrl(),
        updatedServiceEntry.getUrl());
}

@Test
public void createServiceEntryAndReadItOut() {
    createServiceEntry();
    final Services services =
        DATABASE.inTransaction(() -> {
            return serviceEntryDAO.findAll();
        });

    assertNotNull(services);
    assertEquals(services.size(), 1);
    assertEquals(
        services.get(0).getName(),
        "Swaggerizer1");
}

@Test
public void createServiceEntryDeleteItTryToReadItOut() {
```

```
        createServiceEntry();
        DATABASE.inTransaction(() -> {
            serviceEntryDAO.deleteByName("Swaggerize1");
        });
        final Services services =
            DATABASE.inTransaction(() -> {
                return serviceEntryDAO.findAll();
            });
    }

    assertNotNull(services);
    assertTrue(services.isEmpty());
}

@Test
public void createServiceEntryAndFindItByName() {
    createServiceEntry();
    final ServiceEntry serviceEntry =
        DATABASE.inTransaction(() -> {
            return serviceEntryDAO.findByName("Swaggerize1");
        });
    assertNotNull(serviceEntry);
    assertEquals(
        "Swaggerize1",
        serviceEntry.getName());
}

// This test ensures that trying to delete nonexistent
// records will not result in an exception.
@Test
public void tryToDeleteNonexistentServiceEntries() {
    DATABASE.inTransaction(() -> {
        serviceEntryDAO.deleteByName("Dummy");
    });
}

@Test(expected = NullPointerException.class)
public void tryToDeleteWithNullName() {
    DATABASE.inTransaction(() -> {
        serviceEntryDAO.deleteByName(null);
    });
}

@Test(expected = NullPointerException.class)
public void tryToUpdateWithNullName() {
```

```

        DATABASE.inTransaction(() -> {
            serviceEntryDAO.createOrUpdateEntry(
                null, createSampleSDUrl(""));
        });
    }

    @Test(expected = NullPointerException.class)
    public void tryToCreateOrUpdateWithNullSDUrl() {
        DATABASE.inTransaction(() -> {
            serviceEntryDAO.createOrUpdateEntry(
                "Swaggerize", null);
        });
    }

    @Test(expected = NullPointerException.class)
    public void tryToFindWithNullName() {
        DATABASE.inTransaction(() -> {
            serviceEntryDAO.findByName(null);
        });
    }

    @Test(expected = IllegalArgumentException.class)
    public void tryToFindWithWrongName() {
        DATABASE.inTransaction(() -> {
            serviceEntryDAO.findByName("dummy");
        });
    }
}

```

In order to execute this test, we need to add the following dependency inside our build file:

```

<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>test</scope>
</dependency>

```

The H2 in-memory database (visit www.h2database.com) is automatically started up in the background by the Dropwizard Testing module. This is performed by that special DAO`TestRule` class. It is useful, as we have done above, to turn on the verbose SQL mode with `setShowSql(true)`. You will then be able to trace the execution of the Hibernate generated SQL statements. The rest of our test suite just calls the various methods of the `ServiceEntryDAO` instance. Note that a couple of tests reuse the logic to create a service record. This is an exception to the

rule of test independence. For example, to test if the update works properly, you must have an ability to first create a record. All in all, we have just focused our attention on the stuff that is important for us. All the other boilerplate scaffolding was prepared by the Dropwizard framework. At this point, we can be confident that our database handling code is completed.

Implementing the API Endpoints According to our module decomposition view in Fig. 6.4, we have a dependency on the Swagger UI. We should follow the instructions on the Swagger UI Web page (see swagger.io/docs/swagger-tools/#swagger-ui-documentation) to incorporate the full distribution. We will not immediately expose the Swagger UI; just make sure that we have all the necessary information to implement our resource class. After finishing all the steps, the /src/main/resources/assets/swagger-ui project folder will contain all the files required by the Swagger UI engine (instead of dist, we will use swagger-ui). There are two tiny customizations that we will make in the index.xml file, as shown below in bolded font:

```
// Build a system
const ui = SwaggerUIBundle({
  url: "http://localhost:8080/swaggerize/spec/swagger.json",
  validatorUrl: null,
```

The url property now points to the service descriptor of our locally running service. This default value is anyhow going to be changed by our service on every request, so we could have left it as is, though. The validatorUrl should be set to null to avoid Swagger UI from validating each descriptor using an online validator. Companies would probably like to considerably change the look and feel of the UI. In this case, a full build of the Swagger UI is required that will create a new distribution. This is partly the reason that the prepackaged Swagger UI Docker image isn't always the best choice. Of course, if you want to use it as is, then you might consider creating a Docker deployment stack (containers will be the topic of our last chapter). Here, our core service would be linked to the Swagger UI container and execute in an orchestrated fashion. In both variants (embedded Swagger UI/Docker deployment stack), the result is the same. At any rate, the embedded Swagger UI is ready to receive requests of the following form:

```
<host>/swaggerize/spec/swagger-ui/index.html?url=<descriptor
location>
```

The endpoints are implemented by resource classes. In our case, we will have only one such class named as ServicesApi (responsible for the /services family of relative paths). This class, among many others, is autogenerated by the Swagger code generator. We will only reuse a small part from what has been produced. Please, recall that Dropwizard isn't supported as a target by the code

generator. The generated code is a generic JAX-RS service that doesn't fit nicely into our architecture. Here is the listing of this class:

Listing of the ServicesApi Resource Class

```
@Path("/services")
@Consumes({ "application/json" })
@Produces({ "application/json" })
@javax.annotation.Generated(
        value      =      "io.swagger.codegen.languages.
JavaJerseyServerCodegen",
        date = "2017-09-05T04:23:07.433+02:00")
public class ServicesApi {
    private static final String CLASS_NAME =
        ServicesApi.class.getSimpleName();
    private static final Logger LOG =
        Logger.getLogger(ServicesApi.class.getName());
    private static final String SWAGGER_UI_ROOT =
        "/spec/swagger-ui/index.html";
    private final ServiceEntryDAO serviceEntryDAO;

    public ServicesApi(ServiceEntryDAO serviceEntryDAO) {
        this.serviceEntryDAO = serviceEntryDAO;
    }

    @POST
    @Path("/{serviceName}/access")
    @UnitOfWork
    public Response accessService(
        @PathParam("serviceName")
        @NotEmpty
        String serviceName) {
        LOG.entering(CLASS_NAME, "accessService",
            new Object[] { serviceName });
        Response response = null;
        try {
            ServiceEntry serviceEntry =
                serviceEntryDAO.findByName(serviceName);
            assert serviceEntry != null;
            response =
                Response
                    .seeOther(
                        UriBuilder
                            .fromPath(SWAGGER_UI_ROOT)
                            .queryParam(
```

```
        "url", serviceEntry.getUrl())
        .build())
    .build();
} catch (IllegalArgumentException e) {
    throw new ApiException(
        ApiException.ErrorDescriptor.INVALID_ID, e);
} catch (Exception e) {
    throw new ApiException(
        ApiException.ErrorDescriptor.INTERNAL_ERROR, e);
}
LOG.exiting(CLASS_NAME, "accessService", response);
return response;
}

@DELETE
@Path("/{serviceName}")
@UnitOfWork
public Response deleteService(
    @PathParam("serviceName")
    @NotEmpty
    String serviceName) {
LOG.entering(CLASS_NAME, "deleteService",
    new Object[] { serviceName });
Response response = null;
try {
    serviceEntryDAO.deleteByName(serviceName);
    response = Response.noContent().build();
} catch (Exception e) {
    throw new ApiException(
        ApiException.ErrorDescriptor.INTERNAL_ERROR, e);
}
LOG.exiting(CLASS_NAME, "deleteService", response);
return response;
}

@GET
@UnitOfWork
@CacheControl(noCache = true)
public Response getServices() {
    LOG.entering(CLASS_NAME, "getServices");
    Response response = null;
    try {
        Services services = serviceEntryDAO.findAll();
        response =
            Response.ok().entity(services).build();
    }
}
```

```

        } catch (Exception e) {
            throw new ApiException(
                ApiException.ErrorDescriptor.INTERNAL_ERROR, e);
        }
    LOG.exiting(CLASS_NAME, "getServices", response);
    return response;
}

@PUT
@Path("/{serviceName}")
@UnitOfWork
public Response putService(
    @PathParam("serviceName")
    @NotEmpty
    String serviceName,
    @NotNull
    @Valid
    ServiceDescriptorURL payload) {
    LOG.entering(CLASS_NAME, "putServices",
        new Object[] { serviceName, payload });
    Response response = null;
    try {
        ServiceEntry serviceEntry =
            serviceEntryDAO
                .createOrUpdateEntry(serviceName, payload);
        response =
            Response
                .created(
                    UriBuilder
                        .fromPath("/services/" + serviceName)
                        .build())
                .entity(serviceEntry)
                .build();
    } catch (Exception e) {
        throw new ApiException(
            ApiException.ErrorDescriptor.INTERNAL_ERROR, e);
    }
    LOG.exiting(CLASS_NAME, "putServices", response);
    return response;
}
}

```

Every method represents an endpoint. The methods are associated with JAX-RS and validation-related annotations. The `@UnitOfWork` is the marker to tell Hibernate that the method is transactional. Each of them accesses the database via the

ServiceEntryDAO object. The accessService method redirects the client onto the Swagger UI engine. The putService sets the Location HTTP response header to the URL of the service record, and this is a standard way if you return the 201 HTTP status code. The ApiException is a custom exception with predefined application error codes and messages. These will be used by custom exception mappers to produce application-specific error reports. Here is the listing of this exception class:

Listing of the ApiException Class

```
/**  
 * Encompasses various application level exceptions,  
 * that are denoted by a unique  
 * internal error code.  
 *  
 * @author Ervin Varga  
 * @since 1.0  
 * @see com.example.swaggerize.api.ErrorReport  
 */  
@SuppressWarnings("serial")  
public final class ApiException extends RuntimeException {  
    public enum ErrorDescriptor {  
        INTERNAL_ERROR(  
            Status.INTERNAL_SERVER_ERROR.getStatusCode(),  
            1,  
            "The service has experienced an internal error"),  
        INVALID_ID(  
            Status.NOT_FOUND.getStatusCode(),  
            2,  
            "The given identifier doesn't match any service");  
  
        private static final String CODE_PREFIX =  
            "Swaggerize-";  
        private final int status;  
        private final String code;  
        private final String title;  
  
        private ErrorDescriptor(int status, int code, String title)  
        {  
            this.status = status;  
            this.code = CODE_PREFIX + code;  
            this.title = title;  
        }  
    }
```

```

        public int getStatus() {
            return status;
        }

        public String getCode() {
            return code;
        }

        public String getTitle() {
            return title;
        }
    };

    private final ErrorDescriptor errorDescriptor;

    public ApiException(ErrorDescriptor errorDescriptor) {
        this.errorDescriptor = errorDescriptor;
    }

    public ApiException(
        ErrorDescriptor errorDescriptor, Throwable cause) {
        super(cause);
        this.errorDescriptor = errorDescriptor;
    }

    public ErrorDescriptor getErrorDescriptor() {
        return errorDescriptor;
    }
}

```

Without Dropwizard, testing resource classes would be extremely hard. However, we will test our `ServicesApi` resource by using a special test rule and the `Mockito` mock framework. We will mock our `ServiceEntryDAO` class since we don't want to run the whole database in the background. Moreover, we want to verify that our DAO is properly utilized, and we want to simulate error conditions for invalid input. Next is the listing of the `ServicesApiIntegrationTest` class. Please, analyze it carefully to understand how our mocked DAO nicely interplays with the test harness prepared by the Dropwizard testing module:

Listing of the ServicesApiIntegrationTest Class

```

public class ServicesApiIntegrationTest {
    private static final ServiceEntryDAO dao =
        mock(ServiceEntryDAO.class);
    @ClassRule

```

```
public static final ResourceTestRule resources =
    ResourceTestRule.builder()
        .setRegisterDefaultExceptionMappers(false)
        .setClientConfigurator(config -> config.property(
            ClientProperties.FOLLOW_REDIRECTS, false))
        .addResource(new ServicesApi(dao))
        .build();

private Services services;
private ServiceEntry serviceEntry;
private ServiceDescriptorURL sdUrl;

@Before
public void setupMockingAndSampleServiceEntries() {
    sdUrl = new ServiceDescriptorURL();
    sdUrl.setUrl(
        "http://localhost:8080/spec/swagger.json");
    serviceEntry = new ServiceEntry();
    serviceEntry.setName("Swaggerize");
    serviceEntry.setUrl(sdUrl.getUrl());
    services = new Services();
    services.add(serviceEntry);

    // Mock findByName
    when(dao.findByName(eq("dummy")))
        .thenThrow(IllegalArgumentException.class);
    when(dao.findByName(eq(serviceEntry.getName())))
        .thenReturn(serviceEntry);
    // Mock findAll
    when(dao.findAll()).thenReturn(services);
    // Mock createOrUpdateEntry
    when(dao.createOrUpdateEntry(
        eq(serviceEntry.getName()), eq(sdUrl)))
        .thenReturn(serviceEntry);
}

@After
public void resetMockedDAO() {
    reset(dao);
}

@Test
public void getAllServices() {
    final Services response =
        resources
```

```
        .target("/services/")
        .request()
        .get(Services.class);
    assertNotNull(response);
    assertEquals(response).isEqualTo(services);
    verify(dao).findAll();
}

@Test
public void deleteServiceByName() {
    final Response response =
        resources
        .target(
            "/services/"
            + serviceEntry.getName())
        .request()
        .delete();
    assertNotNull(response);
    assertEquals(response.getStatus())
        .isEqualTo(Status.NO_CONTENT.getStatusCode());
    verify(dao).deleteByName(serviceEntry.getName());
}

@Test
public void createService() {
    final Response response =
        resources
        .target(
            "/services/"
            + serviceEntry.getName())
        .request()
        .put(Entity.json(sdUrl));
    assertNotNull(response);
    assertEquals(response.getStatus())
        .isEqualTo(Status.CREATED.getStatusCode());
    assertEquals(response.getLocation().toString())
        .endsWith("/services/" + serviceEntry.getName());
    verify(dao).createOrUpdateEntry(
        serviceEntry.getName(), sdUrl);
}

@Test
public void accessExistentService() {
    final Response response =
        resources
```

```
.target(
    "/services/"
    + serviceEntry.getName()
    + "/access")
.request()
.post(null);

assertNotNull(response);
assertThat(response.getStatus())
.isEqualTo(Status.SEE_OTHER.getStatusCode());
assertThat(response.getLocation().toString())
.endsWith("swagger.json");
verify(dao).findByName(serviceEntry.getName());
}

@Test
public void tryAccessingNonexistentService() {
    try {
        resources
            .target("/services/dummy/access")
            .request()
            .post(null);

        fail("Expected ApiException due to invalid ID.");
    } catch (ProcessingException e) {
        verify(dao).findByName("dummy");
        assertThat(
            ((ApiException) e.getCause())
            .getErrorDescriptor())
            .isEqualTo(
                ApiException.ErrorDescriptor.INVALID_ID);
    }
}
}
```

The central class here is the `ResourceTestRule` test rule. It is responsible for creating the proper environment to test resource classes. We have switched off the standard exception mappers to be able to catch our custom exception (see the `tryAccessingNonexistentService` test). Another important detail is to turn off automatic following of redirects by the Jersey client. Otherwise, the client will try to hit the Swagger UI engine, which isn't yet exposed. To run the previous test suite, we will need to add the following dependencies:

```
<dependency>
<groupId>io.dropwizard</groupId>
```

```

<artifactId>dropwizard-client</artifactId>
<scope>test</scope>
</dependency>
<dependency>
  <groupId>org.mockito</groupId>
  <artifactId>mockito-core</artifactId>
  <scope>test</scope>
</dependency>

```

You can see how our DAO is mocked inside the setup method annotated with `@Before`. The `deleteByName` doesn't need to be explicitly mentioned, as Mockito automatically handles void methods. The mocked DAO is now ready to respond to requests based upon the provided input. Each test case verifies the DAO usage by calling `verify`. Finally, every test case calls our resource class via the `ResourceTestRule` instance.

We are quite near to reaching the feature complete state. Here are the tasks that are left:

- Exposing the static Web resources (assets)
- Adding necessary configurable options inside the configuration file and providing the required support in the `SwaggerizeConfiguration` class
- Implementing the custom exception mapper
- Bootstrapping the application inside the `SwaggerizeApplication` class
- Implementing the automated system test suite

For publishing static assets, we need to first add the following dependency to our build file:

```

<dependency>
  <groupId>io.dropwizard</groupId>
  <artifactId>dropwizard-assets</artifactId>
</dependency>

```

Next, we need to add the following line into the `initialize` method of the `SwaggerizeApplication` class:

```
bootstrap.addBundle(new AssetsBundle("/assets/", "/spec"));
```

Let us now extend our configuration files. We will create a separate test configuration file and put that into the `src/test/resources/` folder. The production version is already sitting in the project's root folder (we will just need to alter it). Here is the content of the test version (for the production version, consult the accompanying source code):

Listing of the config.yml Test Configuration File

```
server:
  applicationContextPath: /
  type: simple
  connector:
    type: http
    # Setting port to zero means that Dropwizard will
    # choose an available one on the system.
    port: 0

database:
  driverClass: org.h2.Driver
  user: sa
  url: jdbc:h2:./target/test-db
  # JDBC driver specific properties used by Hibernate
  properties:
    charSet: UTF-8
    hibernate.dialect: org.hibernate.dialect.H2Dialect
    hibernate.show_sql: true
    hibernate.generate_statistics: false
    hibernate.hbm2ddl.auto: create-drop

logging:
  level: INFO
  loggers:
    com.example: TRACE
```

The service is set to drop all tables at startup and create them afterward. In production, we don't want to drop tables. The listing of our configuration class is given below (the other standard sections like `logging` and `server` don't need action on our part):

Listing of the SwaggerizeConfiguration Class

```
public class SwaggerizeConfiguration extends Configuration {
  @Valid
  @NotNull
  @JsonProperty("database")
  private DataSourceFactory database = new DataSourceFactory();

  @JsonProperty("database")
  public DataSourceFactory getDataSourceFactory() {
    return database;
  }
}
```

The next task from our list is the custom exception mapper, whose listing is presented below:

Listing of the ApiExceptionMapper Class

```
public final class ApiExceptionMapper
    implements ExceptionMapper<ApiException> {
    private static final Logger LOG = Logger.getLogger(
        ApiExceptionMapper.class.getName());

    public Response toResponse(ApiException e) {
        final Level logLevel =
            e.getErrorDescriptor() ==
                ApiException.ErrorDescriptor.INTERNAL_ERROR ?
                    Level.SEVERE :
                    Level.WARNING;
        LOG.log(
            logLevel,
            e.getErrorDescriptor().getTitle(),
            e.getCause());

        final ErrorReport report = new ErrorReport();
        report.setCode(e.getErrorDescriptor().getCode());
        report.setStatus(e.getErrorDescriptor().getStatus());
        if (e.getCause() != null
            && e.getCause().getMessage() != null) {
            report.setTitle(e.getCause().getMessage());
        } else {
            report.setTitle("N/A");
        }

        return Response
            .status(e.getErrorDescriptor().getStatus())
            .type(MediaType.APPLICATION_JSON_TYPE)
            .entity(report)
            .build();
    }
}
```

It implements the `ExceptionMapper` interface and transforms our `ApiException` instance into a JSON response (see Exercise 4). We have all the pieces ready, so all that we need now is to bundle them together. This is done inside the main application class. Here is the listing of the `SwaggerizeApplication` class:

Listing of the SwaggerizeApplication Class

```
public class SwaggerizeApplication
    extends Application<SwaggerizeConfiguration> {
    private static final String CLASS_NAME =
        SwaggerizeApplication.class.getSimpleName();
    private static final Logger LOG = Logger.getLogger(
        SwaggerizeApplication.class.getName());

    public static void main(final String[] args)
        throws Exception {
        LOG.entering(CLASS_NAME, "main", args);
        try {
            new SwaggerizeApplication().run(args);
        } catch (Exception e) {
            LOG.log(
                Level.SEVERE,
                "Fatal error during application startup",
                e);
            System.exit(-1);
        }
    }

    @Override
    public String getName() {
        return "Swaggerize";
    }

    private final
    HibernateBundle<SwaggerizeConfiguration> hibernate =
        new HibernateBundle<SwaggerizeConfiguration>(
            ServiceEntry.class) {
            @Override
            public DataSourceFactory getDataSourceFactory(
                SwaggerizeConfiguration configuration) {
                return configuration.getDataSourceFactory();
            }
        };
    };

    @Override
    public void initialize(
        final Bootstrap<SwaggerizeConfiguration> bootstrap) {
        // Enable variable substitution with environment
        // variables inside the configuration file.
        bootstrap.setConfigurationSourceProvider(
```

```

        new SubstitutingSourceProvider(
            bootstrap.getConfigurationSourceProvider(),
            new EnvironmentVariableSubstitutor(false))) ;

    bootstrap.addBundle(
        new AssetsBundle("/assets/", "/spec")) ;
bootstrap.addBundle(hibernate) ;
}

@Override
public void run(
    final SwaggerizeConfiguration configuration,
    final Environment environment) {
environment
.jersey()
.register(new ApiExceptionMapper()) ;

final ServiceEntryDAO dao = new ServiceEntryDAO(
    hibernate.getSessionFactory());
environment
.jersey()
.register(new ServicesApi(dao)) ;

LOG.fine("Swaggerize service is up and running...");
}
}
}

```

The production configuration file refers to environment variables that may be set before starting up the service. This is the reason to instantiate the SubstitutingSourceProvider class and put it as part of the bootstrapping process. Now, we need to test the whole service and ensure that all details are in place. Below is the listing of the SwaggerizeSystemTest class:

Listing of the SwaggerizeSystemTest Class

```

public class SwaggerizeSystemTest {
    @ClassRule
    public static final
    DropwizardAppRule<SwaggerizeConfiguration> RULE =
        new DropwizardAppRule<SwaggerizeConfiguration>(
            SwaggerizeApplication.class,
            ResourceHelpers.resourceFilePath("config.yml"));

    private Client client;
    private Services services;
}

```

```
private ServiceEntry serviceEntry;
private ServiceDescriptorURL sdUrl;

@Before
public void createSampleServiceAndHttpClient() {
    sdUrl = new ServiceDescriptorURL();
    sdUrl.setUrl(
        "http://localhost:8080/spec/swagger.json");
    serviceEntry = new ServiceEntry();
    serviceEntry.setName("Swaggerize");
    serviceEntry.setUrl(sdUrl.getUrl());
    services = new Services();
    services.add(serviceEntry);

    client = new JerseyClientBuilder(
        RULE
        .getEnvironment())
        .build(
            "Test Client - " + UUID.randomUUID().toString());
    // Need to be set to safe values to prevent system
    // tests to fail.
    client.property(
        ClientProperties.CONNECT_TIMEOUT, 20000);
    client.property(
        ClientProperties.READ_TIMEOUT, 20000);
}

@After
public void closeHttpClient() {
    client.close();
}

@Test
public void ensureAllResourcesAndProvidersAreRegistered()
    throws Exception {
    final ResourceConfig resourceConfig =
        RULE
        .getEnvironment()
        .jersey()
        .getResourceConfig();
    assertTrue(resourceConfig.isRegistered(
        ServicesApi.class));
    assertTrue(resourceConfig.isRegistered(
        ApiExceptionMapper.class));
}
```

```
@Test
public void accessTheOpenApiSpecification() {
    final Response response =
        client.target(
            String.format(
                "http://localhost:%d/spec/swagger.json",
                RULE.getLocalPort())))
    .request()
    .get();
    assertThat(response.getStatus())
        .isEqualTo(Status.OK.getStatusCode());
}

@Test
public void accessTheSwaggerUIEngine() {
    final Response response =
        client.target(
            String.format(
                "http://localhost:%d/spec/swagger-ui/index.html",
                RULE.getLocalPort())))
    .request()
    .get();
    assertThat(response.getStatus())
        .isEqualTo(Status.OK.getStatusCode());
}

@Test
public void createServiceAndDeleteItFollowingTheLocationLink()
{
    Response response = client.target(
        String.format(
            "http://localhost:%d/services/"
            + serviceEntry.getName()
            , RULE.getLocalPort()))
        .request()
        .put(Entity.json(sdUrl));
    assertThat(response.getStatus())
        .isEqualTo(Status.CREATED.getStatusCode());

    final URI followLink = response.getLocation();
    response = client.target(
        String.format(
            "http://localhost:%d/"
            + followLink.getPath(),
            RULE.getLocalPort()))
}
```

```
        .request()
        .delete();
    assertThat(response.getStatus())
        .isEqualTo(Status.NO_CONTENT.getStatusCode());
}

@Test
public void createServiceAndAccessItWithSwaggerUI() {
    Response response = client.target(
        String.format(
            "http://localhost:%d/services/"
            + serviceEntry.getName()
            , RULE.getLocalPort())))
        .request()
        .put(Entity.json(sdUrl));
    assertThat(response.getStatus())
        .isEqualTo(Status.CREATED.getStatusCode());

    response = client.target(
        String.format(
            "http://localhost:%d/services/"
            + serviceEntry.getName()
            + "/access",
            RULE.getLocalPort())))
        .request()
        .post(null);
    assertThat(response.getStatus())
        .isEqualTo(Status.OK.getStatusCode());
}

@Test
public void tryCreatingServiceWithInvalidJson() {
    sdUrl.setUrl(null);
    Response response = client.target(
        String.format(
            "http://localhost:%d/services/"
            + serviceEntry.getName()
            , RULE.getLocalPort())))
        .request()
        .put(Entity.json(sdUrl));
    assertThat(response.getStatus()).isEqualTo(422);
}

@Test
public void tryAccessingANonexistentService() {
```

```

        Response response = client.target(
            String.format(
                "http://localhost:%d/services/"
                + "dummy/access"
                , RULE.getLocalPort()))
            .request()
            .post(null);
        assertThat(response.getStatus())
            .isEqualTo(Status.NOT_FOUND.getStatusCode());
    }
}
}

```

The system is started up in a controlled fashion by the `DropwizardAppRule` class. We register our application class and refer to the test configuration located is inside the `resources` folder. In the test setup method, we create our HTTP client. Each test case uses this client to send the proper HTTP request. The most interesting tests are those that connect different scenarios, for example, creating a service record and using the provided link inside the `Location` header to immediately delete it. Another one is to create a service record and access the registered service via the Swagger UI engine by following the redirect. We just need to change the version number of our service to `1.0.0`. If you invoke `mvn clean package` from the project's root folder, then it should produce the `1.0.0` version of the service's jar file. All in all, we are almost done with the initial version (see also Exercises 5 and 6).

Implementing the Security Requirements Security is often treated as an after-thought of development, i.e., something that is added after the functional part is done. This is a very misguided view. Security must be incorporated from the very beginning and reflected in the architecture. This is the reason why we had included security considerations into our Resource API. Furthermore, the chosen technology, Dropwizard in our case, was evaluated against our architecturally significant requirements. Among them was the requirement to have some endpoints protected by the HTTP Basic authentication/authorization mechanism. Dropwizard has a built-in support for this. Therefore, we were content to leave the finalization of our security aspects to the end. After all, Dropwizard comprises most of our architecture, so we had security in front of us from the very beginning.

We will merely need to follow the guidance from the Dropwizard Authentication section (visit www.dropwizard.io/1.1.4/docs/manual/auth.html). Our first step is to include the following dependency into our build file:

```

<dependency>
    <groupId>io.dropwizard</groupId>
    <artifactId>dropwizard-auth</artifactId>
</dependency>

```

We don't use roles, so every authenticated principal gains all possible rights. Thus, we will just mark the methods inside our `ServicesApi` resource class that creates/updates and deletes the service records with `@PermitAll`. We will introduce for simplicity a configuration parameter `clientPassword`. In the test configuration, this will be a hard-coded value, but the production version will read out the `CLIENT_PASSWORD` environment variable (you should store credentials in some safe encrypted store). Our simple authenticator is presented below, together with the small class representing our principal:

Listing of the SwaggerizeAuthenticator Class

```
public final class SwaggerizeAuthenticator
    implements Authenticator<BasicCredentials, User> {
    private static final Logger LOG = Logger.getLogger(
        SwaggerizeAuthenticator.class.getName());
    private final String clientPassword;

    public SwaggerizeAuthenticator(String clientPassword) {
        Objects.requireNonNull(clientPassword);
        this.clientPassword = clientPassword;
    }

    @Override
    public Optional<User> authenticate(BasicCredentials
credentials)
        throws AuthenticationException {
        LOG.entering(
            SwaggerizeAuthenticator
                .class.getSimpleName(),
            "authenticate",
            new Object[] { credentials });
        if (clientPassword.equals(credentials.getPassword())) {
            LOG.finer(
                credentials.getUsername()
                + " is logged in.");
            return Optional.of(
                new User(credentials.getUsername()));
        }
        LOG.warning(
            "Login attempt has failed: "
            + credentials.getUsername());
        return Optional.empty();
    }
}
```

Listing of the User Class

```
public final class User implements Principal {
    private final String name;

    public User(String name) {
        this.name = name;
    }

    @Override
    public String getName() {
        return name;
    }
}
```

The last missing piece is the following snippet that must be inserted into the `run` method of our application class:

```
environment.jersey().register(new AuthDynamicFeature(
    new BasicCredentialAuthFilter.Builder<User>()
        .setAuthenticator(new SwaggerizeAuthenticator(
            configuration.getClientPassword()))
        .setRealm("Swaggerize Realm")
        .setPrefix("Basic")
        .buildAuthFilter()));
```

You're probably guessing that what comes next is an automated test suite to test our security infrastructure. We must first add the following dependency to our build file:

```
<dependency>
    <groupId>org.glassfish.jersey.test-framework.providers</
groupId>
    <artifactId>jersey-test-framework-provider-grizzly2</
artifactId>
    <scope>test</scope>
    <exclusions>
        <exclusion>
            <groupId>javax.servlet</groupId>
            <artifactId>javax.servlet-api</artifactId>
        </exclusion>
        <exclusion>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

```
</exclusions>
</dependency>
```

Below is the listing of the security test. We first access the two protected methods with the correct security credentials. Afterward, we try to access once without providing any data and once giving wrong data:

Listing of the ServicesApiSecurityTest

```
public class ServicesApiSecurityTest {
    private static final String CLIENT_PASSWORD =
        "testAdmin";
    private static final String AUTH_HEADER =
        "Basic VGVzdDp0ZXN0QWRtaW4=";
    private static final String AUTH_HEADER_WRONG =
        "Basic VGVzdDp0ZXN0";
    private static final ServiceEntryDAO dao =
        mock(ServiceEntryDAO.class);
    @ClassRule
    public static final ResourceTestRule resources =
        ResourceTestRule.builder()
            .setTestContainerFactory(
                new GrizzlyWebTestContainerFactory())
            .addProvider(new AuthDynamicFeature(
                new BasicCredentialAuthFilter.Builder<User>()
                    .setAuthenticator(
                        new SwaggerizeAuthenticator(
                            CLIENT_PASSWORD))
                    .setRealm("Swaggerize Test Realm")
                    .setPrefix("Basic")
                    .buildAuthFilter())))
            .addResource(new ServicesApi(dao))
            .build();

    private Services services;
    private ServiceEntry serviceEntry;
    private ServiceDescriptorURL sdUrl;

    @Before
    public void setupMockingAndSampleServiceEntries() {
        sdUrl = new ServiceDescriptorURL();
        sdUrl.setUrl("http://localhost:8080/spec/swagger.json");
        serviceEntry = new ServiceEntry();
        serviceEntry.setName("Swaggerize");
        serviceEntry.setUrl(sdUrl.getUrl());
        services = new Services();
        services.add(serviceEntry);
```

```
// Mock findByName
when(dao.findByName(eq(serviceEntry.getName())))
    .thenReturn(serviceEntry);
// Mock createOrUpdateEntry
when(dao.createOrUpdateEntry(
    eq(serviceEntry.getName()), eq(sdUrl)))
    .thenReturn(serviceEntry);
}

{@After
public void resetMockedDAO() {
    reset(dao);
}

{@Test
public void deleteServiceByNameAuthenticated() {
    final Response response =
        resources
            .target(
                "/services/"
                    + serviceEntry.getName())
            .request()
            .header("Authorization", AUTH_HEADER)
            .delete();
    assertNotNull(response);
    assertThat(response.getStatus())
        .isEqualTo(Status.NO_CONTENT.getStatusCode());
}

{@Test
public void deleteServiceByNameNonAuthenticated() {
    final Response response =
        resources
            .target(
                "/services/"
                    + serviceEntry.getName())
            .request()
            .delete();
    assertNotNull(response);
    assertThat(response.getStatus())
        .isEqualTo(Status.UNAUTHORIZED.getStatusCode());
}

{@Test
public void createServiceAuthenticated() {
```

```
final Response response =
    resources
    .target(
        "/services/"
        + serviceEntry.getName())
    .request()
    .header("Authorization", AUTH_HEADER)
    .put(Entity.json(sdUrl));
assertNotNull(response);
assertThat(response.getStatus())
    .isEqualTo(Status.CREATED.getStatusCode());
}

@Test
public void createServiceNonAuthenticated() {
    final Response response =
        resources
        .target(
            "/services/"
            + serviceEntry.getName())
        .request()
        .header("Authorization", AUTH_HEADER_WRONG)
        .put(Entity.json(sdUrl));
    assertNotNull(response);
    assertThat(response.getStatus())
        .isEqualTo(Status.UNAUTHORIZED.getStatusCode());
}
}
```

The test resource rule is set up with the Grizzly Web container, and we install the same security provider as in the main application class. The rest of the test suite is intuitive. We will also need to update our system test, as it is missing the same authorization header (see the accompanying source code of this book for details).

Manually Testing the Service There is only one extra test that we should perform, and that is the manual exploratory test. Even though we have managed to automate all sorts of tests, there is still a need to do some manual testing. Usually, you want to ensure that the software will not do something it isn't supposed to do. Moreover, you probably want to inspect the look and feel of the Swagger UI engine, and this is impractical to automate.

The production configuration file assumes a running MySQL server in the background.⁸ It also references the MySQL JDBC driver. Hence, we must put that driver inside our code base, so the build file must contain the next dependency:

⁸You may skip this step if you intend to reconfigure the database settings.

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>6.0.6</version>
</dependency>
```

For the sake of completeness, I will show you briefly how to set up the MySQL server so that you may repeat the upcoming shell commands in this section. To make the discussion more interesting, we will use an Amazon RDS service and choose MySQL as the engine. You should just follow the Amazon Web Services (AWS) wizard (see more details at <https://aws.amazon.com/getting-started/tutorials/create-mysql-db/> including how to install the MySQL client) and make sure to select the Dev/Test variant, choose the minimal hardware capacity, and keep all the other settings on default. Of course, you may decide what should be the master username, password, and schema name (the name of the database that you will provide on the AWS console). Eventually, you should see a screen like the one shown in Fig. 6.6. We will revisit the automation of this process in the last chapter of this book.

From the project's root folder, execute mvn clean package. This will create the 1.0.0 version of the jar file inside the target subfolder. Alter the values of the environment variables below (I'm using Mac OS X Yosemite), and start the service:

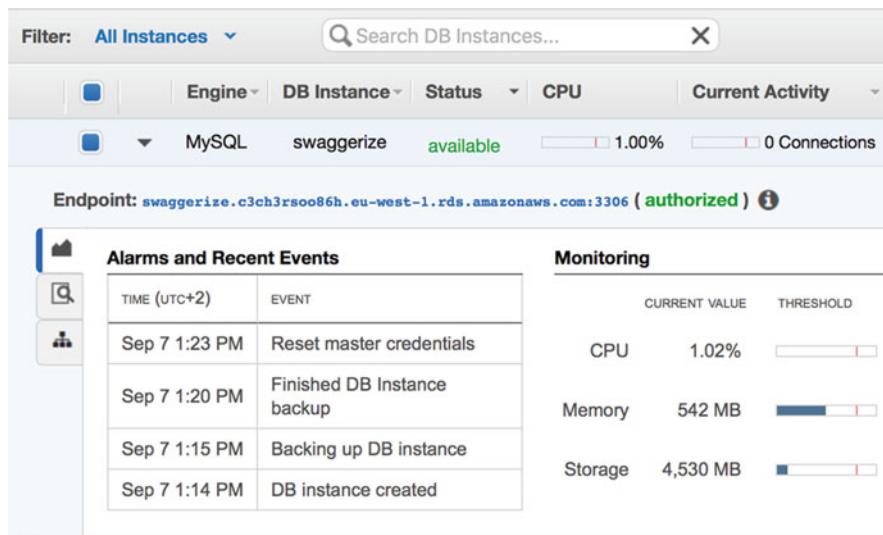


Fig. 6.6 The Web page of the Amazon Web Services console showing the status of our database. You should note the Endpoint URL, as this is required to configure the JDBC connection string. It is possible, as depicted here, to alter the initial settings. For example, I've changed the master password, as indicated in the Events log

```
export CLIENT_PASSWORD='<your password for the service>'; \
export DB_USERNAME='<master username>'; \
export DB_PASSWORD='<master password>'; \
export DB_HOST='<AWS endpoint, host part>'; \
export DB_PORT='<AWS endpoint, port part>'; \
export DB_SCHEMA='<your schema name>'; \
java -server -jar target/swaggerize-1.0.0.jar server \
config.yml
```

Your MySQL client (I'm using MySQL Workbench 6.3) should show the SERVICES table inside your schema. We are now ready to execute some commands. For this, you will need to open another terminal window. I suggest you monitor inside your SQL client what is happening in the database while you run the next commands that modify the data. Let us check the health of our service (see Exercise 7), as documented in the project's README.md file:

```
curl http://localhost:8081/healthcheck
{
  "deadlocks": {"healthy": true},
  "hibernate": {"healthy": true}
}
```

Let us register our own service. If you have changed the default client password, then you will need to generate a new authorization header value (you may do that at www.blitter.se/utils/basic-authentication-header-generator/):

```
curl -v -X PUT -H "Content-Type: application/json" \
-H "Authorization: Basic YWRtaW46dGVzdEFkbWlu" \
-d '{
  "url": "http://localhost:8080/swaggerize/spec/swagger.json"
}' \
http://localhost:8080/swaggerize/services/swaggerize

* Trying ::1...
* Connected to localhost (::1) port 8080 (#0)
> PUT /api/swaggerize/services/swaggerize HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.43.0
> Accept: */*
> Content-Type: application/json
> Authorization: Basic YWRtaW46dGVzdEFkbWlu
> Content-Length: 49
>
* upload completely sent off: 49 out of 49 bytes
< HTTP/1.1 201 Created
< Date: Thu, 07 Sep 2017 13:38:25 GMT
< Location: http://localhost:8080/services/swaggerize
```

```
< Content-Type: application/json
< Content-Length: 69
<
* Connection #0 to host localhost left intact
{"url":"http://localhost:8080/api/swaggerize/spec/swagger.
json","name":"swaggerize"}
```

Oops! We have a major issue in our service. Please, notice that the / swaggerize application context is missing in the highlighted line. How can this happen after all that battery of automated tests?! Well, the test configuration was set up to use/as the root, which is also the default. We must go back and fix this error (see Exercise 8). This is a fine proof why manual testing is always beneficial, and it complements the automated tests. Do remember that any undetected defect may turn into a failure during production usage. It is more stressful and annoying to fix production errors than to spend more time for heightening quality.

The fix is a removal of two characters from the code base, without counting the modifications of tests! What can be simpler than that? Of course, the deletion of a single character, but that wouldn't solve our problem. Recall the Agile Manifesto's (visit agilemanifesto.org) principle about simplicity:

Simplicity—the art of maximizing the amount of work not done—is essential.

This is only viable if you have previously invested a tremendous amount of work to realize where that simplicity lurks. Exercise 8 is an excellent case to remind you about this missing piece from the previous quote. Once you know what to fix, it becomes trivial (again, just a deletion of two characters).

After the fix and repeating the previous steps, we should receive the following response when registering the service:

```
...
HTTP/1.1 201 Created
< Date: Thu, 07 Sep 2017 15:26:24 GMT
< Location: http://localhost:8080/swaggerize/services/swaggerize
< Content-Type: application/json
< Content-Length: 84
<
* Connection #0 to host localhost left intact
{"url":"http://localhost:8080/swaggerize/spec/swagger.
json","name":"swaggerize"}
```

Now, let us try to access our service via the Swagger UI engine using curl and see what happens (type the next command in one line):

```
curl -v -X POST http://localhost:8080/swaggerize/services/
swaggerize/access

...
< HTTP/1.1 303 See Other
< Date: Thu, 07 Sep 2017 15:55:08 GMT
< Location: http://localhost:8080/swaggerize/spec/swagger-ui/
index.html?url=http%3A%2Flocalhost%3A8080%2Fapi%2Fswaggerize%
2Fspec%2Fswagger.json
< Content-Length: 0
```

We see here that the `Location` response header is properly pointing to the entry point of the Swagger UI facility with a link to the service descriptor of our service (those strange characters are part of the URL encoding). If you now enter that URL into a browser, then it will show the dynamic HTML rendering our API specification, as depicted in Fig. 6.7. All in all, we are ready for evolution!

6.3 First Usability Enhancement: Problem

Soon after the first release, the users have reported a couple of usability issues. This shouldn't surprise us since we have already observed a couple of awkward steps in the previous session. The manual tests are perfect to experience such usability problems. First, it isn't possible to access the service via Swagger UI by hitting `localhost:8080/swaggerize/services/swaggerize/access` from the browser. It is a `POST` request instead of `GET`. Moreover, to reach the nice HTML environment, a user must go through those steps to manually register the `Swaggerize` service. These things need to be addressed. Luckily, Dropwizard has a solution that we will now leverage.

6.4 First Usability Enhancement: Resolution

Our job is comprised from two tasks: to self-register our service and to provide a convenient HTML page as a starting point for users (they would simply need to hit `localhost:8080/swaggerize`). We don't need to be fancy here since the users are the guys from the QA team who are technically adept. Therefore, this startup page will just enlist the registered services and offer a possibility to access them via the Swagger UI engine. Now, the version number will increase to `1.1.0`, as this is a compatible feature expansion of our service.

We will use Mustache as a templating engine and proceed according to the Dropwizard Views manual that has also a link to Mustache (see www.dropwizard.io/1.1.4/docs/manual/views.html). Our first step is to introduce the following dependency into our build file:

Curl

```
curl -X GET "http://localhost:8080/swaggerize/services" -H  
"accept: application/json"
```

Server response**Code Details**

200

Response body

```
[  
  {  
    "url": "http://localhost:8080/swagger-  
    ize/spec/swagger.json",  
    "name": "swaggerize"  
  }  
]
```

Response headers

```
cache-control: no-cache, no-transform  
content-length: 82  
content-type: application/json  
date: Sat, 09 Sep 2017 01:46:39 GMT  
vary: Accept-Encoding
```

Fig. 6.7 The Swagger UI engine in action for our service. The screen shows the request and response directly from the Swagger UI for listing the currently registered services

```
<dependency>  
  <groupId>io.dropwizard</groupId>  
  <artifactId>dropwizard-views-mustache</artifactId>  
</dependency>
```

The boilerplate additions (extending the configuration to include views, registering the new index page resource, etc.) may be seen in the accompanying source

code of this book. The interesting novelties are the index page resource and the Mustache template. These are shown below:

Listing of the IndexPage Resource Class

```
@Path("/")
@Produces(MediaType.TEXT_HTML)
public class IndexPage {
    private static final String CLASS_NAME =
        IndexPage.class.getSimpleName();
    private static final Logger LOG =
        Logger.getLogger(IndexPage.class.getName());
    private final ServiceEntryDAO serviceEntryDAO;

    public IndexPage(ServiceEntryDAO serviceEntryDAO) {
        this.serviceEntryDAO = serviceEntryDAO;
    }

    @GET
    @UnitOfWork
    @CacheControl(maxAge = 3, maxAgeUnit = TimeUnit.MINUTES)
    public ServicesView renderServices(@Context UriInfo uriInfo) {
        LOG.entering(CLASS_NAME, "renderServices");
        ServicesView servicesView = null;
        try {
            selfRegisterService(uriInfo);
            Services services = serviceEntryDAO.findAll();
            servicesView = new ServicesView(services);
        } catch (Exception e) {
            throw new ApiException(
                ApiException.ErrorDescriptor.INTERNAL_ERROR, e);
        }
        LOG.exiting(CLASS_NAME, "renderServices");
        return servicesView;
    }

    private void selfRegisterService(UriInfo uriInfo) {
        ServiceDescriptorURL sdUrl =
            new ServiceDescriptorURL();
        sdUrl.setUrl(
            uriInfo.getBaseUri().toString()
            + "spec/swagger.json");
        serviceEntryDAO.createOrUpdateEntry(
            SwaggerizeConfiguration.SERVICE_NAME, sdUrl);
    }
}
```

Listing of the services.mustache Mustache Template

```
<!DOCTYPE html>
<html>
  <body>
    <h1>Registered Services</h1>
    <ul>
      {{#services}}
        <li><a href="spec/swagger-ui/index.html?url={{url}}">
          {{name}}</a></li>
      {{/services}}
    </ul>
  </body>
</html>
```

The index page resource returns an HTML, so the media type is text/html (see Exercise 9). It self-registers our service at each invocation, which in some way protects it from accidental deletion. The HTML rendering is a slow process, so we want to prevent rapid refreshes. Therefore, we have that cache control instruction. The Mustache engine will also internally cache the rendered page (this is turned off during development; see the test configuration file in the source code). The `ServicesView` view class is a wrapper around the `Services` representation class (see the book's source code). The `renderServices` method accepts the `UriInfo` object that is as part of the Web execution context. This is used to get the base Uniform Resource Identifier (URI) of the service. Moreover, this exemplifies how to reach out to those extra contextual data.

The mustache template is comprised of two sections: the HTML wrapper and the looping over data items. The `{{#services}}` starts the iteration over the collection of service records and results in the call to the `services` method of the view class. The inner field accessors just read out the corresponding properties of the `ServiceEntry` object.

6.4.1 Template Engine vs. Programmatic Approach

Someone would argue that for such a simple HTML, we could have done the job from the `renderService` method itself by programmatically building up the HTML content (we would treat it as a string). They would say that this is “simpler” than to bring in a separate templating engine. Unfortunately, this is a shortsighted viewpoint.

First, it goes against the principles of Separation of Concerns and Cohesion. The first talks about separating different aspects of the solution into various domains for better maintainability and control. Obviously, the view design shouldn't be part of the core business logic. The cohesion would also be jeopardized, as our `renderService` method would perform too many unrelated tasks. We have

learned in Chap. 2 that anytime we break a principle, we must have strong arguments. In this case, there are none. The rationale that a templating engine is more complex doesn't hold. I am sure that in Java, you wouldn't be able to construct a final HTML with `StringBuilder` in profoundly fewer lines. Performance? Nothing can be said about this without measurements. Moreover, the return on investment (ROI) would be unjustifiable. Simplicity is a hard-won prize, laziness to learn something new and think are just lame excuses in the name of simplicity.

Nonetheless, the biggest impact of bringing in Mustache pertains to evolvability. The explicit exposition of the UI space enables a totally new evolutionary path. The company is now able to allocate UI designers to make the view much more powerful and esthetically appealing. Of course, you should also be very careful when opening this possibility. The view may start to generate much more change requests than other parts of the ecosystem. Therefore, you must be prepared to shape the service to address the demands regarding usability. This is a quality attribute that expresses how easy, efficient, and sophisticated the usage of the system is from the viewpoint of a client.

There is a huge misconception about structuring the system around views. The well-known Model-View-Controller (MVC) architectural/design pattern isolates the view (UI) from the model (core business logic). Such a delineation suggests that one just needs to apply this pattern, and all problems related to views are magically solved. This is a huge fallacy, as depicted in Fig. 6.8. When companies are struck by a sobering experience, that multitude of usability requirements cannot be easily satisfied, then they search for MVC variants that supposedly fix the deficiencies in the original version. None of them do since it isn't at all about a deficiency of the pattern itself. It is about the nature of the usability attribute. Our next evolutionary increment will demonstrate this in action.

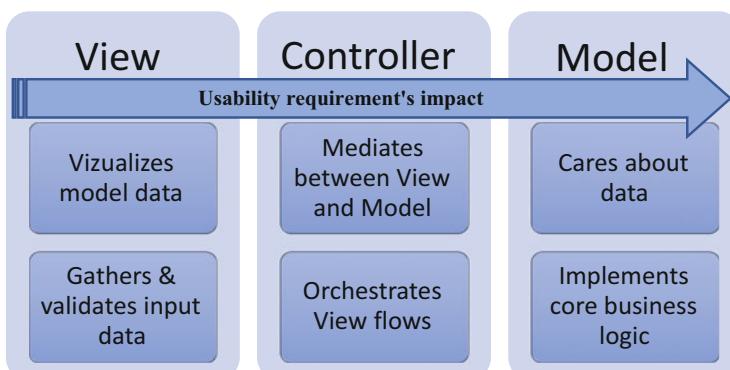


Fig. 6.8 The MVC pattern alone (or any of its variant) cannot protect you from the penetrating power of the usability wish. The whole system, rather than just the view, must be ready for motion

6.5 Second Usability Enhancement: Problem

The company has engaged its UI design team to boost the appearance and sophistication of the index page. They have managed to transform the simple list from the previous version into a full-blown single-page Web application (assume they have been given access to the repository and made all changes in a separate branch). However, two of the improvements demand support from our core service: to filter services by name and to sort the output by name in a specified order (ascending or descending). They have noticed that as the number of registered services grow, an unsorted complete list is cumbersome for finding the desired service.

6.6 Second Usability Enhancement: Resolution

This job has three tasks: extending the service descriptor with two additional query parameters for our GET /services endpoint, accepting these parameters inside the ServicesApi resource class, and extending our DAO to support filtering and sorting by name. All layers are impacted as explained in Fig. 6.8. The benefit of having separated domains is that you may parallelize the work. Another team member may work on the DAO, while you could enhance the API specification and resource class, or vice versa. Of course, we will need to write appropriate tests as well. Here is the patch to the API specification:

```
parameters:  
  - name: name  
    description: The partial or full name of service(s) to  
      include in the response.  
    in: query  
    type: string  
  - name: sort  
    description: The output ordering criteria (ascending or  
      descending).  
    in: query  
    type: string  
    enum: [asc, desc]  
    default: asc
```

Both are optional parameters, so the new version of our service will be 1.2.0. Strictly speaking, the behavior will change (the output is going to be always sorted in ascending order). However, since the previous version of our service didn't say anything about the ordering of output, we may conclude that sorting cannot ruin backward compatibility.

We will start modifying our DAO class, as we have done during development. The findAll method should be augmented with two additional parameters: the optional service name and the sort order. Eclipse IDE may help a lot in finding all

places, where the `findAll` method is called. All previous calls should be replaced from `findAll()` to `findAll(Optional.empty(), SortOrder.ASC)`. The `SortOrder` is a simple enumeration for sorting criteria (see the book's source code). Here are the updates to our `ServiceEntryDAO` class:

Listing of the Patch to the ServiceEntryDAO Class

```
/**  
 * Retrieves service entries from the database.  
 *  
 * @param name the partial or full name of service(s) to  
 * include in the response.  
 * @param sortOrder the output ordering criteria  
 * (ascending or descending).  
 * @return the service entries packed inside the  
 * Services entity.  
 * @throws NullPointerException if the sortOrder is  
 * {@code null}.  
 * @see Services  
 */  
public Services findAll(  
    Optional<String> name, SortOrder sortOrder) {  
    Objects.requireNonNull(sortOrder);  
    LOG.entering(CLASS_NAME, "findAll",  
        new Object[] { name, sortOrder });  
  
    final CriteriaQuery<ServiceEntry> criteria =  
        constructCriteria(name, sortOrder);  
    final Services services = new Services();  
    services.addAll(  
        currentSession()  
            .createQuery(criteria)  
            .getResultList());  
  
    if (LOG.isLoggable(Level.FINE)) {  
        LOG.fine("Found: " + services.size() + " services.");  
    }  
    LOG.exiting(CLASS_NAME, "findAll",  
        new Object[] { services });  
    return services;  
}  
  
// Builds the proper JPA criteria for selecting and  
// sorting records.  
private CriteriaQuery<ServiceEntry> constructCriteria(
```

```

    Optional<String> name, SortOrder sortOrder) {
final CriteriaBuilder builder =
        currentSession().getCriteriaBuilder();
final CriteriaQuery<ServiceEntry> criteria =
        builder.createQuery(ServiceEntry.class);
final Root<ServiceEntry> from =
        criteria.from(ServiceEntry.class);
final EntityType<ServiceEntry> table = from.getModel();

criteria.select(from);
if (name.isPresent()) {
    criteria.where(
        builder.like(
            from.get(
                table
                    .getSingularAttribute("name", String.class)),
            name.get() + '%'));
}
if (sortOrder == SortOrder.ASC) {
    criteria.orderBy(builder.asc(from.get("name")));
} else {
    criteria.orderBy(builder.desc(from.get("name")));
}
return criteria;
}

```

Most changes are localized inside the private `constructCriteria` method. An astute reader may have noticed that the variable names are altered. This is an example of an opportunistic refactoring technique when you improve the code base as you modify it. In this case, the renaming aids clarity since the names now better communicate their intent. The above code changes reflect the WHERE and ORDER BY SQL statements using the JPA API. It is important to add that extra % character to the search string since we are using the SQL LIKE form. At this moment, all the existent tests must pass. Of course, we need to test the new feature, and here are the matching test cases for that:

Listing of the Test Cases to Exercise Searching and Sorting

```

@Test
public void filterAndSortServicesByName() {
    createServiceEntry("Swaggerize1");
    createServiceEntry("Swaggerize2");
    createServiceEntry("Swaggerize3");
    Services services =
        DATABASE.inTransaction(() -> {

```

```
        return serviceEntryDAO.findAll(
            Optional.of("Swaggerize"),
            SortOrder.ASC);
    });

assertNotNull(services);
assertEquals("Swaggerize1", services.get(0).getName());
assertEquals("Swaggerize2", services.get(1).getName());
assertEquals("Swaggerize3", services.get(2).getName());

services = DATABASE.inTransaction(() -> {
    return serviceEntryDAO.findAll(
        Optional.of("S"), SortOrder.DESC);
});
assertNotNull(services);
assertEquals("Swaggerize3", services.get(0).getName());
assertEquals("Swaggerize2", services.get(1).getName());
assertEquals("Swaggerize1", services.get(2).getName());

services = DATABASE.inTransaction(() -> {
    return serviceEntryDAO.findAll(
        Optional.of("Swaggerize1"), SortOrder.DESC);
});
assertNotNull(services);
assertEquals(1, services.size());
assertEquals("Swaggerize1", services.get(0).getName());

services = DATABASE.inTransaction(() -> {
    return serviceEntryDAO.findAll(
        Optional.of("Dummy"), SortOrder.DESC);
});
assertNotNull(services);
assertTrue(services.isEmpty());
}

@Test(expected = NullPointerException.class)
public void tryFilterAndSortWithNullSortingCriteria() {
    DATABASE.inTransaction(() -> {
        return serviceEntryDAO.findAll(
            Optional.of("Swaggerize"), null);
    });
}
```

The previous tests cover the major combinations of input parameters. The update to our resource class becomes rather trivial once our DAO class is equipped with

searching and sorting capabilities. However, if you don't know how to process query parameters in your resource class, then you may utilize the Swagger code generator (see the beginning of this chapter). The generated code may give you hints as to what must be added to your code. Here is the patch to the ServicesApi class as well as the corresponding test cases:

Listing of the Patch to the ServicesApi Class

```
@GET
@UnitOfWork
@CacheControl(noCache = true)
public Response getServices(
    @QueryParam("name")
    String name,
    @QueryParam("sort")
    @DefaultValue("ASC")
    String sortOrder) {
    LOG.entering(CLASS_NAME, "getServices",
        new Object[] { name, sortOrder });
    Response response = null;
    try {
        Services services = serviceEntryDAO.findAll(
            Optional.ofNullable(name),
            SortOrder.valueOf(sortOrder.toUpperCase()));
        response = Response.ok().entity(services).build();
    } catch (IllegalArgumentException e) {
        throw new ApiException(
            ApiException.ErrorDescriptor.INVALID_SORT_ORDER);
    } catch (Exception e) {
        throw new ApiException(
            ApiException.ErrorDescriptor.INTERNAL_ERROR, e);
    }
    LOG.exiting(CLASS_NAME, "getServices", response);
    return response;
}
```

Listing of the New Integration Test Case for Our Updated Resource Class

```
@Test
public void filterAndSortServices() {
    // Register two services.
    resources
        .target("/services/" + serviceEntry.getName())
        .request()
        .put(Entity.json(sdUrl));
```

```
resources
.target("/services/" + serviceEntry.getName().substring(0, 7))
.request()
.put(Entity.json(sdUrl));

// Prepare the mock for findAll.
serviceEntry = new ServiceEntry();
serviceEntry.setName("Swagger");
serviceEntry.setUrl(sdUrl.getUrl());
services = new Services();
services.add(serviceEntry);
when(dao.findAll(Optional.of("Sw"), SortOrder.DESC))
.thenReturn(services);

// Filter and sort services.
final Services response =
    resources
        .target("/services/")
        .queryParam("name", "Sw")
        .queryParam("sort", SortOrder.DESC.name())
        .request()
        .get(Services.class);
assertNotNull(response);
assertThat(response).isEqualTo(services);
verify(dao).findAll(Optional.of("Sw"), SortOrder.DESC);
}
```

Listing of the New System Test Case for Our Updated Service

```
@Test
public void filterAndSortServices() {
    Response response = client.target(
        String.format(
            BASE_URI + "/services",
            RULE.getLocalePort()))
        .request()
        .get();
    assertThat(response.getStatus())
        .isEqualTo(Status.OK.getStatusCode());

    response = client.target(
        String.format(
            BASE_URI + "/services"
            + "?name=S&sort=deSc",
            RULE.getLocalePort()))
```

```
        .request()
        .get();
    assertThat(response.getStatus())
        .isEqualTo(Status.OK.getStatusCode());

    response = client.target(
        String.format(
            BASE_URI + "/services"
            + "?sort=INC",
            RULE.getLocalPort()))
        .request()
        .get();
    assertThat(response.getStatus())
        .isEqualTo(Status.BAD_REQUEST.getStatusCode());
}
```

In the implementation of the `getServices` method in our resource class, we must take care to protect ourselves against invalid sort criteria (this is exemplified in the above system test case). This is the reason to extend our exception class with another item, `INVALID_SORT_ORDER` (see the book's source code). Furthermore, we have converted the sort criteria characters to uppercase to increase flexibility (again, see the above system test case for an example). In our integration test case, we must prepare the mock of the `findAll` method differently than for the test case where all data is retrieved. Therefore, we have done some refactoring here too. All in all, we are ready to make a new release (certainly, all previous tests must pass as well).

6.7 Multiple Environments: Problem

The last version of the service was so successful that the QA team now considers Swaggerize as their key asset. The number of their registered services is indeed impressive. However, the team has noticed a difficulty of handling services deployed in various environments (development, staging, etc.). They would like an ability to choose the service by name for a particular environment. Of course, they don't want to lose existing data. So far, all services that were registered execute in the development area. They have already told the UI design team to start crafting the new index page.

As a sidenote, they also complain that on each occasional restart of the service, they need to restore the database from the last backup. They demand from us a pressing fix.

6.8 Multiple Environments: Resolution

This case study illustrates our obligation to worry about data migration during evolution. It is important to remember that all parts of the system should support evolution. If the data store is rigid, then it may equally hinder progress as any other constituent part. At some moment, the data itself will become the most precious property of an enterprise. Our job here is comprised of the major tasks: to resolve the problem of losing data on startup and to expand the system to support multiple environments. Dropwizard does have a solution, and this was an additional argument for its selection at the very beginning. Otherwise, we would need to integrate a suitable data migration framework ourselves or develop our own solution.

Preventing Data Loss The test configuration sets the `hibernate.hbm2ddl.auto` property to `create-drop`, while the production version assigns the value of `create`. The difference is that the former also drops the database schema when the system shuts down. However, in both cases the actual data in the database is pruned away. The reason that I've included this omission into our story line is to teach you a life-saving approach in urgent situations. This is the so-called improvisation technique coupled with an out-of-the-box thinking.

Many times, production problems require a two-pronged resolution: a dirty quick fix that would allow the system to continue with execution and a clean solution targeted for the next regular release (even if it is a purely maintenance one). Even though the hot fix isn't the right way to solve the problem, it may give you enough time to work on the correct approach. In our case, it has turned out that the QA team uses their central database to store registered services. Currently, they are running the Swaggerize service and have lots of data. An unfortunate restart of the service would delete them all. This gives us an idea for the dirty hack. We must tell them to alter the configuration file, while the service is still running, and set the previous property to `validate`. In this case, the service will just verify on startup that the target database does possess the required schema (table(s), index(es), etc.). This action solves the urgent need to prevent data loss. We have gained time to work on our next scheduled incremental release.

Supporting Multiple Environments This is a major upgrade of our service, as it changes both the API and the database model. Therefore, we will increase the version number to `2.0.0`. Let us start with the modified API as presented below:

Listing of the Patch to the Service Descriptor

```
'/services/{serviceName}/environments/{environmentName}  
/access':  
    parameters:  
        - name: serviceName  
          description: The unique name of the service.  
          in: path
```

```
type: string
required: true
- name: environmentName
  description: The unique name of the environment for this
    service.
  in: path
  type: string
  required: true
...
ServiceEntry:
  description: The record per service in the registry.
  allOf:
    - type: array
      items:
        $ref: '#/definitions/ServiceDescriptorURL'
    - type: object
      description: The unique name of the service.
      required:
        - name
      properties:
        name:
          type: string
ServiceDescriptorURL:
  description: The URL from where to retrieve the service
    descriptor.
  type: object
  required:
    - environment
    - url
  properties:
    url:
      description: The URL of the service descriptor.
      type: string
    environment:
      description: The designator for the environment
        (development, staging, etc.).
      type: string
```

To access the service via Swagger UI, we must know the correct URL for the service descriptor. Therefore, we must provide both the service's name as well as the matching environment's name. The service record now contains an array of ServiceDescriptorURL objects, one for each environment. Here is the listing of the patch to the service record representation class (see the book's source code for the ServiceDescriptorURL class):

Listing of the Patch to the ServiceEntry Representation Class (Getter/Setter for URLs Is Omitted)

```
@JsonDeserialize(using = ServiceEntry.ServiceEntryDeserializer.class)
public class ServiceEntry {
    static class UrlsSerializer
        extends JsonSerializer<Map<String, String>> {
            @Override
            public void serialize(
                Map<String, String> urls,
                JsonGenerator generator,
                SerializerProvider provider)
                    throws IOException, JsonProcessingException {
                generator.writeStartArray();
                for (Entry<String, String> entry : urls.entrySet()) {
                    generator.writeStartObject();
                    generator.writeStringField(
                        "environment", entry.getKey());
                    generator.writeStringField(
                        "url", entry.getValue());
                    generator.writeEndObject();
                }
                generator.writeEndArray();
            }
        }

        static class ServiceEntryDeserializer
            extends JsonDeserializer<ServiceEntry> {
            @Override
            public ServiceEntry deserialize(
                JsonParser parser,
                DeserializationContext context) throws IOException {
                ObjectCodec codec = parser.getCodec();
                JsonNode rootNode = codec.readTree(parser);
                JsonNode name = rootNode.get("name");
                JsonNode urls = rootNode.get("urls");
                ServiceEntry serviceEntry = new ServiceEntry();
                serviceEntry.setName(name.asText());
                for (int i = 0; i < urls.size(); i++) {
                    JsonNode sdUrl = urls.get(i);
                    serviceEntry.getUrls().put(
                        sdUrl.get("environment").asText(),
                        sdUrl.get("url").asText());
                }
            }
        }
    }
}
```

```

        return serviceEntry;
    }

}

@ElementCollection
@JoinTable(
    name = "SERVICE_URLS",
    joinColumns = @JoinColumn(name="Name"))
@MapKeyColumn(name = "Environment")
@Column(name = "Url")
@NotNull
@JsonSerialize(using = UrlsSerializer.class)
private Map<String, String> urls = new HashMap<>();

...
/***
 * Gets the URL of the service descriptor for the given
 * environment.
 *
 * @param environmentName the environment name.
 * @return the URL for the matching environment.
 * @throws NullPointerException if the input parameter
 * is {@code null}.
 * @throws IllegalArgumentException if the given
 * environment name isn't registered.
 */
public String getUrl(String environmentName) {
    Objects.requireNonNull(environmentName);
    if (urls.containsKey(environmentName)) {
        return urls.get(environmentName);
    } else {
        throw new IllegalArgumentException(
            "Bad environment identifier");
    }
}

```

At this moment, our code will not compile, so we will need to alter it for the compilation to pass. Our next step is to modify the fixture for the new JSON format of the service record class. This will serve as a reference to drive our custom serialization/deserialization implementation. The fixture is shown below:

```
{
  "name" : "Swaggerize",
  "urls": [
    {
      "url" : "http://localhost:8080/swagger.json",

```

```
        "environment": "development"
    },
{
    "url" : "http://localhost:8082/swagger.json",
    "environment": "staging"
}
]
}
```

The `ServiceEntry` class contains the `urls` field, which is a map that associates an environment with the URL (see Exercise 10). To serialize/deserialize our representation class into the format dictated by the fixture and the API specification, we must implement our own facility. This is done via those two static inner classes. We must also write tests for the utility method `getUrl` (see the book's source code for these tests).

The `urls` field is now a map, so it cannot be stored as such inside the relational database. Consequently, we must create a new table to store each element of the collection. Finally, we must establish a foreign key relationship between the collection and main tables. Of course, we could have structured the tables differently. However, this arrangement allows attaching other metadata to the service record (like an arbitrary description) without duplication. In some sense, it is more flexible to separate the URLs into a new table. These steps are all dictated by the relational database model. All in all, the `ServiceEntry` class requires the biggest augmentation in the 2.0.0 version of the service.

The DAO class requires only minimal changes. The `ServicesApi` resource class must accommodate the new way to access services. Besides the service's name, the client must also provide the name of the target environment. Here is the modified method that handles the service access:

Listing of the Patch to the ServicesApi Class

```
@POST
@Path("/{serviceName}/environments/{environmentName}/access")
@UnitOfWork
public Response accessService(
    @PathParam("serviceName")
    @NotEmpty
    String serviceName,
    @PathParam("environmentName")
    @NotEmpty
    String environmentName) {
    LOG.entering(CLASS_NAME, "accessService",
        new Object[] { serviceName, environmentName });
    Response response = null;
    try {
```

```

        ServiceEntry serviceEntry =
            serviceEntryDAO.findByName(serviceName);
        assert serviceEntry != null;
        response =
            Response
                .seeOther(
                    UriBuilder
                        .fromPath(SWAGGER_UI_ROOT)
                        .queryParam(
                            "url",
                            serviceEntry
                                .getUrl(environmentName))
                        .build())
                .build();
    } catch (IllegalArgumentException e) {
        throw new ApiException(
            ApiException.ErrorDescriptor.INVALID_ID, e);
    } catch (Exception e) {
        throw new ApiException(
            ApiException.ErrorDescriptor.INTERNAL_ERROR, e);
    }
    LOG.exiting(CLASS_NAME, "accessService", response);
    return response;
}

```

The error descriptor for the invalid identifier is changed to encompass both bad service and environment name. The integration, security, and system tests are altered to mirror the amendments, and you may analyze them in the book's source code. The index page resource must use the default environment (that was decided to be development) during service self-registration. The index page HTML rendering must be corrected as well. We cannot do a detour to include the more sophisticated UI design, though. So, we will stick with our simple Mustache template. Here is the new template to iterate over the registered services:

Listing of the Modified Mustache Template

```

<!DOCTYPE html>
<html>
<body>
    <h1>Registered Services</h1>
    <ul>
        {{#services}}
            <h2>{{name}}</h2>
            {{#urlSet}}
                <li>

```

```

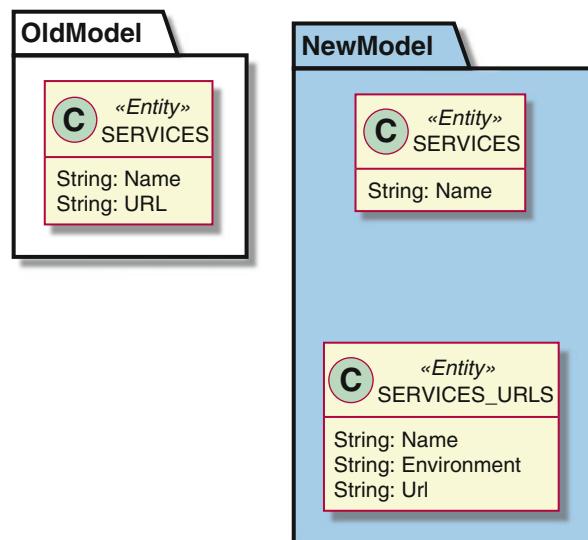
<a href="spec/swagger-ui/index.html?url={{value}}">
    {{key}}
</a>
</li>
{{/urlSet}}
{{/services}}
</ul>
</body>
</html>

```

To iterate over the URLs map, we need to get its content transformed into a set of entries. Therefore, the `ServiceEntry` class contains an auxiliary method `getUrlSet` (see the book's source code) to provide such a view. It is very important to annotate this method with `@JsonIgnore`. We don't want this helper view to show up in the JSON output. Without the necessity to migrate the old data over into the new model and properly solve the data preservation issue, we would be finished.

Migrating Data Into the New Format Figure 6.9 shows the entity-relationship (ER) diagrams for the old and the new model. The transformation is quite simple. For each service record in the old format, we just need to create the matching URL record with the environment set to development. We first need to add the following dependency into our build file:

Fig. 6.9 The ER diagrams of the old and the new model. The Name column is the primary key of the SERVICES table and is also the foreign key in the SERVICE_URLS table



```
<dependency>
    <groupId>io.dropwizard</groupId>
    <artifactId>dropwizard-migrations</artifactId>
</dependency>
```

Our next step is to add the `MigrationsBundle` bundle inside the `initialize` method of our main application class (see the Dropwizard Migrations guide for more details). The whole migration is driven in the background by the Liquibase database refactoring tool (see www.liquibase.org). The instructions for the tool are placed inside the `migrations.xml` file (you may also use other formats, like YAML or JSON) that is situated inside the `src/main/resources` folder. Here is the listing of this database migration driver file:

Listing of the migrations.xml File

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

<databaseChangeLog
    xmlns="http://www.liquibase.org/xml/ns/dbchangelog"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation=
        "http://www.liquibase.org/xml/ns/dbchangelog
        http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-3.1.
xsd">

    <preConditions
        onFailMessage= "Service is configured for MySQL only!">
        <dbms type="mysql"/>
    </preConditions>

    <changeSet id="createSERVICESTable"
        author="Ervin Varga">
        <preConditions onFail="MARK_RAN">
            <not>
                <tableExists tableName="SERVICES"/>
            </not>
        </preConditions>
        <createTable tableName="SERVICES">
            <column name="Name" type="varchar(255)">
                <constraints primaryKey="true"
                    nullable="false"/>
            </column>
        </createTable>
    </changeSet>
    <changeSet id="createSERVICESIndex"
```

```
        author="Ervin Varga">
<preConditions onFailure="MARK_RAN">
    <not>
        <indexExists indexName="IDX_SERVICES_PK"/>
    </not>
</preConditions>
<createIndex indexName="IDX_SERVICES_PK"
            tableName="SERVICES"
            unique="true">
    <column name="Name" type="varchar(255)"/>
</createIndex>
</changeSet>
<changeSet id="createSERVICE_URLSTable"
           author="Ervin Varga">
<preConditions onFailure="MARK_RAN">
    <not>
        <tableExists tableName="SERVICE_URLS"/>
    </not>
</preConditions>
<createTable tableName="SERVICE_URLS">
    <column name="Name" type="varchar(255)">
        <constraints nullable="false"/>
    </column>
    <column name="Environment" type="varchar(255)">
        <constraints nullable="false"/>
    </column>
    <column name="Url" type="varchar(255)">
        <constraints nullable="false"/>
    </column>
</createTable>
</changeSet>
<changeSet id="addFKConstraint"
           author="Ervin Varga">
<preConditions onFailure="MARK_RAN">
    <not>
        <foreignKeyConstraintExists
            foreignKeyName="FK_SERVICE_URLS"/>
    </not>
</preConditions>
<addForeignKeyConstraint
    constraintName="FK_SERVICE_URLS"
    baseColumnNames="Name"
    baseTableName="SERVICE_URLS"
    onDelete="CASCADE"
    onUpdate="RESTRICT"
```

```

        referencedColumnNames="Name"
        referencedTableName="SERVICES" />
    </changeSet>
    <changeSet id="moveDataFromV1.2"
               author="Ervin Varga">
        <preConditions onFail="MARK_RAN">
            <columnExists tableName="SERVICES"
                           columnName="URL"/>
        </preConditions>
        <sql>
            INSERT INTO SERVICE_URLS(Name, Environment, Url)
            SELECT Name, 'development', URL
            FROM SERVICES;
        </sql>
        <dropColumn columnName="URL"
                    tableName="SERVICES"/>
    </changeSet>
</databaseChangeLog>
```

Before we try to execute the migration, we must attach the `?nullNamePatternMatchesAll=true` suffix to the JDBC connection string. This is only MySQL specific and is caused by the past change in the default value of the previous property. Moreover, we must set the Hibernate strategy to validate instead of `create`, as we have already discussed. Our migration procedure must accomplish the following requirements:

- It must create all database artifacts when executed against a blank database.
- It must not overwrite or delete existing data nor schema elements.
- It must move data from previous version into the new model and update the old schema.

The migration is executed in steps, which are demarcated by change sets. Each change set should contain an atomic unit of work. Each set is protected by the precondition that dictates what should happen when it succeeds or fails. In our case, a failure means that the matching step is redundant. For example, this will happen when some of the database artifacts already exist from the previous version. We can also condition the whole script. Our first precondition is to ensure that we are running the script against the MySQL database. We have configured our service to currently support only this database, so there is no sense to try to create a schema in a foreign database.

The Liquibase tool automatically creates administrative tables in the target database. Figure 6.10 shows the content of the change log table inside the MySQL Workbench tool after we migrate stuff from the previous version of the service. We are now ready to release our 2.0.0 version of the service.

The screenshot shows a database query results window. At the top, there is a SQL command: `1 • SELECT * FROM Swaggerize_DB.DATABASECHANGELOG;`. Below the command is a table with the following data:

ID	AUTHOR	FILENAME	DATEEXECUTED	ORDEREXECUTED	EXECUTYPE
createSERVICESTable	Ervin Varga	migrations.xml	2017-09-10 23:32:40	1	MARK_RAN
createSERVICESIndex	Ervin Varga	migrations.xml	2017-09-10 23:32:41	2	MARK_RAN
createSERVICE_URLSTable	Ervin Varga	migrations.xml	2017-09-10 23:32:43	3	EXECUTED
addFKConstraint	Ervin Varga	migrations.xml	2017-09-10 23:32:44	4	EXECUTED
moveDataFromV1.2	Ervin Varga	migrations.xml	2017-09-10 23:34:59	5	EXECUTED

Fig. 6.10 When we execute the `db migrate` command (refer to the documentation available at www.dropwizard.io/1.1.4/docs/manual/migrations.html) on the previous version of the database model, we will skip the first two change sets. They are marked as `MARK_RAN`. The `ID` column designates the identifier of the change set

6.9 Record of Actions: Problem

As clients had been intensively using Swaggerize, they had stumbled across issues with target services. Sometimes, it was difficult to remember all the steps that were executed until the error had popped up. Repeating only the last action wasn't good enough, as that partial information couldn't reproduce the problem. The developers of the target service would have preferred a complete sequence. On the other hand, manually writing down everything wasn't viable either. Therefore, the clients of Swaggerize are asking us for an extension of the service to remember the actions taken by the user. They will have sent such a recording together with the bug report. So, they need also a way for our service to export the stored actions.

6.10 Record of Actions: Resolution

Issue reporting as a process should follow the same quality standards as any other endeavor. A well-crafted report is of tremendous help for a developer. If you want a prompt response about your issue, then you must first invest energy to properly document it. Otherwise, why would anyone pay attention for something about which even you don't care? If your description is vague, then people will either start a discussion to squeeze out more information from you (a lucky case) or will just ignore your report. During your professional career, you will be many times in both positions, as a reporter of an issue and as a receiver of such a report.

Before you jump into writing a single line of code, you should analyze the real need. We should be vigilant to recognize the mixture of statements from both problem and solution domains. The first part of the description seems to be located inside the problem domain. However, the second part is clearly the write-up of the solution. If we extract the essential parts of the request, then we will see that the client is interested in having some HTTP request/response recording facility. This doesn't necessarily imply extra code on our side. We shouldn't blame the client for trying to persuade us to expand our service. The members of our QA team don't know what other options exist. It is our job to find out and evaluate. We must propose a solution that will be a win-win situation. Of course, if we can solve the problem without even touching our service, that would be the best.

It turns out that we can employ an HTTP proxy with session recording capability. There are lots of available tools, and we will choose Charles (visit www.charlesproxy.com). It is a commercial product with a trial period that is available on all major operating systems (Windows, Mac OS X, and Linux). It may act as a Web proxy that sits between the client's browser and the network. It can record the HTTP traffic including requests, responses, and headers.⁹ Of course, we may also use Wireshark (see www.wireshark.org) to monitor the network activities, but that is a low-level network protocol analyzer. Obviously, we must choose a user-friendly tool that is comprehensible by our clients.

Charles has a very easy configuration mechanism that may automatically set up the system proxy settings (I'm using here the Mac OS proxy settings) when it starts up and shuts down. This means any browser on the Mac will use Charles as a proxy server when Charles is running. No extra step is required from the user.

All actions that happen inside the browser while Charles is recording them are grouped inside a named session. The session may be saved or loaded to/from disk. It is also possible to select a subset of request/responses for further examination. The session information is organized around two views: structure view and sequence view. The former offers a tree representation of actions categorized by the host name at the top. The latter shows the actions in chronological order as they were executed. Both views are useful to understand how things were done during the supervised period.

For a quick sanity check that everything properly works, we will start Charles, accept the Mac OS proxy autoconfiguration option, initiate the recording (choose the *Proxy → Start Recording* menu item), and visit example.com from our browser (we will use Google Chrome as our QA team). Figure 6.11 shows the structure view of the GET HTTP request. We may observe that the browser has issued a conditional GET request by setting the request header `If-None-Match` to the value it had received last time from the server. The server has responded with the 304 status code and set the `ETag` response header accordingly. This cycle has revealed to us that the browser just presented the cached content. Without peeking into this

⁹Charles has many other useful features, like simulating slow networks via bandwidth throttling, but we will only focus our attention on the HTTP proxy part.

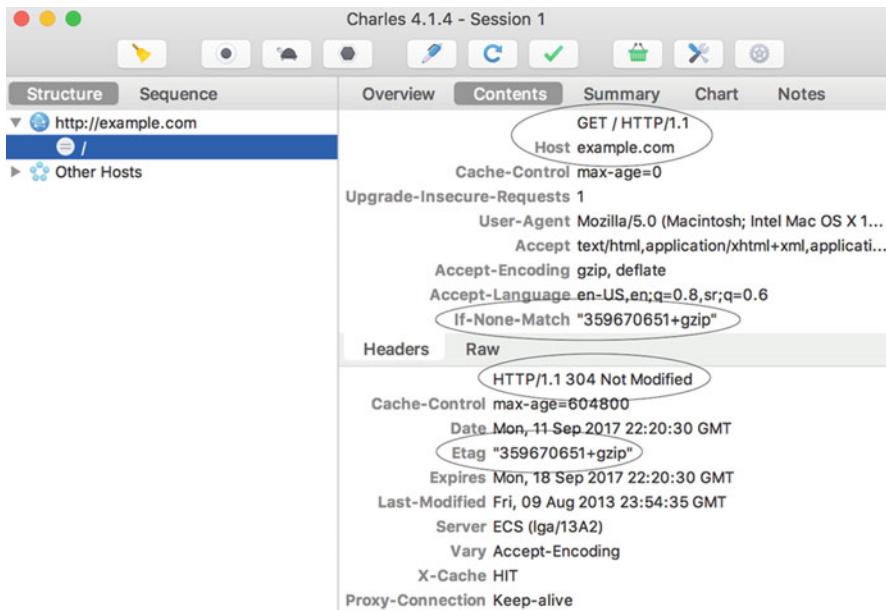


Fig. 6.11 The screenshot of Charles with a single request to visit the example.com Web site. On the left side, we see the example.com host name, and on the right, we may analyze what has happened in the request/response cycle after visiting the site. The most important details are annotated with ovals

level of detail, we couldn't tell the difference between the generated content and the cached one. Many times, an error happens because a stale cached content is served instead of a freshly produced one.

Providing a Detailed Recorded Session for Exercise 10 To make this discussion more pragmatic, we will produce an addendum to Exercise 10. This time, we will do exactly what our clients will also want to accomplish with their services. We will use the latest version of our service and execute all actions via its own Swagger UI. So here are the steps:

- Start the Swaggerize service as previously described.
- Visit the localhost:8080/swaggerize page from your browser.
- Select the Swaggerize service link (the autoregistered link is associated with the development environment) to open the Swagger UI page.
- Select the HTTP authorization option in Swagger UI. Enter Admin for the username and testAdmin for the password. This will “unlock” the endpoint for registering new services.
- Start recording the session inside Charles.

- Register a service under `Exercise10`, and use the same `dev.exercise10.com/swagger.json` URL for development and staging environments. This creates the situation as explained in Exercise 10.
- Stop the recording in Charles.
- Export the recorded session (choose *Export...* from the drop-down menu after selecting your actions). Figure 6.12 shows the sequence view of our actions. In case of a clutter in your session (due to multiple unrelated recorded activities), you may select your actions and choose *Clear Others* from the context menu.

There are many export formats and you may choose any from the list. I recommend the HTTP Archive format (HAR), as it is well-established. There are lots of tools that understand this format. By attaching this file to your report (in this

Code	Method	Host	Path	Start	Duration	Size
201	PUT	localhost:8080	/swaggerize/services/Exercise10	01:20:09	375 ms	861 bytes
201	PUT	localhost:8080	/swaggerize/services/Exercise10	01:20:19	328 ms	923 bytes

```
{
  "url": "dev.exercise10.com/swagger.json",
  "environment": "staging"
}

{
  "urls": [
    {
      "environment": "development",
      "url": "dev.exercise10.com/swagger.json"
    },
    {
      "environment": "staging",
      "url": "dev.exercise10.com/swagger.json"
    }
  ],
  "name": "Exercise10"
}
```

Fig. 6.12 The steps that we have used to showcase the problem associated with Exercise 10. We have selected the details for the second PUT request. On the lower part of the screen, we may observe the service's response, where it is clearly visible that the same URL is paired with both environments

case, Exercise 10), anyone may confidently reproduce the problem. Charles even allows you to replay the actions or edit them before replay.

Therefore, our task here would be to write a short document to the QA team (like this section) on how to install, configure, and use Charles. This concludes our Swaggerize story line.

6.11 Summary

This chapter has presented the development and evolution of the level 2 REST service called Swaggerize (see [7–9] for examples of how to combine this service style with model-driven software engineering and apply it in critical infrastructure systems). We have witnessed the smoothness of evolution when development follows the architecture-centric approach. This entails proper requirements work, focusing on APIs and quality (both assurance and control). The architecture plays the role of a reference model during change management and helps estimation, planning, and construction. The modular decomposition of a system allows parallel team work, especially in the case of large software-intensive systems. Every piece of the software must be covered with high-quality tests. This is essential; otherwise, the system would become brittle, and everybody would be afraid to touch it. Many times (as shown in the previous chapter), the system is technically legacy even after the first release.

Each segment in our story has demonstrated a facet of development and maintenance/evolution work. The development phase must consider evolution for the overall success of the product. Any rush work is just a technical debt with huge penalties (detrimental interest rate). The evolution is the longest software life cycle phase, and most changes happen there. Every new feature of a product potentially opens a novel evolutionary path. So, it isn't only about delivering that feature as soon as possible but also to prepare the product for future changes. The vector of changes (most probably directions where variations are expected) should always be reevaluated.

With a disciplined approach, you can eschew duplication, which has a huge impact on maintainability. Duplication doesn't necessarily mean avoiding repetitive code. It is a much broader term that advocates avoidance of all sorts of recurring work. If you solve a problem once, then you would like to reuse the solution multiple times (the whole pattern movement stems from this adage). This increases productivity and establishes consistency. For example, if you have solved a particular merge conflict in Git, then you wouldn't want to spend time on figuring out how to fix the same conflict again. If you've never heard of `Git rerere`, then read [10] for a superb introduction. Discipline is the foundation of professionalism.

The key takeaways from this chapter are as follows:

- The product cannot be an outcome of successive short-term projects. It must be framed and governed by product management, where architecture, APIs, and quality play the central role.

- A disciplined software engineering approach is the foundation for maintenance/evolution and the guarantee for the long-term success of a product. Agile methods require the same amount of discipline as plan-driven ones.
- The architecture is the gatekeeper of conceptual integrity of the product during maintenance and evolution. Without it, the maintenance process would be reminiscent of the *design by committee* phenomenon.
- The architecture is the determining factor whether you will have real evolution or will be obliged to switch into servicing mode (see [14] in Chap. 1). For an expected vector of changes, the architecture should enable smooth evolution. It isn't possible to flex on everything since quality attributes themselves don't play nicely together.
- Nothing can replace knowledge, experience, as well as willingness to think and act creatively.
- Disciplined development involves judicious and responsible reuse of code. The reuse must be evaluated against the requirements and cannot happen in an ad hoc manner. Irresponsible reuse, when you uncaringly amass pieces of code from the Internet, may have serious legal consequences [11]. Therefore, you must have control over reuse.

6.12 Exercises

1. Implement the necessary unit tests for representation classes (they are situated inside the `api` subpackage). Ensure that your unit tests end with `UnitTest`. This naming convention allows you to selectively turn tests on/off using the Maven's flag `-Dtest=\!TestName` (to selectively exclude) or `-Dtest=TestName` (to selectively include). You may also use wildcards. For example, to only run unit tests, you would use `-Dtest=*UnitTest`.
2. We have already mentioned in Chap. 5 that tests should be independent. Therefore, instead of putting deserialization logic into the serialization test, you should compare JSON documents directly. Of course, you cannot just compare them as ordinary strings since two semantically equivalent JSON documents may differ in the order of fields (JSON doesn't follow the strict ordering rule as XML). You may consider using the `JsonUnit` library (visit github.com/lukas-krecan/JsonUnit) or any similar for that matter. These libraries do internally use a JSON parser, but at least your test will not specifically depend on the deserialization logic related to your class.
3. Using the established procedure, implement the missing integration tests for the remaining representations. Try to avoid duplication. One option is to derive your tests from the new base class containing generic methods to test serialization and deserialization.
4. Implement the other exception mappers, and register them inside the application (for `JerseyViolationException` and `JsonProcessingException` exceptions). This would provide a better user experience, as all errors would then have a standardized response in the

form of our `ErrorReport` structure. You would want to introduce a common base mapper class to avoid duplication (don't forget to refactorize the `ApiExceptionMapper` class too).

5. Quality is comprised of two major constituents: quality assurance and quality control. Testing in general belongs to quality control. It cannot deliver quality alone. You need to also have quality assurance to diminish the probability of introducing defects. Tools may help you a lot. An excellent tool that may pinpoint potential defects in the Java code is FindBugs (see findbugs.sourceforge.net). It has decent defaults, so you will only need to tune the settings if you stumble across a warning that really doesn't apply to your system. Incorporate this tool into the build process and see what it will report. Try to repair potential warnings. As an extra assignment, try to include CheckStyle (see checkstyle.sourceforge.net) too. It has a lot of checks related to code, besides formatting aspects.
6. Associated with code quality is the quality of your tests. Again, tools may help you here. Combine testing with code coverage analysis, and configure your build process so it reports the discoveries. A good option is to use JaCoCo (see www.eclemma.org) or the recently open-sourced Atlassian Clover (see www.atlassian.com/software/clover).
7. Try to disconnect from the network or stop the database instance on Amazon to observe the status change in the `hibernate` health realm. This health check is done by Hibernate that internally uses the *circuit-breaker* mechanism [6].
8. Fix the bug in the 1.0.0 version of the service with minimal changes. This exercise is the perfect opportunity for you to practice debugging. Please, remember that we technically don't need to increase the version number, as we haven't released anything to the customer yet. For internal tracking, you may want to introduce a build number. At any rate, both your code and tests must function perfectly. (Hint: If you are stuck, then look at the solution named as `swaggerize_v2` in the accompanying source code of this book).
9. Implement the integration test suite for the `IndexPage` resource, similarly as we have done for the `ServicesApi` class.
10. In version 2.0.0, it is possible to attach the same URL to different environments. A resource may have multiple URLs, but a URL can only point to a single resource. Modify the code to incorporate more rigorous integrity checks. As an extra task, think about how this constraint might change and in what way if you would like to implement the last two items from our initial list (see the beginning of this chapter). For example, if we have a shared service descriptor, then obviously, the URL to retrieve it would be the same for all relevant services.

References

Further Reading

1. Tekinerdogan B, Grundy J, Ali N, Soley MR, Mistrik I (2015) Software quality assurance. Morgan Kaufmann, Burlington, MA. This book covers quality assurance for all phases of software development and maintenance. It will give you an insight how quality spreads out over the software's life-cycle, the importance to deliver quality everywhere. People usually think that quality is the primary concern only of a separate QA department. With such an attitude, there is no way to attain quality. This book will teach you why you should strive for quality in your daily work

Regular Bibliographic References

2. Robertson S (2015) The brown cow model. www.volere.co.uk/video-training/brown-cow. Accessed 02 Sept 2017
3. Fowler M (2010) Richardson maturity model. martinfowler.com/articles/richardsonMaturityModel.html. Accessed 02 Sept 2017
4. Cockburn A (2001) Writing effective use cases. Addison-Wesley, Boston, MA
5. Newman S (2015) Building microservices: designing fine-grained systems. O'Reilly Media, Sebastopol, CA
6. Nygard TM (2007) Release it!: design and deploy production-ready software. Pragmatic Bookshelf, Raleigh
7. Varga E, Lendak I, Gavrić M, Erdeljan A (2011) Applicability of RESTful web services in control center software integrations. IEEE 7th international conference on innovations in information technology INNOVATIONS'11, Abu Dhabi, United Arab Emirates
8. Lendak I, Varga E, Erdeljan A, Gavrić M (2010) RESTful web services and the common information model (CIM). IEEE international energy conference and exhibition ENERGYCON 2010, Manama, Kingdom of Bahrain
9. Lendak I, Varga E, Erdeljan E, Gavrić M (2010) RESTful access to power system state variables. IEEE region 8 international conference on computational technologies in electrical and electronics engineering, Irkutsk, Russia
10. Porteneuve C (2014) Fix conflicts only once with git rerere. hackernoon.com/fix-conflicts-only-once-with-git-rerere-7d116b2cec67. Accessed 13 Sept 2017
11. Sojer M, Henkel J (2011) License risks from Ad Hoc reuse of code from the internet. Commun ACM 54(12):74–81

Part III

Re-engineering and Reverse Engineering

Reverse engineering is about uncovering the secrets behind the product so that you may change it according to your needs. It is a passive activity, i.e., it doesn't alter the software. However, it is a crucial step before starting any reengineering effort (we assume that you don't have a nice and tidy software to work with). You cannot change or fix something that you don't comprehend. Redocumentation is a special case of reverse engineering. Nonetheless, many times it is about documenting unraveled stuff. Reengineering is the process of amending the product to some new form. A special case of reengineering is refactoring that is a set of small incremental behavior-preserving changes to the code base to increase its maintainability. This chapter unifies these maintenance techniques into a case study and presents a flow that can be applied in practice. The aim is to illuminate how to properly proceed in extensive restructuring work.

Frequently, products are intentionally safeguarded against reverse engineering (read [1] for an extensive treatment of reverse engineering). One standard method is code obfuscation (see [4] for evaluating code obfuscators based upon sustainability [5] criteria). We will demonstrate in our case study how you may apply this in your project. However, a legacy software usually looks like it was thoroughly obfuscated. The same remark pertains to a heavily optimized code. Therefore, you should resist performance tuning until you really need it.

Occasionally, you will need to reengineer a product to enable further evolution. We have already seen such an example in Chap. 5. Such endeavor is also required when the new mandatory feature cannot nicely fit into the current architecture. Sometimes, you are enforced to upgrade the technology, as the old one isn't supported anymore. This is especially pressing from the security standpoint. An unmaintained technology (e.g., an old operating system or application framework) with a security hole is extremely dangerous. Even without a security problem, with an outdated technology, you are prevented from keeping up with market needs.

An efficient way to postpone reengineering efforts is to continuously refactor your code (see [2–3] to deeply master this technique). This is commonly a viable approach, especially when you do have a superb tool support. Most Integrated

Development Environment (IDE) are equipped with an arsenal of restructuring types; some of them have been already demonstrated so far. There are two ways to refactor code: in small interleaved increments (floss refactoring) and in a batched mode (root canal refactoring, when a large set of changes are applied consecutively before moving on) [6]. Both ways are valid and you should decide what works best for you. Nevertheless, you may want to run a full suite of unit tests after each step. Even though an IDE should guarantee the correctness of actions (e.g., if you rename an element, then the IDE should execute a cascading update of references) your tests should serve as an extra safety net.

Refactoring is assumed to always improve things, as it keeps behavior intact while increasing maintainability. However, this isn't quite true when sustainability is considered. Again, a nice example where different quality attributes oppose each other (in this case, maintainability and sustainability). There is an anti-pattern called the God class (see <http://wiki.c2.com/?GodClass> for an overview about variations on this topic). This is a situation where you have a central class in your code that controls everything else. Sure, it is comprised of a huge lump of inconsistent elements with terribly low cohesion. You may want to mark it as an obvious target for refactorization. Read [7] how this decision might hamper sustainability. This is the principal reason why you should necessitate proper requirements handling and identify significant quality attributes buttressed by your system. You cannot just do something like refactoring because everybody writes about its positive effects. You must ensure that it also benefits your peculiar setting.

Reengineering and reverse engineering are inherently challenging jobs that require creativity and inventive ways of thinking. After all, you are in this venture to alter artifacts, which don't satisfy some conditions. The next series of examples will elucidate this in action. The first two are especially interesting reengineering specimens, as they focus on seeking a profoundly different alternative to attack the initial problem.

7.1 Performance Test Suite: Problem

The Electronic Design Automation (EDA) system team (recall the story from Chap. 5) has developed an ultrafast matrix library by leveraging Graphics Processing Unit (GPU) (a perfect use case for the SIMD family of computer architectures). The library may be used via the same generic client API as before. Unfortunately, the existing test suite isn't that persuasive, as we had discussed in Chap. 5. The range of values is limited to positive numbers that follow an exact pattern. The QA team had started an initiative for enhancing the tests by running the library with true random matrices. Their focus was on the matrix multiplication.

The developer assigned to the task had produced the following extension to the ClientMatrixAPIIntegrationTest class:

```
// Calculates C = A * B for regular matrices.
private Matrix<Double> multiplyMatrices(
    Matrix<Double> a, Matrix<Double> b) {
    final Matrix<Double> c = MatrixFactory.newInstance(
        a.getRows(), b.getColumns(), CHARACTERISTICS);
    for (int row = 0; row < a.getRows(); row++) {
        for (int column = 0; column < c.getColumns(); column++) {
            for (int k = 0; k < a.getColumns(); k++) {
                c.setElement(row, column,
                    c.getElement(row, column)
                    + a.getElement(row, k) * b.getElement(k, column));
            }
        }
    }
    return c;
}

private Matrix<Double> createRandomMatrix(int rows, int columns) {
    final Matrix<Double> result =
        MatrixFactory.newInstance(rows, columns, CHARACTERISTICS);
    for (int row = 0; row < rows; row++) {
        for (int column = 0; column < columns; column++) {
            result.setElement(row, column,
                ThreadLocalRandom
                    .current()
                    .nextDouble(Double.MIN_VALUE, Double.MAX_VALUE));
        }
    }
    return result;
}

private final int TESTS_COUNT = 20;
private final int MAX_DIMENSION_VALUE = 1300;

@Test
public void multiplyRandomMatrices() {
    final Random rnd = new Random();

    for (int count = 0; count < TESTS_COUNT; count++) {
        final int dimOuterRow = rnd.nextInt(MAX_DIMENSION_VALUE);
        final int dimInner = rnd.nextInt(MAX_DIMENSION_VALUE);
```

```

        final int dimOuterColumn = rnd.nextInt(MAX_DIMENSION_VALUE);
        final Matrix<Double> a =
            createRandomMatrix(dimOuterRow, dimInner);
        final Matrix<Double> b =
            createRandomMatrix(dimInner, dimOuterColumn);
        final Matrix<Double> cOrig = a.multiply(b);
        final Matrix<Double> cTest = multiplyMatrices(a, b);
        assertEquals(cTest, cOrig);
    }
}

```

The `multiplyMatrices` helper method is an example of the *N-version programming* technique.¹ The idea is to independently develop multiple implementations based upon the same specification (in our case, multiplying two matrices). These multiple variants of the software should eliminate the same defect in all versions (unless the specification contains an obvious error). It also attempts to reduce the chance of misunderstanding the specification in the same fashion, as those versions are supposed to be realized by different teams. In our case, the developer has created the simplest matrix multiplication routine. The rationale is sound, as he has wanted to ensure that the expected result is produced in the most straightforward manner.

The `multiplyRandomMatrices` executes matrix multiplication `TESTS_COUNT` of times with random matrices of various dimensions. The developer has let the test run for quite a long time and was happy to see the green bar at the end. Unfortunately, the QA team has refused to accept this expansion, as it completely ruined the purpose of the original tests. The aim was to have a suite of fast tests that can be run on demand. The current test suite could run for more than 10 min, which was unacceptable. Of course, the critical path is the simple brute-force test matrix multiplication procedure. All in all, our job is to improve the performance of the test suite while maintaining its capability.

Here, you must deal with reengineering the tests. So, it isn't always about changing the production code. The test suite must also follow up the technological improvements of the product. If your tests are sluggish, then they would become the bottleneck for further evolution. Even worse, they could be marked with `@Ignore`, thus directly reducing the quality of the deliverable.

¹The infamous catastrophe that has its root cause in the lack of N-version programming coupled with bad reuse is the Ariane 5 Flight 501 Failure (read more at www-users.math.umn.edu/~arnold/disasters/ariane.html). In software, you cannot just replicate the same code on multiple nodes and presume that you have a fault-tolerant design. A condition that causes a failure in one software module would cause a failure in another one too. So, you must execute different versions of the software on multiple hardware devices to protect against a cascading effect. Every software has its own peculiar set of defects. The goal is to prevent alignment. A good allegory on this is the so-called *Swiss cheese model of system accidents* (see www.ncbi.nlm.nih.gov/pmc/articles/PMC1117770/).

7.2 Performance Test Suite: Resolution

There are two general paths that you may choose: to speed up the `multiplyMatrices` method and to devise a completely dissimilar tactic to check for correctness. The former bears a considerable risk. A more advanced matrix multiplication could introduce an error, so your tests may fail (either by needlessly alarming an issue or give a false impression that everything is ok). Therefore, we will elect the latter option.

The previous brute-force method belongs to the class of deterministic approaches, where the same code will always execute in the same way, giving the same result for the given input. There is also a category of so-called probabilistic methods that can operate under some chance regarding the correctness of the result. Nonetheless, many of these methods may provide a solid assurance about the outcome in a very short time.

Suppose that we exchange matrix-matrix with matrix-vector multiplication. The asymptotic behavior would change from $O(n^3)$ to $O(n^2)$, and this is a significant improvement. The idea is to generate a random vector, V . We would produce our original matrix $C = AB$ with the ultrafast routine as before. However, we will check the rightness of the fast procedure by calculating the next two vectors, R_{orig} and R_{test} , relying on the associative property of matrix multiplication:

$$R_{\text{test}} = A(BV) \tag{7.1}$$

$$R_{\text{orig}} = CV \tag{7.2}$$

Let us denote the dimensionality of our matrices A and B as $n \times m$ and $m \times k$, respectively. The dimensionality of the resulting matrix C will obviously be $n \times k$. The dimensionality of the vector V must be $k \times 1$. Finally, the dimensionality of the resultant vectors R_{test} and R_{orig} will be $n \times 1$. If $R_{\text{test}} \neq R_{\text{orig}}$, then the fast routine has an error. Nonetheless, if $R_{\text{test}} = R_{\text{orig}}$, then we still cannot be 100% sure in its correctness. We would need to repeat the above steps for some `CHECKS_COUNT` times. The final complexity would become $O(\text{CHECKS_COUNT} * n^2)$. Keeping the `CHECKS_COUNT` on 5 would be more than enough. Here is the modified implementation of the test suite (notice that we can absolutely reuse the brute-force matrix multiplication routine):

```
private final int CHECKS_COUNT = 5;

@Test
public void multiplyRandomMatricesFastCheck() {
    final Random rnd = new Random();

    for (int test_count = 0; test_count < TESTS_COUNT;
         test_count++) {
        final int dimOuterRow = rnd.nextInt(MAX_DIMENSION_VALUE);
```

```

        final int dimInner = rnd.nextInt(MAX_DIMENSION_VALUE);
        final int dimOuterColumn = rnd.nextInt(MAX_DIMENSION_VALUE);
        final Matrix<Double> a =
            createRandomMatrix(dimOuterRow, dimInner);
        final Matrix<Double> b =
            createRandomMatrix(dimInner, dimOuterColumn);
        final Matrix<Double> cOrig = a.multiply(b);

        for (int check_count = 0; check_count < CHECKS_COUNT;
             check_count++) {
            // We will represent the vector as a distorted matrix.
            final Matrix<Double> v =
                createRandomMatrix(dimOuterColumn, 1);
            final Matrix<Double> r1 =
                multiplyMatrices(a, multiplyMatrices(b, v));
            final Matrix<Double> r2 = multiplyMatrices(cOrig, v);
            assertEquals(r1, r2);
        }
    }
}

```

The original test case was renamed to `multiplyRandomMatricesSlowCheck` and annotated with `@Ignore` (you might want to remove it completely). The new way of checking for correctness executes within a reasonable time. Of course, in our code base, we still don't have that ultrafast GPU-based matrix multiplication, so the tests wouldn't run that fast. At any rate, the QA department is happy to accept our reengineered solution.

7.3 Extension of the System: Problem

The ACME Corporation had developed the 1.0.0 version of the Tank Filler software for its client. The number of tanks is in the range of $[2, \text{MAX_TANKS}]$ and is an input to the program (let us denote this as T_{num}). Each tank T_i has some initial unknown amount of liquid (may be zero as well) that should be the output of the calculation. The only additional input is the final desired amount in each tank denoted as Z . The filling process executes the following logic (given here in pseudocode) to reach Z :

```

for i = 1 .. (Tnum - 1) do
    Ti = Ti / 2
    Ti+1 = Ti+1 + Ti
end
T1 = T1 + TTnum / 4
TTnum = 3 * TTnum / 4

```

For example, with $Z = 6$ and $T_{\text{num}} = 2$, the program would output 8 and 4 for the first and second tanks, respectively. With $Z = 6$ and $T_{\text{num}} = 3$, the result would be 8, 8, and 2. Here is the listing of the `TankFiller` class as well as the tests:

Listing of the `TankFiller` Class

```
public class TankFiller {  
    public static final int MAX_TANKS = 100;  
    private final int numTanks;  
    private final double z;  
  
    /**  
     * Initializes the TankFiller calculator with input values.  
     *  
     * @param numTanks the number of tanks.  
     * @param z the final amount to reach in each tank.  
     * @throws IllegalArgumentException if the input is invalid.  
     */  
    public TankFiller(int numTanks, double z) {  
        if (numTanks < 2 || numTanks > MAX_TANKS) {  
            throw new IllegalArgumentException(  
                "The number of tanks must be in the range of [2,"  
                + MAX_TANKS + "]");  
        }  
        if (z < 0) {  
            throw new IllegalArgumentException(  
                "The final amount of liquid cannot be a negative number");  
        }  
  
        this.numTanks = numTanks;  
        this.z = z;  
    }  
  
    /**  
     * Performs the calculation for the given input.  
     *  
     * @return the initial amounts (the first tank's index is zero).  
     */  
    public double[] calculate() {  
        assert numTanks >= 2 && numTanks <= MAX_TANKS:  
        "The number of tanks is out of range";  
    }  
}
```

```
double[] tanks = new double[numTanks];
switch (numTanks) {
    case 2: return calculateForTwoTanks(tanks);
    case 3: return calculateForThreeTanks(tanks);
    default: return calculateForManyTanks(tanks);
}
}

private double[] calculateForTwoTanks(double[] tanks) {
    tanks[0] = 4.0 * z / 3.0;
    tanks[1] = 2.0 * z / 3.0;
    return tanks;
}

private double[] calculateForThreeTanks(double[] tanks) {
    tanks[0] = 4.0 * z / 3.0;
    tanks[1] = 4.0 * z / 3.0;
    tanks[2] = z / 3.0;
    return tanks;
}

private double[] calculateForManyTanks(double[] tanks) {
    tanks[0] = 4.0 * z / 3.0;
    tanks[1] = 4.0 * z / 3.0;
    for (int i = 2; i < numTanks - 1; i++) {
        tanks[i] = z;
    }
    tanks[numTanks - 1] = z / 3.0;
    return tanks;
}

/**
 * Main entry point into the program.
 *
 * @param args the input parameters: [number of tanks]
 * [final amount z]
 */
public static void main(String[] args) {
    if (args == null || args.length != 2) {
        System.err.println(
            "Usage: java -jar tank-filler.jar "
            + "<number of tanks> <final amount z>");
        return;
    }
    double[] tanks =
```

```
        new TankFiller(
            Integer.parseInt(args[0]),
            Double.parseDouble(args[1]))
        .calculate();
    System.out.println(
        "Initial amounts for tanks:" + Arrays.toString(tanks));
}
}
```

Listing of the Unit Test Suite

```
public class TankFillerUnitTest {
    private boolean checkSolution(double[] tanks, double z) {
        for (int i = 0; i < tanks.length - 1; i++) {
            tanks[i] /= 2.0;
            tanks[i + 1] += tanks[i];
        }
        tanks[0] += tanks[tanks.length - 1] / 4.0;
        tanks[tanks.length - 1] *= 3.0 / 4.0;
        return DoubleStream.of(tanks).allMatch(x -> x == z);
    }

    @Test
    public void calculateForDifferentNumberOfTanks() {
        for (int num_tanks = 2; num_tanks < TankFiller.MAX_TANKS;
             num_tanks++) {
            final TankFiller tankFiller =
                new TankFiller(num_tanks, 6);
            assertTrue(
                "Num_tanks: " + num_tanks,
                checkSolution(tankFiller.calculate(), 6));
        }
    }

    @Test(expected = IllegalArgumentException.class)
    public void tryCreatingTankFillerWithTooFewTanks() {
        new TankFiller(1, 5);
    }

    @Test(expected = IllegalArgumentException.class)
    public void tryCreatingTankFillerWithTooManyTanks() {
        new TankFiller(TankFiller.MAX_TANKS + 1, 5);
    }
}
```

```

    @Test(expected = IllegalArgumentException.class)
    public void tryCreatingTankFillerWithNegativeFinalAmount() {
        new TankFiller(3, -5);
    }

    @Test
    public void startProgramUsingTheMainMethod() {
        // Capture the standard output.
        final ByteArrayOutputStream stdOutputBuffer =
            new ByteArrayOutputStream();
        final PrintStream standardOutput = System.out;
        System.setOut(new PrintStream(stdOutputBuffer));

        // Execute the program and check the outcome.
        TankFiller.main(new String[] { "3", "6" });
        assertEquals(
            "Initial amounts for tanks: [8.0, 8.0, 2.0]\n",
            stdOutputBuffer.toString());

        // Release the capture of the standard output.
        System.setOut(standardOutput);
    }
}

```

The logic of the program revolves around the three major cases: two, three, and more than three tanks (see Exercises 1 and 2). The formulas used in the various `calculate*` methods were derived directly from the tank filling logic. You may verify them by simply solving a set of linear equations. The test suite uses the filling logic to reexamine that the output from the program indeed leads to final amount Z in all tanks. Essentially, both the formulas and the tests “fast forward” the filling logic toward the variable Z.

After the success with this version, the ACME Company had started to offer this program to other clients too. As usual, they had different expectations. One strategic client had demanded the following extensions:

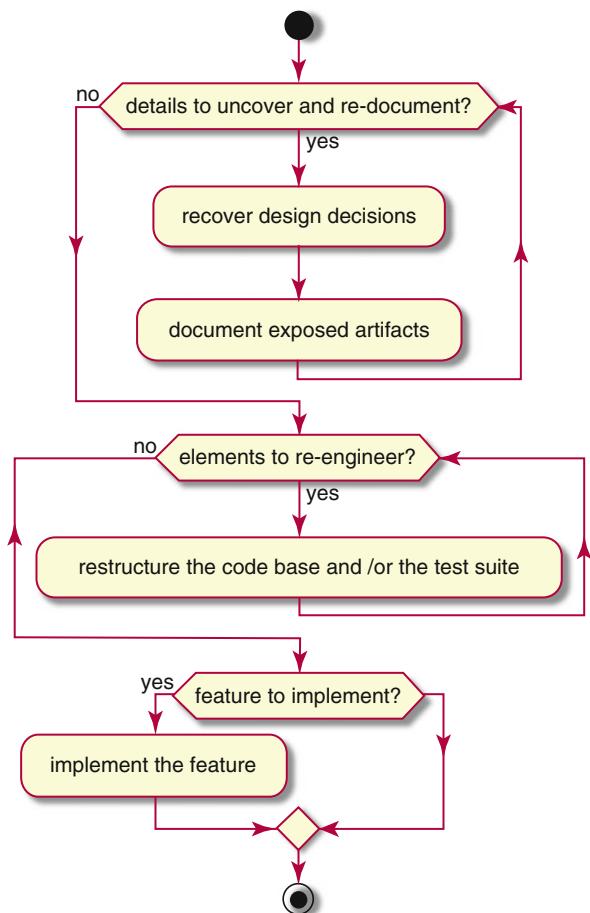
- The final amount should be per tank, rather than a shared value for all tanks.
- The filling logic should be programmable by allowing the following commands: `move_absolute(tank_from, tank_to, absolute amount)` and `move_relative(tank_from, tank_to, percentage)`. The system should provide a simple API for creating tank filling mini-programs.
- They want a more user-friendly test suite for showcasing the usage of the program. The `startProgramUsingTheMainMethod` test is arcane for them.

We are engaged to reshape the original program to accommodate the new requirements.

7.4 Extension of the System: Resolution

This job has three main constituents: redocumenting the behavior of the software for the client, reengineering the calculator, and implementing a new programmable tank filling logic. Figure 7.1 shows a general Unified Modeling Language (UML) activity diagram regarding reengineering and reverse engineering (a good illustration of pairing reverse engineering with construction is given inside my JXUnit Primer series [8]).

Fig. 7.1 The UML activity diagram of the reverse engineer, reengineer, and construct iterations. A complete endeavor may contain multiple such iterations



Revealing and Redocumenting the Planned Behavior for the Client We will start with the reverse engineering pursuit from the perspective of the client. Clients cannot understand cryptic unit tests since they aren't accustomed to reading Java code. Rewriting the test specification into a form acceptable by clients has twofold benefits: we achieve the stated goal to have higher-level tests and we also get the blueprint for our upcoming reengineering work. A test specification at a decent abstraction level uncovers hidden details related to software usage and behavior. For this purpose, we will utilize behavior-driven development (BDD), which is a variant of TDD. We will use Cucumber (visit cucumber.io), which is a special-purpose tool to support BDD. Another similar option is JBehave (visit jbehave.org).

In Cucumber, you specify a test using a formatted English text. This is executed by matching segments of that text with regular expressions. Each matched expression triggers an appropriate action implemented in a target programming language (in our case, Java). Behind the scenes, all actions are run under the JUnit framework, as Cucumber provides its own JUnit test runner. Below is the listing of the acceptance test for our client (this replaces that incomprehensible unit test from our previous list). The location of the specification file should follow the package structure of step definitions. In our case, the file is situated in the `src/test/resources/com/example/tank_filler/client/cucumber` folder. The step definitions will go into the `src/test/java/...` directory. The “basic” program is our original one from the beginning of this case study. The “alternate” is a simple program that transfers 50% of the content from a lower indexed tank to a higher indexed term for odd indexes and 50% of the content from a higher indexed tank to a lower indexed tank for even indexes.

Listing of the Tank Filler Acceptance Test Specification

```

@TankFillerAcceptanceTest
Feature: Performing Calculations Using a Command Line Interface
    I want to calculate the initial amounts of liquid in each tank
    given the tank filling program's name and final amounts. The
    number of tanks must also be provided on input (the range is
    [2, MAX_TANKS], where MAX_TANKS is configured to be 100).

Background: An already prepared programs
    Given the programs basic and alternate exist

Scenario: Missing Required Arguments
    When I do not provide all the required arguments on startup
    Then the application should show the help message

Scenario: Non-existent Program
    When I do not provide the proper program variant name
    Then the application should show an error

```

Scenario: Invalid Number of Tanks

When I provide a wrong number of tanks (out of range)
 Then the application should show an error

Scenario: Incomplete Final Amounts

When I do not provide all final amounts
 Then the application should show an error

Scenario: Invalid Initial Conditions

When I specify the number of tanks: 4
 And I specify the program variant name: alternate
 And I specify the final amount(s): 4,8,10,12
 Then the application should enter illegal state

Scenario Outline: Calculate Initial Amounts

When I specify the number of tanks: <numTanks>
 And I specify the program variant name: <progName>
 And I specify the final amount(s): <finalAmounts>
 Then the output should be: <initialAmounts>

Examples:

numTanks	progName	finalAmounts	initialAmounts
2	basic	6,6	[8.0, 4.0]
3	basic	6,6,6	[8.0, 8.0, 2.0]
4	basic	6,6,6,6	[8.0, 8.0, 6.0, 2.0]
2	alternate	3,5	[3.0, 5.0]
3	alternate	6,6,6	[12.0, 6.0, 0.0]
4	alternate	4,14,5,12	[8.0, 2.0, 1.0, 24.0]

The Cucumber specification file has an extension, `feature`. I recommend you use an appropriate plug-in for your IDE to edit such files. Step definitions should be implemented for all lines starting with Given, When, And, and Then. Each scenario describes a user story (a concrete occurrence of an interaction with the application). The outline scenario is a template, which receives data from examples. Scenarios are grouped into a feature. I recommend you spend some time giving your feature a good title and description. After all, this is the summary of what is contained the specification file. A feature may be annotated to trigger some preparatory work before its execution. The background is a generic precondition that is applied at the beginning of each scenario (remember that an outline scenario will produce many subordinate clones as there are rows in the Examples table).

The main advantage of using Cucumber tests is its comprehensibility even for nontechnical people. They can just read a plain English text and understand what is going on. All the gory details are hidden inside step definitions. The Cucumber specification file of a feature should contain both happy scenarios and edge cases. In this way, a client can grasp a full picture regarding the application's usage (see

Exercise 3). We will need to add the following dependencies to our build file (the property `cucumber.version` is set to 1.2.5):

```
<dependency>
    <groupId>info.cukes</groupId>
    <artifactId>cucumber-java</artifactId>
    <version>${cucumber.version}</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>info.cukes</groupId>
    <artifactId>cucumber-junit</artifactId>
    <version>${cucumber.version}</version>
    <scope>test</scope>
</dependency>
```

The next task is to implement the `TankFillerAcceptanceTest` driver class that just initiates the Cucumber framework to execute the specification. Here is the listing of this class:

Listing of the TankFillerAcceptanceTest Class

```
package com.example.tank_filler.client.cucumber;

import org.junit.Test;
import org.junit.runner.RunWith;

import com.example.tank_filler.TankFiller;

import cucumber.api.CucumberOptions;
import cucumber.api.junit.Cucumber;

@RunWith(Cucumber.class)
@CucumberOptions(
    monochrome = true,
    plugin = {
        "pretty",
        "json:target/cucumber.json",
        "rerun:target/rerun.txt"
    },
    dryRun = false
)
public class TankFillerAcceptanceTest {
```

The driver is instructed to produce a JSON test run report inside the target folder. We should execute the above test from the IDE even without any step definitions. The console will contain the following helper messages for us (a very useful way to boost our productivity):

You can implement missing steps with the snippets below:

```
@Given("^the programs basic and alternate exist$")
public void the_programs_basic_and_alternate_exist() throws Throwable
{
    // Write code here that turns the phrase above into concrete
    // actions
    throw new PendingException();
}

@When("^I do not provide all the required arguments on startup$")
public void i_do_not_provide_all_the_required_arguments_on_startup()
throws Throwable {
    // Write code here that turns the phrase above into concrete
    // actions
    throw new PendingException();
}
...
```

If we now implement the preliminary version of our step definitions and run the acceptance test, then the JUnit report will clearly mark our scenarios as ignored but keep the bar green. The console will contain the following messages:

```
cucumber.api.PendingException: TODO: implement me
                                at com.example.tank_filler.client.cucumber.
TankFillerAcceptanceTestStepdefs.
the_programs_basic_and_alternate_exist
(TankFillerAcceptanceTestStepdefs.java:12)
    at *.Given the programs basic and alternate exist(com/example/
tank_filler/client/cucumber/PerformingCalculations.feature:10)
...
...
```

Reengineering the Calculator Our original approach cannot prevail. There is no way to formulate the initial amounts in such a tidy manner, as we have done before. We will need a different strategy. Figure 7.2 shows a general problem solving pattern known as the primal-dual method. This approach is best recognized in the realm of mathematical optimization theory, where an optimization problem may be

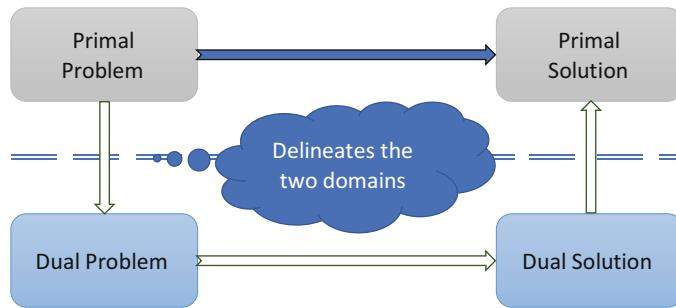


Fig. 7.2 If we cannot move inside the primal domain, then we can transform the primal problem into the dual problem. Once we get the dual solution, then we apply an inverse transform to reach the primal solution. Sometimes, these two solution spaces overlap

viewed from two angles: the primal problem and/or the dual problem. However, as a paradigm, it has a much broader influence. Table 7.1 shows the steps for attaining the solution exclusively inside the primal domain (for simplicity, we will reduce the number of tanks to three).

The last row in Table 7.1 gives the set of equations that indicate all must be equal to Z . If we substitute $\frac{T_3 + T_1}{2}$ with Z , then we can find the expression for T_3 as a function of Z . Afterward, we may solve for T_1 and finally for T_2 .

The dual case is shown in Table 7.2. Instead of starting from the beginning and working toward the end, we turn the problem upside down. We start from the final state and proceed toward the initial amounts. In the dual space, we don't solve linear equations but just perform trivial inverse actions related to the flipped tank filling process (as if the program would execute from the end to the beginning). In this case, the dual solution is the same as the primal solution, so we don't need to apply any extra transformation at the end. As a matter of fact, the last row here is exactly the solution for our set of linear equations from the primal problem.

We now have a clear perception of how to find the initial amounts. We will just execute the tank filling program in an opposite direction and apply inverse actions. For example, `move_absolute(T1, T2, 10)` would be run as `move_absolute(T2, T1, 10)`. We can do this since all commands in our tank filling program language are uniquely invertible.² Notice that we may easily figure out whether the program is correct for the given input. If at any moment, we encounter a tank with a negative amount of liquid, we can immediately stop the execution with an error signal.

²This tank filling language belongs to the family of languages known as domain-specific languages (DSLs). DSLs are very popular, as they considerably reduce the accidental complexity for writing domain-specialized software.

Table 7.1 Procedure to solve the tank filling calculation in the primal space

T_1	T_2	T_3
$\frac{T_1}{2}$	$T_2 + \frac{T_1}{2}$	T_3
$\frac{T_1}{2}$	$\frac{T_2 + \frac{T_1}{2}}{2}$	$T_3 + \frac{T_2 + \frac{T_1}{2}}{2}$
$\frac{T_1}{2} + \frac{1}{4} \left(\frac{T_3 + T_2 + \frac{T_1}{2}}{2} \right)$	$\frac{T_2 + \frac{T_1}{2}}{2}$	$\frac{3}{4} \left(\frac{T_3 + T_2 + \frac{T_1}{2}}{2} \right)$

Table 7.2 Procedure to solve the tank filling calculation in the dual space

Z	Z	Z
$\frac{2}{3}Z$	Z	$\frac{4}{3}Z$
$\frac{2}{3}Z$	$2Z$	$\frac{1}{3}Z$
$\frac{4}{3}Z$	$\frac{4}{3}Z$	$\frac{1}{3}Z$

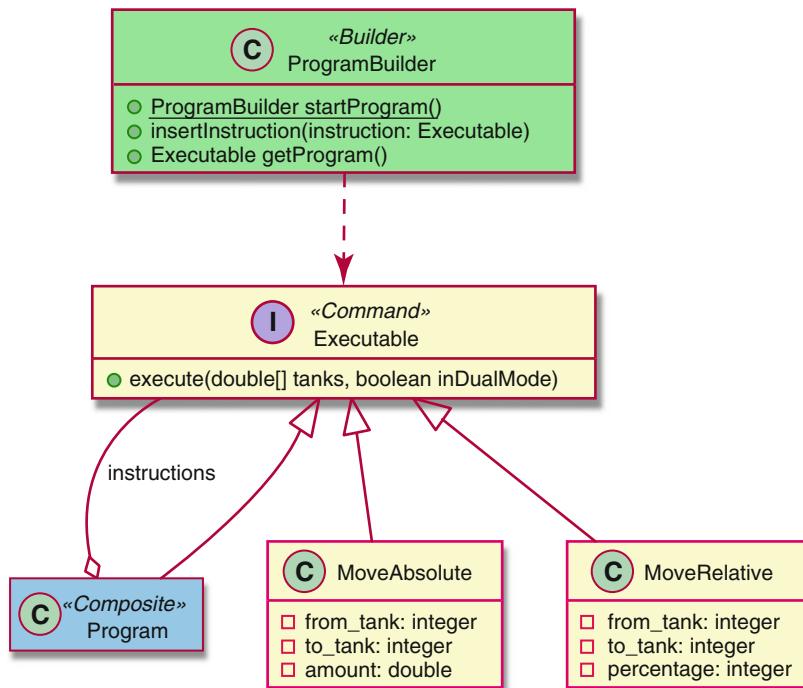


Fig. 7.3 The UML class diagram showing the major elements of the program builder module. In dual mode, the `Program` composite will execute the instructions in the opposite direction. Likewise, the instructions will perform their inverse in dual mode. There are also utility methods in the `ProgramBuilder` class for producing individual instructions

Implementing the Programmable Tank Filling Logic The implementation revolves around the Builder, Composite, and Command patterns (read [9] for more details about these patterns). Figure 7.3 presents the UML class diagram of the program builder module. The application of the Composite pattern allows us to

introduce subprograms. For example, we may easily create a “mix” program by joining our “basic” and “alternate” subprograms inside a top composite.

Here is the listing of the Executable interface:

Listing of the Executable Interface

```
/**
 * An entry point to execute both an instruction and a program.
 *
 * @author ACME Developers
 * @since 2.0
 */
public interface Executable {
    /**
     * Executes the instruction or the program. In dual mode, a
     * program is executed in the opposite order, and the instruction
     * must run its own reverse.
     *
     * @param tanks the array of amounts for tanks.
     * @param inDualMode a flag indication whether dual mode is
     * turned on or off.
     * @throws IllegalStateException if the execution of this
     * instruction or program leads to an illegal system state. This
     * may happen if there is a negative amount of liquid in some
     * tank, or the indexes are out of bounds.
     */
    void execute(double[] tanks, boolean inDualMode);
}
```

Next is the listing of the MoveAbsolute concrete instruction with unit tests (for the other instruction, see the book’s source code and Exercise 4):

Listing of the MoveAbsolute Instruction

```
/**
 * This instruction moves the given amount of liquid from the source
 * tank into the target tank.
 *
 * @author ACME Developer
 * @since 2.0
 */
```

```
final class MoveAbsolute implements Executable {
    private final int fromTank;
    private final int toTank;
    private final double amount;

    /**
     * Creates an instance of the instruction to move a given
     * amount of liquid from one tank into another.
     *
     * @param fromTank the source tank's index.
     * @param toTank the target tank's index.
     * @param amount the amount to move.
     * @throws IllegalArgumentException if any argument is negative.
     */
    public MoveAbsolute(int fromTank, int toTank, double amount) {
        if (fromTank < 0 || toTank < 0 || amount < 0) {
            throw new IllegalArgumentException();
        }

        this.fromTank = fromTank;
        this.toTank = toTank;
        this.amount = amount;
    }

    @Override
    public void execute(double[] tanks, boolean inDualMode) {
        try {
            if (inDualMode) {
                tanks[fromTank] += amount;
                tanks[toTank] -= amount;
            } else {
                tanks[fromTank] -= amount;
                tanks[toTank] += amount;
            }
            if (tanks[toTank] < 0 || tanks[fromTank] < 0) {
                throw new IllegalStateException(
                    "Negative amount of liquid in tank(s)");
            }
        } catch (IndexOutOfBoundsException e) {
            throw new IllegalStateException(
                "Indexes are out of bounds", e);
        }
    }
}
```

Listing of the MoveAbsoluteUnitTest Class

```
public class MoveAbsoluteUnitTest {
    private static final double TOLERANCE = 0;
    private final double[] tanks = new double[] { 10.0, 7.0 };
    private final Executable instruction =
        new MoveAbsolute(0, 1, 5.0);

    @Test
    public void executeInPrimalMode() {
        instruction.execute(tanks, false);
        assertEquals(5.0, tanks[0], TOLERANCE);
        assertEquals(12.0, tanks[1], TOLERANCE);
    }

    @Test
    public void executeInDualMode() {
        instruction.execute(tanks, true);
        assertEquals(15.0, tanks[0], TOLERANCE);
        assertEquals(2.0, tanks[1], TOLERANCE);
    }

    @Test(expected = IllegalArgumentException.class)
    public void tryCreatingAnInstructionWithNegativeFromIndex() {
        new MoveAbsolute(-1, 1, 5.0);
    }

    @Test(expected = IllegalArgumentException.class)
    public void tryCreatingAnInstructionWithNegativeToIndex() {
        new MoveAbsolute(0, -1, 5.0);
    }

    @Test(expected = IllegalArgumentException.class)
    public void tryCreatingAnInstructionWithNegativeAmount() {
        new MoveAbsolute(0, 1, -5.0);
    }

    @Test(expected = IllegalStateException.class)
    public void tryExecutingAnInstructionCausingNegativeAmount() {
        new MoveAbsolute(0, 1, 50.0).execute(tanks, false);
    }

    @Test(expected = IllegalStateException.class)
    public void tryExecutingAnInstructionWithOutBoundToIndex() {
```

```
        new MoveAbsolute(0, 2, 5.0).execute(tanks, true);
    }

    @Test(expected = IllegalStateException.class)
    public void tryExecutingAnInstructionWithOutBoundFromIndex() {
        new MoveAbsolute(2, 1, 5.0).execute(tanks, false);
    }
}
```

The program encompasses multiple instructions and/or subprograms. Notice that the reversal of execution in dual mode has a cascading effect for subprograms. Here are the listings of the `Program` class and the accompanying unit tests:

Listing of the Program Class

```
/**
 * Class that holds a program. It consists of instructions or other
 * subprograms.
 *
 * @author ACME Developer
 * @since 2.0
 */
final class Program implements Executable {
    private final LinkedList<Executable> program;

    Program(Collection<Executable> instructions) {
        program = new LinkedList<>(instructions);
    }

    @Override
    public void execute(double[] tanks, boolean inDualMode) {
        Iterator<Executable> instructions;
        if (inDualMode) {
            instructions = program.descendingIterator();
        } else {
            instructions = program.iterator();
        }
        while (instructions.hasNext()) {
            instructions.next().execute(tanks, inDualMode);
        }
    }
}
```

Listing of the ProgramUnitTest Class

```

public class ProgramUnitTest {
    private static final double TOLERANCE = 0;
    private static Program TEST_PROGRAM;

    @BeforeClass
    public static void constructASimpleProgram() {
        final ArrayList<Executable> instructions = new ArrayList<>();
        instructions.add(new MoveRelative(0, 1, 50));
        instructions.add(new MoveRelative(1, 2, 50));
        instructions.add(new MoveRelative(2, 0, 25));
        TEST_PROGRAM = new Program(instructions);
    }

    @Test
    public final void runProgramInPrimalMode() {
        final double[] tanks = new double[] { 8.0, 8.0, 2.0 };
        TEST_PROGRAM.execute(tanks, false);
        assertEquals(6.0, tanks[0], TOLERANCE);
        assertEquals(6.0, tanks[1], TOLERANCE);
        assertEquals(6.0, tanks[2], TOLERANCE);
    }

    @Test
    public final void runProgramInDualMode() {
        final double[] tanks = new double[] { 6.0, 6.0, 6.0 };
        TEST_PROGRAM.execute(tanks, true);
        assertEquals(8.0, tanks[0], TOLERANCE);
        assertEquals(8.0, tanks[1], TOLERANCE);
        assertEquals(2.0, tanks[2], TOLERANCE);
    }
}

```

Inside the constructor, we make a copy of the instructions and keep it safe. All instructions by themselves are immutable, including our Program composite element. Moreover, they are also stateless, thus making them an ideal candidate for concurrent execution (you need to take care to confine each tanks array to a single thread). Finally, here is the listing of the ProgramBuilder class together with unit tests (observe that the concrete implementations of the Executable interface are package private):

Listing of the ProgramBuilder Class

```
/**  
 * A utility class to build tank filling programs. The starting  
index  
 * for tanks is zero.  
 *  
 * @author ACME Developer  
 * @since 2.0  
 * @see Executable  
 */  
public final class ProgramBuilder {  
    private ArrayList<Executable> instructions = new ArrayList<>();  
  
    private ProgramBuilder() {}  
  
    /**  
     * Factory method to create a new builder instance. Each  
     * instance is responsible for a single program.  
     */  
    public static ProgramBuilder startProgram() {  
        return new ProgramBuilder();  
    }  
  
    /**  
     * Adds a new instruction to the current program.  
     *  
     * @param instruction a concrete instruction or a subprogram.  
     * @throws NullPointerException if the instruction is  
     * {@code null}.  
     * @throws IllegalArgumentException if the type of the  
     * instruction is unknown.  
     */  
    public void insertInstruction(Executable instruction) {  
        Objects.requireNonNull(instruction);  
        if (!(instruction instanceof Program ||  
              instruction instanceof MoveAbsolute ||  
              instruction instanceof MoveRelative)) {  
            throw new IllegalArgumentException(  
                "Unknown instruction type");  
        }  
        instructions.add(instruction);  
    }  
}
```

```
/**  
 * Gets the program containing all instructions added up so far.  
 *  
 *      @return an immutable program for the registered  
instructions.  
 */  
public Executable getProgram() {  
    return new Program(instructions);  
}  
  
/**  
 * Creates the move absolute instruction.  
 *  
 * @return an instruction to move a given amount of liquid from  
 * one tank into another.  
 * @param fromTank the index of the source tank.  
 * @param toTank the index of the target tank.  
 * @param amount the amount to move.  
 * @throws IllegalArgumentException if any argument is negative,  
 * or the tank indexes are out of bounds.  
 */  
public static Executable moveAbsolute(  
    int fromTank, int toTank, double amount) {  
    return new MoveAbsolute(fromTank, toTank, amount);  
}  
  
/**  
 * Creates the move relative instruction.  
 *  
 * @return an instruction to move a given percentage of liquid  
 * from one tank into another.  
 * @param fromTank the index of the source tank.  
 * @param toTank the index of the target tank.  
 * @param percentage the percentage of the amount to move.  
 * @throws IllegalArgumentException if any argument is negative,  
 * or the percentage is above 100, or the tank indexes are out of  
 * bounds.  
 */  
public static Executable moveRelative(  
    int fromTank, int toTank, int percentage) {  
    return new MoveRelative(fromTank, toTank, percentage);  
}  
}
```

Listing of the Matching Unit Test Suite

```
public class ProgramBuilderUnitTests {
    private static final double TOLERANCE = 0;
    private final ProgramBuilder builder =
        ProgramBuilder.startProgram();

    @Test(expected = IllegalArgumentException.class)
    public void tryAddingAnUnknownInstruction() {
        builder.insertInstruction(new Executable() {
            @Override
            public void execute(double[] tanks, boolean inDualMode)
        });
    }

    @Test
    public void createASimpleProgram() {
        builder.insertInstruction(moveRelative(0, 1, 50));
        builder.insertInstruction(moveRelative(1, 2, 50));
        builder.insertInstruction(moveRelative(2, 0, 25));
        final double[] tanks = new double[] { 8.0, 8.0, 2.0 };
        builder.getProgram().execute(tanks, false);
        assertEquals(6.0, tanks[0], TOLERANCE);
        assertEquals(6.0, tanks[1], TOLERANCE);
        assertEquals(6.0, tanks[2], TOLERANCE);
    }
}
```

The above `createASimpleProgram` unit test shows how easy it is to create a new tank filling program. The instructions are created with those utility methods that are statically imported here. Static imports may aid readability, and you should always keep an eye for an opportunity to utilize them. The programs are stored inside an in-memory registry (see the book's source code) that is initialized at startup with known programs (see Exercise 5). For now, we have two variants: basic and alternate. The program's name is the concatenation of the program type with the number of tanks (e.g., `basic3`). Our final task is to restructure the `TankFiller` main application class. Afterward, we may execute our acceptance test to ensure that everything is done according to specification (see the book's source code for the implementation of step definitions). Here is the listing of the reworked `TankFiller` class:

Listing of the New Version of the TankFiller Class

```
/**  
 * The TankFiller software as explained in chapter 7, PS 2.  
 *  
 * @author ACME Developer  
 * @since 1.0  
 */  
public class TankFiller {  
    public static final int MAX_TANKS = 100;  
    private final int numTanks;  
    private final double[] z;  
    private final Executable program;  
  
    /**  
     * Initializes the TankFiller calculator with input values.  
     *  
     * @param numTanks the number of tanks.  
     * @param programVariantName the name of the program type  
     * (for example, basic, alternate, etc.).  
     * @param z the final amounts to reach in each tank.  
     * @throws NullPointerException if the program name or the final  
     * amounts array is {@code null}.  
     * @throws IllegalArgumentException if the input is invalid.  
     */  
    public TankFiller(  
        int numTanks, String programVariantName, double[] z) {  
        Objects.requireNonNull(programVariantName);  
        Objects.requireNonNull(z);  
        if (numTanks < 2 || numTanks > MAX_TANKS) {  
            throw new IllegalArgumentException(  
                "The number of tanks must be in the range of [2,"  
                + MAX_TANKS + "]");  
        }  
        if (DoubleStream.of(z).anyMatch(x -> x < 0)) {  
            throw new IllegalArgumentException(  
                "The final amount of liquid cannot be a negative number");  
        }  
        if (z.length != numTanks) {  
            throw new IllegalArgumentException(  
                "The final amounts are incomplete");  
        }  
    }
```

```
        this.numTanks = numTanks;
        program =
            ProgramRegistry
                .getInstance()
                .getProgram(programVariantName + numTanks);
        this.z = z;
    }

/**
 * Performs the calculation for the given input.
 *
 * @return the initial amounts (the first tank's index is zero).
 */
public double[] calculate() {
    assert numTanks >= 2 && numTanks <= MAX_TANKS:
        "The number of tanks is out of range";
    assert program != null;
    assert z != null;
    assert z.length == numTanks;

    double[] tanks = z.clone();
    program.execute(tanks, true);
    return tanks;
}

/**
 * Main entry point into the program.
 *
 * @param args the input parameters: [number of tanks]
 * [program variant name]
 * [final amounts for tanks separated by colons].
 */
public static void main(String[] args) {
    if (args == null || args.length != 3) {
        System.err.println(
            "Usage: java -jar tank-filler-<version>.jar "
            + "<number of tanks> <program variant name> "
            + "<final amounts for tanks separated by colons>");
        return;
    }
    double[] tanks =
        new TankFiller(
            Integer.parseInt(args[0]),
            args[1],
            parseFinalAmounts(args[2])));
}
```

```
        .calculate();
    System.out.println(
        "Initial amounts for tanks:" + Arrays.toString(tanks));
}

private static double[] parseFinalAmounts(String amounts) {
    double[] parsedAmounts =
        Stream
            .of(amounts.split(","))
            .mapToDouble(s -> Double.parseDouble(s))
            .toArray();
    return parsedAmounts;
}
```

Notice that the `calculate` method is only commencing the appropriate program in dual mode. There are no concrete formulas to evaluate. The final amounts are an input; this is the reason for cloning the `z` array. All instructions of the tank filling program directly modify the input array. All in all, after increasing the version number to 2.0.0, the new reengineered tank filling program is ready for shipping.

7.5 Legacy Application: Problem

Suppose we receive the following legacy application with three mysterious methods (`puzzle1-3`) to analyze, redocument, and refactor [10]:

Listing of the Fictional Legacy Application

```
/**  
 * Legacy application with three mysterious methods.  
 */  
public class LegacyApp {  
    public static int puzzle1(int n) {  
        int p = 0, w = 1, s = n, f;  
  
        while (w <= n)  
            w <=> 2;  
        while (w != 1) {  
            w >=> 2;  
            f = p + w;  
            p >=> 1;  
            if (s >= f) {  
                s <=> f;  
            }  
        }  
        return s;  
    }  
}
```

```
        p += w;
        s -= f;
    }
}
return p;
}

public static int puzzle2(int x) {
    int a = 2, b = (int) (Math.pow(10, x) - 1), r = 0;

    for (int num = a; num <= b; num++) {
        int d = num;
        int y = 0;
        for (int j = 1; j <= x; j++) {
            y = 10 * y + d % 10;
            d /= 10;
        }
        r += num - y;
    }
    return r;
}

public static int puzzle3(int[] bytes) {
    int[] f = new int[255];
    int s = 0, k = 0;

    for (int b : bytes) {
        f[b]++;
    }
    do {
        s += f[k++];
    } while(s < bytes.length / 2);
    return k;
}

public static void main(String[] args) {
    if (args.length == 1) {
        switch(args[0]) {
            case "puzzle1": dumpPuzzle1IntoCSV(); break;
            case "puzzle2": dumpPuzzle2IntoCSV(); break;
            case "puzzle3": dumpPuzzle3IntoCSV(); break;
            default: System.err.println("Unknown puzzle");
        }
    } else {
        System.err.println("Provide the name of the puzzle");
    }
}
```

```

private static void dumpPuzzle1IntoCSV() {
    System.out.println("Input,Output");
    for (int i = 0; i <= 100; i++) {
        System.out.println(i + "," + puzzle1(i));
    }
}

private static void dumpPuzzle2IntoCSV() {
    System.out.println("Input,Output");
    for (int x = 1; x <= 7; x++) {
        System.out.println(x + "," + puzzle2(x));
    }
}

private static void dumpPuzzle3IntoCSV() {
    System.out.println("b1,b2,b3,b4,b5,b6,Output");
    int[] bytes = new int[] { 2, 2, 3, 3, 3, 3 };
    System.out.println("2,2,3,3,3," + puzzle3(bytes));
    bytes = new int[] { 2, 2, 2, 2, 3, 3 };
    System.out.println("2,2,2,2,3," + puzzle3(bytes));
    bytes = new int[] { 2, 2, 2, 4, 4, 4 };
    System.out.println("2,2,2,4,4," + puzzle3(bytes));
}
}

```

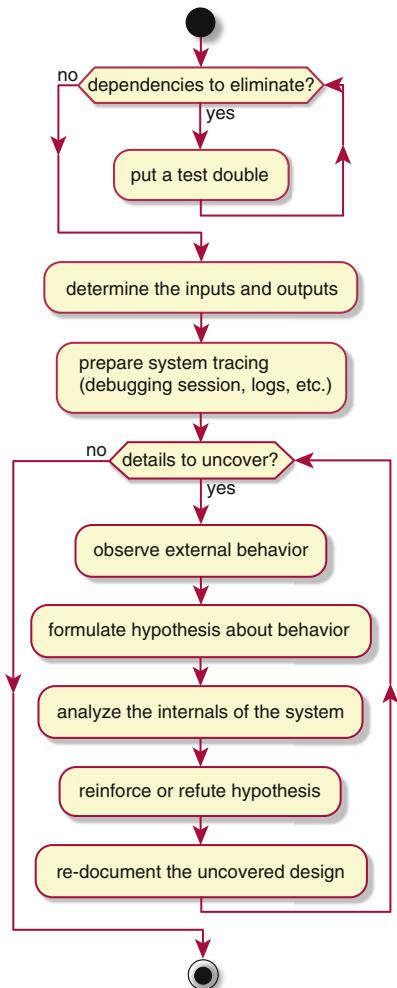
Our task is to provide back a more readable and comprehensible version of the software (see Exercise 6).

7.6 Legacy Application: Resolution

We will tackle the three methods in sequential order, starting with `puzzle1`. Nonetheless, there are some commonalities that we will discuss first. Figure 7.4 illustrates the flow for solving each task. Note that more complex situations would boil down to this general flow. The only real difference would be the granularity of elements participating in the activity (e.g., instead of methods, you would encounter subsystems and try to decipher that they do).

Setting Up a Debugging Session I recommend having your debugger ready to trace the execution. All modern IDEs come with a sophisticated debugger. I will use Eclipse IDE for Java developers (Oxygen Release 4.7.0) on Mac OS X. The basic steps are listed below (I am intentionally avoiding fancy shortcuts to make the elaboration straightforward):

Fig. 7.4 The UML activity diagram that shows the principal reverse engineering flow for uncovering the secrets behind a software system. Notice that the process is iterative. We cannot scoop all the stuff at once



1. Position the cursor on the first line of the matching method, and set a breakpoint on it via the *Run → Toggle Breakpoint* menu item.
2. Create a call inside `main` to trigger your method (e.g., `new App().puzzle1(13)`).
3. Run the application inside the debugger via the *Run → Debug As → Java Application* menu item.
4. The application will stop at the line with the breakpoint.

5. Choose the *Window → Show View → Other...* menu item and select the *Display* and *Expressions* from the dialog box, as depicted in Fig. 7.5.
6. Inside the Display window, you may alter the variables or execute arbitrary Java statements in the context of the stopped method. One handy way to examine what is happening is to modify the arguments of the method. When you enter `n=19;` (for the `puzzle1` method) and highlight the previous expression, after pressing `Cmd + Shift + D`, the argument `n` will be set to 19.

In the Expressions window, you can arrange what expressions to trace as you step through your code. You may want to track all the internal variables. As you execute your code line by line, using the F5–F8 keys, the values will change instantaneously.

Besides watching the internal state of the software, it is also useful to turn on debug logs. Logs may provide a complementary insight into the behavior. I will talk more about logging in Chap. 9.

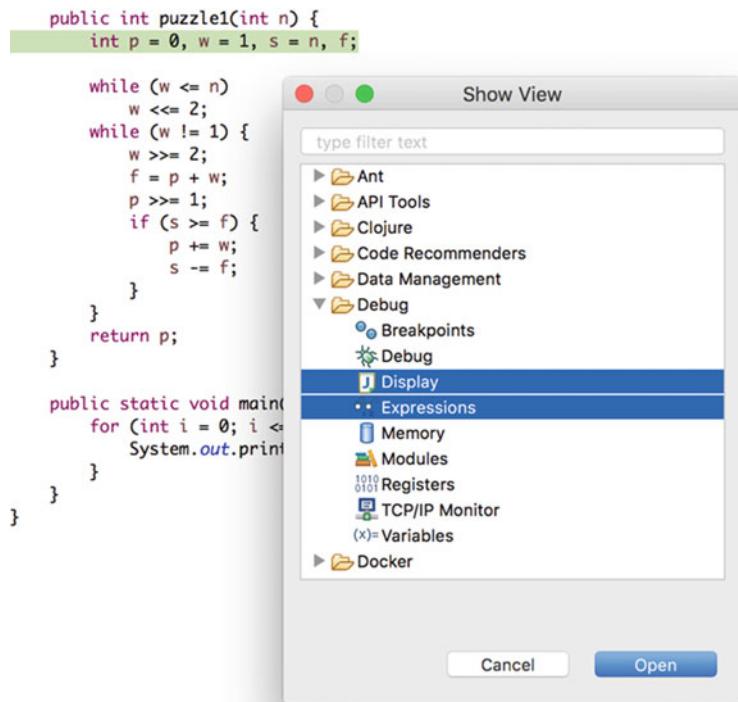


Fig. 7.5 The *Show View* dialog box with selected items relevant for our case study. The background shows the `puzzle1` method with the first line highlighted (this is where execution has stopped due to a breakpoint)

Examining the External Behavior The first thing you should do is to run the corresponding method with various inputs. The generated outputs will hopefully reveal certain aspects of the behavior. For example, calling `puzzle1` from inside a loop will quickly unravel its purpose. Of course, knowing what the method does from outside is just the beginning of the reverse engineering endeavor. You must also comprehend the method's internals. However, knowing what the method is doing is of a great help while examining the routine's logic (see Exercise 7). So, you should attempt to get a clue by looking at the output first. You will also discern that sometimes, as in the case of `puzzle1` and `puzzle2`, a seemingly rapid progress at the beginning doesn't mean an easy job later.

Reverse Engineering `puzzle1` If we start our legacy application with the `puzzle3` command line argument, then it will produce the following output (it is reduced to emphasize only the important parts):

```
Input,Output  
0,0  
1,1  
2,1  
3,1  
4,2  
5,2  
...  
8,2  
9,3  
10,3  
...  
15,3  
16,4  
17,4  
...  
23,4  
24,4  
25,5  
26,5  
...
```

It isn't hard to deduce that it outputs the number k , such that the input argument falls in the range of $[k^2, (k+1)^2]$. For example, if $n = 13$, then $k = 3$ because 13 is in the range of $[3^2, 4^2]$. A trivial implementation may look like as follows:

```
public static int isqrt(int n) {  
    return (int) Math.sqrt(n);  
}
```

Why is the legacy code so complicated, compared to this straightforward solution? What does it really do?

Finding the Asymptotic Running Time of `puzzle1` You should notice that the second loop is executing the same number of times as the first one. The first loop multiplies `w` by 4 until it becomes larger than `n`, and the second one divides `w` by 4 in each iteration until it reaches 1. These operations are expressed as bit shifts. Shifting 2 bits to the left is the same as multiplying by 4, and shifting 2 bits to the right is equivalent to dividing by 4. All in all, the run-time complexity is $O(\log_4 n)$, which is usually written just as $O(\log n)$.

How fast is the standard `sqrt` function? This insight will be valuable while analyzing the `puzzle1` method. `Math.sqrt` internally uses a fast-bitwise algorithm and applies the primal-dual paradigm. It first converts the `double` argument into the matching `long` representation (primal problem → dual problem), performs the dual calculation bit-by-bit in $O(\log n)$ steps, and converts back the `long` result into `double` (dual solution → primal solution). Except these transformations, the core algorithm is like `puzzle1`.

Sometimes, the legacy code contains highly optimized routines that were not part of the language's standard library in the past. The `puzzle1` method shows a striking difference between a clean `isqrt` solution and an extremely optimized variant of it. Performance optimizations thwart maintainability, and over time, many custom optimizations become obsolete as standard libraries get more powerful. Therefore, it is worthwhile to revisit them and provide a fast and tidy alternative. Hiding complexity inside a library/framework is a win-win situation; both performance and maintainability are improved at the same time.

Sure, `puzzle1` is still faster than our `isqrt`, but is the difference still that huge as it was in the past? Of course, even if you leave the code as it is, you must redocument your conclusions. There is no sense for someone else to repeat this same process.

Reverse Engineering `puzzle2` If we start our legacy application with the `puzzle2` command line argument, then it will spill out the following output:

```
Input,Output
1,0
2,9
3,99
4,999
5,9999
6,99999
7,999999
```

If we denote the input value with N , then it looks like the output is $10^{N-1} - 1$. For example, $N = 2$ produces $10^1 - 1 = 9$. Nonetheless, the internals of the method suggest that a much more convoluted process is involved in the background. This method is ideal for an interactive debugging session. You should only watch what

happens with $N = 2$ for a couple of numbers (e.g., 2, 3, and 4). You will see that the variable `r` accumulates the difference between the two-digit number and its opposite form (when the digits are flipped). For example, if the current number is 3, then the difference will be $03 - 30 = -27$. However, it is still an enigma why it generates that result.

This is where visualization and rearrangement of data may help. Let us picturize the table for $N = 2$:

	02	03	04	05	06	07	08	09
10	11	12	13	14	15	16	18	19
20	21	22	23	24	25	26	27	29
30	31	32	33	34	35	36	37	39
40	41	42	43	44	45	46	47	49
50	51	52	53	54	55	56	57	59
60	61	62	63	64	65	66	67	69
70	71	72	73	74	75	76	77	79
80	81	82	83	84	85	86	87	89
90	91	92	93	94	95	96	97	99

All the diagonal elements cancel themselves out (I've decorated only the main diagonal), including the solitary 99. Remember that we start iterating from 02, which means 01 is missing. Therefore, 10 remains as the sole nonzero element. The result is $10 - 01 = 9$. It is easy to extend this for larger tables. For example, if $N = 3$, then only 100 will give a nonzero value ($100 - 001 = 99$). In other words, each number xyz will have its corresponding anti-pair zyx except 100. We can now simplify `puzzle2` and redocument it accordingly. All this proves behavior is crucial, as we cannot just assume that our observation is valid for all cases. We must truly understand what is going on before reshaping the legacy code.

Reverse Engineering `puzzle3` If we start our legacy application with the `puzzle3` command line argument, then it will produce the following output:

```
b1,b2,b3,b4,b5,b6,Output
2,2,3,3,3,4
2,2,2,2,3,3,3
2,2,2,4,4,4,3
1,2,2,2,3,14,3
```

The input argument is defined as `int[] bytes`, which indicates that the method expects an array of bytes. I've seen this kind of “ingenuity” in code bases. Java's built-in `byte` data type is a signed value, and here the author wanted unsigned bytes. This assumption is reinforced by the definition of the internal array `f`. Frequently, you need to collect multiple evidences spread out in the legacy code to understand some trait of the system. You may even feel that you are in the role of a software detective.

Our next detective assignment is to find out whether the numbers in the input comprise a set or a multiset (bag). We can conclude that it is the latter, as the first loop counts the frequencies of occurrence of numbers. Thus, $f[k]$ will contain the repetition count of k in the multiset. Obviously, based upon the counting logic, the input array doesn't need to be sorted.

The variable s holds the sum of occurrences, and k is the index into the bins (represented as f). It seems that the method tries to find the bin, which is in the middle of the distribution. However, looking at the output, this statement isn't quite correct. The first line shows 4, which is bigger than any of the input numbers. On the other hand, the third line refers to 3, which isn't a member of the multiset. When you cannot reconcile your hypothesis by looking at the output and the source code, this might indicate a defect in the code. In this case, your hypothesis reflects the intention of the original developer, rather than the state of affairs of the implementation.

What would happen if we would switch $k++$ with $++k$? It is ok to experiment with the code in an isolated fashion (meaning that you will do that in a separate private branch and take care not to push it back into the central repository). The new output is shown below:

```
b1,b2,b3,b4,b5,b6,Output  
2,2,3,3,3,3,3  
2,2,2,2,3,3,2  
2,2,2,4,4,4,2  
1,2,2,2,3,14,2
```

This makes more sense. Now, our initial postulation starts to solidify. However, something is still fishy. The output cannot support the following presuppositions:

- The output is the nearest existing number k , such that the quantity of numbers less than k is closest to the quantity of numbers greater or equal to k .
- The output is the nearest existing number k , such that the quantity of numbers less than or equal to k is closest to the quantity of numbers greater than k .

We have reached the dead end (see Exercise 8). We can never resolve the above mystery without seeking out more information. At this point, we can try to “fix” the code by choosing one alternative, but further investigation is necessary to reassure us that we are on the right path. Whatever you choose at this moment, you should thoroughly clarify and make a note of what other checks must be done.

What are we doing by changing $k++$ to $++k$ or performing what Exercise 8 suggests? Well, we try to eliminate the most rudimentary coding errors. If you think about this, we are manually simulating what mutation testing tools are doing for us automatically. They randomly induce modifications into the code base and examine whether our tests will catch those mutations. This technique belongs to the statistical testing category, which may give some prediction about the reliability.

This puzzle³ has emphasized the fact that without tests and documentation (including requirements and design descriptions), we cannot always succeed in reverse engineering everything from the source code. The reality is that we rarely encounter a bug-free system. Once we stumble across a bug, the source code stops to be a bulletproof specification of the behavior.

7.7 Restructure of the System: Problem

The EDA system had been established as a standard among users, and the ACME Corporation has decided to initiate a restructuring endeavor. They would like, by default, to make the matrix multiplication multithreaded for mutable matrices. These are anyhow not shared between threads, so the concurrent execution should happen under controlled conditions. The idea is to internally implement multiplication in parallel fashion while still retaining the illusion that the operation is synchronous. The intention is to apply the Half Sync-Half Async pattern in the matrix library. They also demand that all current tests must pass without modification. They just want a pure speed improvement. The ultrafast GPU version will be utilized on some selected platforms.

7.8 Restructure of the System: Resolution

Thanks to the previous improvements that had been made to the matrix library, the developer of the ACME Company is in a good position to accomplish the job smoothly. He has decided to follow the so-called *bag of tasks* [11] approach to parallel computing. It is a straightforward model, where tasks are units of work that are concurrently executed. The shared logic is associated with different data sections, and each task is responsible for a chunk of data. In matrix multiplication, we may map each task to calculate one row of the target matrix. If we denote our multiplication as $C = A * B$, then a task will calculate the elements of $C[\text{row}, *]$. The number of tasks is thus fixed and equal to the number of rows of the target matrix. This number may be larger than the available cores in the CPU. Luckily, he will not need to deal with low-level details, as Java comes with a sophisticated Fork/Join framework for writing parallel programs [12].

The Fork/Join framework is ideal for compute intensive tasks, that don't do any I/O or locking. It is especially useful when you can express the algorithm in a recursive *divide and conquer* fashion. Matrix multiplication, as we have described above, has a rather simple execution plan, so the bare Java Executor service would do as well. However, if you anticipate a mixed situation (with simple and graph/tree-based execution structures), then using a single unifying framework enhances the conceptual integrity of the system. The Fork/Join shines even for simple execution plans, though.

For matrix multiplication, the developer has created Fork/Join tasks that implement the `RecursiveAction` API. Such tasks don't yield a return value on the

`join()` call, but rather directly operate on the shared state (the target matrix in this case) in a lock-free manner. Below is the listing of the restructured `DenseRealMatrix` class:

Listing of the Patch to the `DenseRealMatrix` Class

```
final class DenseRealMatrix extends RealAbstractMatrix {
    private final ForkJoinPool forkJoinPool = new ForkJoinPool();
    ...
    @SuppressWarnings("serial")
    private class VectorMultiplicationTask extends RecursiveAction
    {
        private final AbstractMatrix<Double> target;
        private final Matrix<Double> other;
        private final int row;

        VectorMultiplicationTask(
            int row,
            AbstractMatrix<Double> target,
            Matrix<Double> other) {
            this.target = target;
            this.other = other;
            this.row = row;
        }

        @Override
        protected void compute() {
            for (int column = 0; column < other.getColumns();
                column++) {
                double sum = 0.0;
                for (int k = 0; k < getColumns(); k++) {
                    sum += multiply(
                        getElement(row, k),
                        other.getElement(k, column));
                }
                target.set(row, column, sum);
            }
        }
    }

    @SuppressWarnings("serial")
    private class MatrixMultiplicationTask extends RecursiveAction
    {
```

```

        private final AbstractMatrix<Double> target;
        private final Matrix<Double> other;

        MatrixMultiplicationTask(AbstractMatrix<Double> target,
            Matrix<Double> other) {
            this.target = target;
            this.other = other;
        }

        @Override
        protected void compute() {
            List<VectorMultiplicationTask> tasks = new ArrayList<>();
            for (int row = 0; row < getRows(); row++) {
                tasks.add(new VectorMultiplicationTask(
                    row, target, other));
            }
            invokeAll(tasks);
        }
    }

    @Override
    public Matrix<Double> multiply(Matrix<Double> other) {
        Objects.requireNonNull(other);
        if (other.getRows() != getColumnCount()) {
            throw new IllegalArgumentException(
                "The dimensions of matrices don't match");
        }

        final AbstractMatrix<Double> result =
            newMatrix(getRows(), other.getColumns());
        forkJoinPool.invoke(
            new MatrixMultiplicationTask(result, other));
        return result;
    }
}

```

The implementation is intuitive and nicely fits into the existing matrix library. The `ForkJoinPool` is created by invoking the default constructor. It will configure the pool to match the number of CPU cores. The `MatrixMultiplicationTask` task creates as many subordinate tasks as there are rows in the target matrix. The top-level `invoke` call on the pool is a blocking call, which is the desired behavior. So, the ACME developer has successfully implemented the Half Sync-Half Async pattern and considerably increased the performance of the matrix multiplication (see Exercise 9). The Fork/Join

framework linearly scales with the number of available hardware processing units. This example also emphasized the fact that reengineering is highly supported by a clean architecture.

7.9 Summary

Reverse engineering and reengineering are paired activities. The reverse engineering phase recovers hidden details in the system that are essential inputs to further restructuring jobs. On the other hand, a restructured system is more susceptible to having its inner details revealed. Refactoring is a continuous activity to enhance maintainability, and redocumentation is crucial to avoid misunderstandings and repetitive “excavation” work. The latter is especially important since it isn’t always possible to uncover everything from the source code, as we have witnessed.

Reverse engineering also occurs when incompatible software teams are working together on the product. If an eXtreme Programming (XP) development team hands over the code base to a maintenance team, then the missing documentation must be recovered from the code base. This extra gap can plague further evolution for a pretty long time. Due to misaligned practices, software engineers may also develop barriers pertaining to refactoring that could further complicate matters [13].

The key takeaways from this chapter are listed below:

- Readability should be the most important facet of the code base. A readable code speaks for itself and doesn’t require extensive documentation. The code should always be shaped as if it has just gone through a meticulous reverse engineering + refactoring foray.
- The code will rot over time due to constant changes that damage its structure. Reengineering is thus necessary from time to time to bring it back into an evolvable form.
- If the code base isn’t any more amenable for reengineering, then it should be slowly retired and replaced by a new generation of software.

7.10 Exercises

1. Prove the correctness of the `calculateForManyTanks` method. (Hint: Using mathematical induction, you should analyze what happens when a new tank is inserted at the third from last position with an initial value, `z`.)
2. Extend the test suite to also vary the final amount `Z`.
3. Improve the specification by replacing the generic phrase “the application should show an error” with more concrete variants. Sure, you will have to implement the associated step definitions too.
4. Implement the corresponding `MoveRelativeUnitTest` similarly as we have done for the `MoveAbsolute` instruction.

5. (Project assignment) Implement external storage for tank filling mini-programs. The program builder facility would need to provide an ability to save programs on disk. At startup, based upon the program's name, the system would load the corresponding program from disk. The program serialization format may vary, and you should explain your choice.
6. Sometimes, the legacy system is written in an archaic language. Suppose the following TURBO PASCAL 5.5 program is given to you to reverse engineer it [10]:

```
program Trace;
uses Crt;
type
  Sub = procedure(b: boolean);
var
  i: integer;

(*$F+*)

procedure OutB(b: boolean);
begin
  writeln(b)
end;

procedure Reset(b: boolean);
begin
  b := false
end;

(*$F-*)

procedure Puzzle(b: boolean; q: Sub);
begin
  inc(i);
  if b then begin
    Puzzle(false, OutB);
    OutB(b);
    q(b)
  end else begin
    OutB(b);
    if i < 3 then begin
      Puzzle(b, q);
      Puzzle(true, OutB)
    end;
  end;
end;
```

```

q(b)
end;
dec(i)
end;

begin
  ClrScr;
  i := 0;
  Puzzle(true, Reset);
  repeat until KeyPressed
end.

```

To solve this conundrum, you will need to learn TURBO PASCAL 5.5. Moreover, you will also need to decipher how to install TURBO PASCAL 5.5 onto your modern system. As a starting point, you should visit www.schoolfreeware.com/Freeware_Turbo_Pascal_Download.html. I recommend the DOSBox utility (www.dosbox.com); this is how I managed to install TURBO PASCAL 5.5 on Mac OS X and run it properly (there are a couple of annoyances with TP 5.5 installation, but these should be part of the challenge in this exercise). Once you have accomplished the deployment, follow the flow from Fig. 7.4. You should try to figure out the output even without running the program. This would be a good exercise for manually tracing the procedure call stack.

7. Timing information is another superb side-channel for reverse engineering a system. Embellish the preparatory step from Fig. 7.4 to measure the methods' execution times. Based upon these data, you may formulate hypotheses about asymptotic running times. Afterward, you should try to reinforce your finding.
8. Try also to alter the condition (with `k++` and `++k`) in the loop from `s < bytes.length / 2` to `s <= bytes.length / 2`. What would you get now? Are the new outputs helpful?
9. Implement a similar concurrent matrix addition. Moreover, refactor the code to move out the generic concurrency stuff from the `DenseRealMatrix` class into the `RealAbstractMatrix` class.

References

Further Reading

1. Eilam E (2005) Reversing: secrets of reverse engineering. Wiley, Hoboken, NJ. This book is the primer for various reverse engineering techniques, covering a broad range of domains and platforms. It has a separate section about security related reverse engineering. It also covers the topic of cracking different protection mechanisms
2. Fowler M, Beck K, Brant J, Opdyke W, Roberts D (1999) Refactoring: improving the design of existing code. Addison-Wesley, Upper Saddle River, NJ. This is a reference book about

refactoring. Most modern IDEs already implement many techniques from this book. Refactoring is an important technique to learn, as it helps you make controllable incremental improvements to the code base to boost its maintainability.

3. Kerievsky J (2005) Refactoring to patterns, 1st edn. Addison-Wesley, Boston, MA. This is another excellent book about refactoring. It shows you real code samples and a series of refactoring steps to make them adhere to known design patterns. In this manner, a code becomes intuitive, readable and more structured

Regular Bibliographic References

4. Đuković M, Varga E (2015) Load profile-based efficiency metrics for code obfuscators. *Acta Polytech Hung* 12(5):191–212. ISSN:1785-8860
5. Penzenstadler B, Raturi A, Richardson D, Tomlinson B (2014) Safety, security, now sustainability: the nonfunctional requirement for the 21st century. *IEEE Softw* 31:40–47
6. Liu H, Liu Y, Xue G, Gao Y (2013) Case study on software refactoring tactics. *IET Softw* 8:1–11
7. Pérez-Castillo R, Piattini M (2014) Analyzing the harmful effect of god class refactoring on power consumption. *IEEE Softw* 31:48–54
8. JJUnit Home Page. qare.sourceforge.net/web/2001-12/products/jxunit/index.html. Accessed 17 Sept 2017
9. Sarcar V (2016) Java design patterns: a tour of 23 gang of four design patterns in Java. Apress, New York, NY
10. Varga E (1990) Zbirka Rešenih Zadataka iz Programiranja u Programskom Jeziku TURBO PASCAL 5.5. Savremena Administracija, Belgrade
11. Andrews RG (2000) Foundations of multithreaded, parallel, and distributed programming. Addison-Wesley, Reading, MA
12. Ponge J (2011) Fork and join: Java can excel at painless parallel programming too! www.oracle.com/technetwork/articles/java/fork-join-422606.html. Accessed 20 Sept 2017
13. Tempore E, Gorscak T, Angelis L (2017) Barriers to refactoring. *Commun ACM* 60(10): 54–61

Part IV

DevOps

Coupling as a notion has many sides (common, data, control, environmental, etc.); thus, dozens of metrics are associated with this concept. From the perspective of automated deployment/delivery, we are most interested in differentiating between logical and physical coupling. This variance is tightly interrelated with the level of reuse, as a desired goal is to make a unit of reuse also a unit of deployment. Sure, logical coupling is a prerequisite to attain the latter, but they aren't the same. In a typical OO framework, you cannot simply pick a couple of classes, leaving out the rest. You must bring in the whole framework. Packaging only helps to organize the clutter, but the amount of imported stuff doesn't change. The SOA (especially in the form of microservices) tries to deliver the promise of being able to reuse individual services without pulling in the whole distributed system. We will revisit what are the preconditions to attain physically loosely coupled solutions and what are the benefits.

You have already seen an example of a microservice in the previous chapters (recall the Swaggerize service). It embodies many elements associated with the microservices paradigm (see [1] for more details about this concept). Moreover, you have observed the difference between a loosely coupled design and deployment (a term coined in [2]). Although the existent ACME services were loosely coupled in design space, the Swagger User Interface (UI) was logically isolated from the rest of the code base, ACME had failed to achieve a vision of loosely coupled deployment. The Swagger UI was shipped together with each service, and all of them looked like monoliths. When we will have extracted the Swagger UI engine and wrapped it inside our new service, we will have moved toward the ideal of having loosely coupled deployment.

There are many reasons for falling into the all-or-nothing release model trap:

- A gigantic product is artificially cut into pieces (each embodied as a service) without considering the business boundaries. When a monolith is smashed into a multitude of small services by only considering technical boundaries, the outcome is much worse than the original situation. The small pieces still need to be

managed and deployed in a lock-step fashion. Synchronizing a myriad of small interrelated services and, probably, development teams is hard. No technology or tooling would help remedy the problem.

- Clients fully depend on the provider's API, even though they use only small subsets of that API. This can happen easily when a client just autogenerates the client side code based upon the service descriptor. Any unrelated (from the client's perspective) change in the API would impact the client. In this case, the service descriptor could be a Resource API description or a message schema. This one of the reason why a RESTful API, which is inherently hypermedia driven, doesn't rely on service descriptors nor code generation.
- Lack of APIs. This might sound ridiculous, but often applications are built with implicit APIs. In other words, the API is an emergent property of the system that just happens to be as it is in any given moment in time. In this scenario, services may depend on hidden assumptions about the behavior and structure of other services. Often, side-channels are created for communication purposes, where one service passes data to another one via some mediator (frequently, this middle service isn't even aware of such data tunneling). Any implementation change in any of the services could easily blow up the whole system. Therefore, they must be treated in a unified manner and deployed together. In this setup, versioning of individual elements is purely a formalism.
- Lack of architecture. When systems are built in an ad hoc style (see Chap. 5), no one is responsible for attaining relevant quality attributes. Furthermore, there is even a lack of consensus as to what are those attributes that the system should care about. Teams implement stuff as they judge appropriate. Multiple isolated selfish decisions rarely happen to align nicely. A corollary is a huge ball of mud (or a huge pile of small muds lumped together), which is cumbersome to administer.

The benefits of having a loosely coupled deployment (together with an API-centric architecture-based approach) are manifold:

- Boosts reuse. Each service or component is reusable independently of the other artifacts. This allows better structuring of systems, lessens the maintenance burden by importing only what is necessary, and increases dependability. The latter is again related to the number of incorporated elements. You can more easily manage security, availability, and resiliency when you do know what you're bringing in with each reuse.
- Enables versioning. Versioning is meaningful when you can govern each service or component in isolation. This is tightly associated with configuration management, where you can establish rules on what combination of elements are cooperative.
- Aids evolution. Knowing that elements are independent helps a lot in impact analysis. You can come up with a more accurate estimate of the amount of required work when you're aware of the physical boundaries in the system. You

can speed up the request–impact analysis–implementation–delivery cycle when elements are individually manageable.

- Allows distributed development. An individual service or component with a well-defined API can be implemented by a separate team. The teams may have their own way of working and pace. The coordination of teams is possible since they interact via APIs and deliver only their piece of software. A product manager has full control over the final product by taking the role of an assembler of separate compatible software elements.

8.1 Reuse: Problem

The EDA system had been spreading over complex domains, and there was a need to introduce more convoluted matrix types (complex, user-defined, etc.). There was a need to speed up evolution of the matrix library. Nonetheless, the current project wasn't good enough for parallel development, even though it was logically properly decoupled, as teams had collided over the single code base. The company has requested a restructuring of the project to introduce physically independent parts.

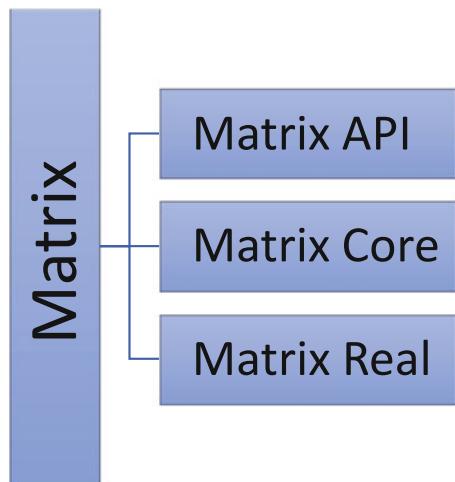
8.2 Reuse: Resolution

Independent reuse doesn't necessarily imply a distributed system. The OSGi component model has a notion of μ Services that are components running inside the same address space as the application. We will implement each matrix type as a sovereign entity using a more lightweight technique. Different parts of the project will produce reusable jar files that could be separately imported. The goal is to enable various teams to work in parallel without collision on those parts. As this is a compatible modification of the library from a client's viewpoint, we will bump the version number to 3.2.0.

We must restructure our current Maven project into a multimodule Maven project. This is a fine example that proves restructuring may also apply to other types of artifacts, not only to source code. Here are the steps that we must accomplish:

- We must change the parent's project packaging from jar to pom. The packaging is defined inside the `pom.xml` file.
- Next, we need to use the Eclipse IDE (or your favorite one) and select *Maven → New Maven Module Project* from the drop-down menu (first you need to select the parent project). We must repeat this for all our modules, as depicted in Fig. 8.1.
- For each module, we must move the relevant entities from the parent project into the module. For example, the `matrix-api` module will only contain the `Matrix` interface.

Fig. 8.1 The new organization of the matrix library project with multiple modules. The dependency relationship goes from Matrix Real toward Matrix Core and finally to Matrix API



- We must install each module (invoke `mvn install` on the parent), as higher-level modules will import the lower-level ones.
- Finally, we must ensure that our overarching integration tests (that we have inherited from the previous version of the project) execute properly. To run the tests from Eclipse IDE, you will need to add the subordinate projects (module projects) to your parent project's build path. Select *Properties* from the project's context menu, then on the *Java Build Path* dialog box, choose the *Projects* tab.

The nice side effect of this restructuring is that clients should also be able to selectively import modules (see Exercise 1). We have seen how this works with Dropwizard, where we have imported stuff on demand.

8.3 Summary

The previous case study has illustrated the fact that loose coupling in design is the necessity to achieve loose coupling in deployment, i.e., logical coupling should be settled before thinking about the next step. As usual, reaching the destination is an iterative and incremental feat. Compare the initial version of our matrix library with the one above. The difference is tremendous.

An effective way to control the level of coupling is the application of consumer-driven contracts (we have already talked about this earlier). These contracts may serve as a good indicator of how much of the API is consumed by a specific client. A set of these agreements may be a useful hint of how to restructure the API to increase the alignment between demand and offering. Moreover, the highly utilized parts of the API should gain most of our attention. The key takeaways from this chapter are as follows:

- Modularity embraces both logical and physical aspects of coupling and helps reaching the desired balance. This is the reason why it is so fundamental as a concept in software engineering.
- Various couplings in a final product may reflect organizational, developmental, and operational demeanors. It isn't surprising then to see when the software structure sometimes mirrors the hierarchical buildup of an organization. In this case, to achieve loose coupling in the product, you may want to first start with reorganizing the company itself.
- Plain Old Java Objects (POJOs) are attractive since they don't bear a strong environmental coupling toward some underlying framework or platform. Again, whether such coupling is an issue is dictated by pertinent quality attributes, such as portability.

8.4 Exercises

1. There is one caveat in the code base, which waits your additional restructuring. The `MatrixFactory` class is still sitting in the top-level module (`matrix-real`). This is unfortunate since it is part of the matrix library's public API. On the other hand, it cannot be simply moved into the `matrix-api` module, as that would introduce a circular dependency between modules. Try to solve this problem so that we eschew duplication when reshaping the other matrix types (part of earlier exercises) into this new model. (Hint: Think about applying the *Abstract Factory* pattern together with dependency injection.)

References

Further Reading

1. Wolff E (2016) Microservices: flexible software architecture. Addison-Wesley Professional, Boston, MA. This book dives deeply into the crux of the SOA and micro-services paradigms in a technology neutral way. It explains the importance of having clear business boundaries, and how domain-driven design fits into the picture. This book is a prerequisite to understand the myriad of existent technologies and tools related to micro-services

Regular Bibliographic References

2. Pardon G (2008) Loosely-coupled deployment vs. loosely-coupled design. www.atomikos.com/Blog/LooselyCoupledDeploymentVsLooselyCoupledDesign. Accessed 10 Sep 2017

In production, you must have visibility of what is going on within your system. Part of the DevOps culture is to share the burden of providing such supervising capability with both development and operations teams. In parallel to shipping metrics and log messages to the central location, you must also have facilities to trigger alarms if something goes wrong. An alarm may further instigate messaging of relevant parties or taking autonomous actions to remedy the issue. Most sophisticated monitoring tools (Zenoss, Ganglia, Nagios, Riemann, Graphite, etc.) do have an ability to configure alarms, specify actions, and generate reports based on the collected data. Logs also contain valuable information, and many business intelligence tools could use them to synthesize high-level reports or initiate actions. Again, there are many viable options regarding tools (Elasticsearch/Logstash/Kibana, LogPoint, Loggly, etc.). Of course, going into great detail on any of these tools is surely out of the scope of this book. Hence, this chapter will primarily illustrate how to properly collect/send metrics in your application and how to structure/send logs toward the appropriate infrastructure node. The aim is to showcase techniques of squeezing out the maximum insight with minimal effort (expressed in data collection, transmission, and processing time as well as cost). Monitoring and logging must be judiciously managed; it isn't about simply choosing "the best" monitoring/logging tool, starting to insanely collect whatever is possible on nodes, and blindly sending them over to a central place. The job isn't done when you're entirely flooded with data; the job is done when you have clear goals regarding the data and you have gathered enough data to accomplish your objectives.

9.1 Collecting the Data

This is the first step in the data handling journey. If you make an error here, then all the rest (interpreting, decision-making, etc.) is going to be negatively impacted (like requirement defects in software). To make this elaboration more concrete,

I will use an example pertaining to the performance measurement of a Java program running on the HotSpot Java Virtual Machine (JVM).

To reliably get performance metrics, you must know what is happening under the hood. The Java compiler first translates the Java source code into a Java byte code amenable to direct execution. Nonetheless, interpreting this byte code is much slower than running the equivalent machine code. This is where one of the components of the HotSpot JVM called the Just-in-Time compiler kicks in. It traces the Java code execution and selectively compiles pertinent methods on the fly into machine code. This default execution regime of a Java program is known as *mixed* mode execution. The major benefit of this approach is that the Java program is optimized by the compiler, depending on how it is used. In C++, you must decide in advance what compiler optimizations to turn on. The JVM's strategy is a fine example of delaying decisions when you do have more information to make a better resolution. In the case of C++, you base your optimization strategy on a set of hypotheses (expectations) about future usage. The JVM optimizes the code based upon concrete measured facts. Therefore, you may even outperform a machine-compiled code with a JVM.

Of course, there's no such thing as a free lunch. The JVM does require some time to collect enough data to be able to properly decide what and how to translate into machine code. If you don't give a chance for the JVM to "warm up," then the collected performance data would be misleading. It will falsely indicate performance issues. Acting upon such wrong data is problematic since needless performance optimizations hamper maintainability. So, you must be knowledgeable about these inner details, and before starting the measurement, you need to do a JVM warm up session (run your app for a while in an intended manner dictated by use cases).

Another mandatory factor to understand is the difference between the two major VMs: *client* and *server*. I've seen many times long-running background services started up under Client VM (-client option on the command line) instead of Server VM (-server option on the command line). The distinction is important in the aspect of compilation too. The Client VM is more aggressive in compiling a section of code to boost usability and perceived performance of the client application. The Server VM is more conservative and requires, by default, 10,000 interpreted method invocations before considering compilation. The Client VM needs 10× less.

I've seen "smart" solutions to reduce the precompilation period by fiddling with the -Xcomp and/or -XX:CompileThreshold options. The former immediately forces the VM to compile the method, while the latter sets the threshold related to the number of interpreted method invocations before compilation. Rashly lowering this parameter would cause damage more than it creates benefits. Without enough time, the VM would lack the required volume of data to conduct an efficient compilation.

All in all, these contemplations are here to remind you about the level of detail needed to attain an effective data collection plan. The above scenario equally applies in performance monitoring a production system as well as for optimizing

it during development/maintenance. The main message is that data acquisition (as everything else in software engineering) must be carefully devised and fulfilled (see Exercise 2).

9.2 Interpreting the Collected Data

Gathering huge amounts of data is only the beginning. You must understand how to correctly analyze the content, find patterns, and infer new relationships. One powerful principle that you should keep in mind is Bonferroni's principle [2]. If your algorithm to identify and classify data finds more “interesting” details than you would expect under normal circumstances, then it might indicate a wrong data mining approach. Without being knowledgeable about this phenomenon, you could produce too many false negatives or false positives. Acting upon them isn't a wise tactic. There is nothing more annoying and detrimental in system monitoring than frequent false alarms (see Exercise 1).

The cornerstone principle in data analysis is that *correlation does not imply causation!* Many embarrassing errors had been made in the past even by scientists. If you notice a correlation between some variables (i.e., if one increases its value, the other follows the same tendency), then you should resist the temptation to conclude that there are some cause/effect mechanisms in the background. This is especially important to remember during performance optimizations. Many hidden reasons may cause a slowdown or speedup that aren't included in your observation.

9.2.1 Devious Reinforcements

Often, the acquired data are used as a mechanism to reinforce our beliefs, instead of a technique for pursuing impartiality. People tend to select only plausible facts while neglecting the conflicting ones. The following example demonstrates how things can go astray when an initial false conviction is buttressed by equally bad tests (aiming to prove a fallacious belief by using measurements). Below is the listing of the so-called `InfinitelyScalableService` class (infinite scalability in software is equivalent to perpetuum mobile in physics):

Listing of the Fallacious InfinitelyScalableService Class

```
/**  
 * Demonstrates what happens when a wrong measurement  
 * reinforces a false belief.  
 *  
 * WARNING: DON'T USE THIS CODE!!! READ THE TEXT ABOVE!!!  
 */
```

```
public class InfinitelyScalableService {
    private final ExecutorService executor;

    /**
     * Creates a service configured to a desired parallelism level.
     *
     * @param parallelismLevel the level of parallelism to apply.
     * @throws InterruptedException if an interruption signal happens
     * during construction.
     * @see #stop()
     */
    public InfinitelyScalableService(int parallelismLevel)
        throws InterruptedException {
        if (parallelismLevel == 1) {
            executor = Executors.newSingleThreadExecutor();
        } else {
            executor = Executors.newFixedThreadPool(
                parallelismLevel);
        }
    }

    /**
     * Runs the tasks with a configured parallelism level.
     *
     * @param tasks the list of tasks to execute. Each task should
     * return one on completion.
     * @return the sum of tasks that have been completed.
     * @throws NullPointerException if the input argument is
     * {@code null}.
     * @throws InterruptedException if an interruption occurs during
     * execution of tasks.
     */
    public Long runTasks(List<Callable<Long>> tasks)
        throws InterruptedException {
        Objects.requireNonNull(tasks);
        List <Future<Long>> results = executor.invokeAll(tasks);
        return results.stream().mapToLong(f -> {
            try {
                return f.get();
            } catch (Exception e) {
                e.printStackTrace();
            }
            return 0;
        });
    }
}
```

```
    }).sum();
}

/**
 * Stops this service.
 */
public void stop() {
    executor.shutdown();
}
}
```

The idea of the above service is that if you have N tasks with some running time T , then to execute all of them in time T , you should create N threads. Is this a dream or reality? Well, the developer had provided the following unit test suite as a “proof”:

Listing of the Unit Test Suite for the Above Service

```
public class InfinitelyScalableServiceUnitTest {
    // Simulates a task whose execution requires 1 second.
    private static class BusyTask implements Callable<Long> {
        @Override
        public Long call() throws Exception {
            Thread.sleep(1000);
            return 1L;
        }
    }

    private final List<Callable<Long>> tasks = new ArrayList<>();
    private InfinitelyScalableService service;

    @After
    public void stopService() {
        service.stop();
    }

    public void createBusyTasks(int num) {
        for (int i = 0; i < num; i++) {
            tasks.add(new BusyTask());
        }
    }

    @Test
    public final void runThreeTasksInSingleThreadedMode()
```

```
        throws InterruptedException {
    service = new InfinitelyScalableService(1);
    createBusyTasks(3);
    assertEquals(Long.valueOf(3L), service.runTasks(tasks));
}

@Test
public final void runThreeTasksWithThreeThreads()
    throws InterruptedException {
    service = new InfinitelyScalableService(3);
    createBusyTasks(3);
    assertEquals(Long.valueOf(3L), service.runTasks(tasks));
}

@Test(timeout = 1500)
public final void runHundredTasksWithHundredThreads()
    throws InterruptedException {
    service = new InfinitelyScalableService(100);
    createBusyTasks(100);
    assertEquals(Long.valueOf(100L), service.runTasks(tasks));
}
}
```

The last test even has a constraint that it must finish in less than 1.5 s (that extra half second is kind of a safety net). The developer was absolutely assured that he had done everything properly.

What is wrong in the previous reasoning? First, 100 tasks (each taking time T to execute) can only finish their job in time T if you can truly run them in parallel. On a laptop with four cores, this is simply impossible. However, the tests are even weirder. The `BusyTask` class simulates a one second execution time by putting the thread to sleep for one second. This is a huge mistake. A sleeping thread doesn't consume the CPU. Therefore, it isn't a problem at all to keep all threads for one second in sleeping state and afterward waking them up. In this manner, all tasks can run in "parallel" even on a single-core CPU!

9.3 Visualizing the Gathered Data

Another danger in handling data lurks in summary statistics used to summarize a collection of observations. Occasionally, such summaries are even abused to distort the report about the situation on the terrain. A superb example about this topic is given by the Anscombe's quartet. This is comprised of four different datasets with essentially equivalent summary statistics. By looking at only their average value, you would think that there are no discernible variations between them. This is the reason why you always need to present data in complementary ways. Since Java

isn't quite good for fast visualizations of data, I have provided here a Python program to produce graphs of the previously mentioned quartet:

Listing of the Python Program to Visualize the Anscombe's Quartet

```
import pylab

# Set of pairs (X, Y), where the members are arrays.
quartets = [
    ([10.0, 8.0, 13.0, 9.0, 11.0, 14.0, 6.0, 4.0, 12.0, 7.0, 5.0],
     [8.04, 6.95, 7.58, 8.81, 8.33, 9.96, 7.24, 4.26, 10.84, 4.82, 5.68]),
    ([10.0, 8.0, 13.0, 9.0, 11.0, 14.0, 6.0, 4.0, 12.0, 7.0, 5.0],
     [9.14, 8.14, 8.74, 8.77, 9.26, 8.10, 6.13, 3.10, 9.13, 7.26, 4.74]),
    ([10.0, 8.0, 13.0, 9.0, 11.0, 14.0, 6.0, 4.0, 12.0, 7.0, 5.0],
     [7.46, 6.77, 12.74, 7.11, 7.81, 8.84, 6.08, 5.39, 8.15, 6.42, 5.73]),
    ([8.0, 8.0, 8.0, 8.0, 8.0, 8.0, 19.0, 8.0, 8.0, 8.0, 8.0],
     [6.58, 5.76, 7.71, 8.84, 8.47, 7.04, 5.25, 12.50, 5.56, 7.91, 6.89])
]

for quartet in range(1, 5):
    # Read out the current dataset
    xVals, yVals = quartets[quartet - 1]
    xVals = pylab.array(xVals)
    yVals = pylab.array(yVals)

    # Perform linear regression
    coef = pylab.polyfit(xVals, yVals, 1)
    linRegressionFun = pylab.poly1d(coef)

    # Produce the graph for the current dataset
    pylab.subplot(2, 2, quartet)
    pylab.plot(xVals, yVals, 'ro', xVals, linRegressionFun(xVals),
               '--k')
    pylab.xlabel("X" + str(quartet))
    pylab.ylabel("Y" + str(quartet))

    # Print summary statistics for the current dataset
    print "Quartet: " + str(quartet)
    print "Mean X: " + str(xVals.mean())
    print "Variance X: " + str(xVals.var())
```

```
print "Mean Y: " + str(round(yVals.mean(), 2))
print "Variance Y: " + str(round(yVals.var(), 2))
print "Pearson's correlation coef.:"
    + str(round(pylab.corrcoef(xVals, yVals)[0][1], 2))

pylab.show()
```

Here is the textual output showing the summary statistics (Fig. 9.1):

```
Quartet: 1
Mean X: 9.0
Variance X: 10.0
Mean Y: 7.5
Variance Y: 3.75
Pearson's correlation coef.:0.82
Quartet: 2
Mean X: 9.0
Variance X: 10.0
Mean Y: 7.5
Variance Y: 3.75
Pearson's correlation coef.:0.82
Quartet: 3
Mean X: 9.0
Variance X: 10.0
Mean Y: 7.5
Variance Y: 3.75
Pearson's correlation coef.:0.82
Quartet: 4
Mean X: 9.0
Variance X: 10.0
Mean Y: 7.5
Variance Y: 3.75
Pearson's correlation coef.:0.82
```

A visual representation is useful for us to see trends. For example, when you trace memory leak problems, the plot of the available memory shows a gradual decrease. However, the performance of the system may show a slight degradation until one point, where it would start to drop rapidly. Now with only summary statistics, you wouldn't be able to experience such variations and see exactly how the system behaves.

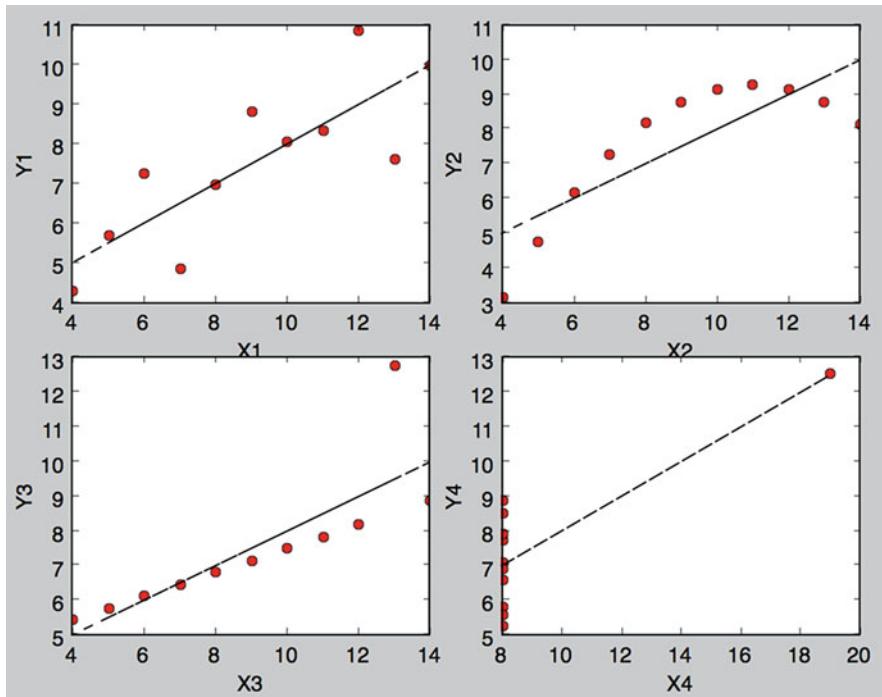


Fig. 9.1 The visualization of the four different datasets. You would never imagine that these have nearly identical summary statistics. This figure proves the legitimacy of the old proverb “*One picture is worth ten thousand words.*” In our case, we should replace “words” with summary statistics

9.4 Embracing the Holistic Notion of Monitoring

Monitoring imparts observability into the system. To attain crucial business goals, it must be coordinated in a holistic fashion. Monitoring must gather meaningful metrics and focus on important data. In some sense, through a set of metrics, it should establish a universal language across disparate domains: the user, business/management, and technical/development [5]. The development collects raw metrics related to the underlying hardware and software. CPU load level doesn't tell much to the business, even less to a user. Nevertheless, this could be a valuable input to derive/relate the revenue per minute business indicator of the Web site. The user is interested in some higher-level metrics as purchase completion time. To understand how raw performance counters may have meaning at the higher levels, all metrics must be interrelated and understood in different contexts. This is excruciatingly important for answering strategic questions like whether a high CPU load warrants further investments in hardware or software optimization. At any rate, the sole need to handle metrics in this manner opens pertinent

communication channels between domains and prevents the creation of specialized silos (user, business, and development). An interdisciplinary team gathered around common goals is in a better position to concoct an ingenious plan and select the important metrics to support that. For example, if the development team has noticed a slow performance in delivering complex multimedia Web site content, then the solution might be to deliver the content incrementally instead of all at once. While the user is reading the initial increment, the next can be prepared in the background. This approach might call for a totally different set of metrics. All in all, these sorts of solutions are unreachable when domains are isolated.

Another facet of monitoring is to increase resiliency [6], as more insight coupled with semiautomated operational procedures may enable optimal decisions. Again, what to monitor is directly governed by the goals the operations team would like to achieve. Remember that piling up useless metrics (just because they are easy to collect) would produce clutter and increase the chance of an operator to miss important details of the system's behavior. Moreover, it is hard to properly interrelate dozens of metrics to configure alarming conditions. At least, you should group collected data to create suitable views into the system.

9.5 Measurements: Problem

In a large dynamic environment, the centralized Swaggerize service is called in a rapid fashion by other services to register/deregister themselves. The QA department would like to set up a cluster of Swaggerize services with an autoscaling facility. They would like to monitor performance counters from the service and create or destroy instances as needed. The ACME Corporation would like us to incorporate adequate measurements into the Swaggerize service.

9.6 Measurements: Resolution

To address the above demand, we should record the following metrics:

- The duration of HTTP requests
- The rate of requests for each endpoint
- The number of errors that occur on each endpoint

Measuring how often exceptions appear while processing requests is valuable to see whether the service has some issues under high load. For example, inherent concurrency problems are more probable to occur when the service is bombarded with requests.

Dropwizard already has support for recording these metrics. We just need to augment the resource methods with built-in annotations. Here is the patch to the `ServicesApi` resource class:

Listing of the Partial Patch to the ServicesApi Resource Class

```
@PUT  
@Path("/{serviceName}")  
@Timed(name = "putService.time")  
@Metered(name = "putService.frequency")  
@ExceptionMetered  
@UnitOfWork  
@PermitAll  
public Response putService(
```

Another method annotated with these new stuff is the `deleteService`. These two methods are called by other services, so we need to track them. The `ExceptionMetered` annotation automatically embellishes the metric name with a proper suffix. The first two must be explicitly named. Otherwise, a duplicate metric name would trigger a run-time error. All in all, these are the changes related to the source code.

The next thing is to configure the service to collect these metrics (visit www.dropwizard.io/1.1.4/docs/manual/configuration.html#metrics). The metrics configuration has two sections: frequency and reporters. The first defines the regularity to report metrics, while the latter the destinations for the collected data. Dropwizard allows you to send the same data to multiple places. It is also possible to report metrics as logs and thus send them into the same location, where other logs are situated. For now, we will only use the basic console reporter. In the next chapter, we will setup a small infrastructure automatically and send metrics to Graphite. Here is the extension to our configuration file:

```
metrics:  
  reporters:  
    - type: console  
      locale: en_US
```

The other values are kept on default. It is very important to standardize based on the locale and time zone (by default it is set to UTC). Never use a system standard locale, as different nodes may be set up differently (or not set up at all and use the default OS setting). If we start our service and register a new endpoint, then after a minute (the default report sending rate is 1 min), we should see something like as follows (there are lots of other metrics that are reported, and you can select in the configuration what to include/exclude):

```
com.example.swaggerize.resources.ServicesApi.putService.time  
  count = 1  
  mean rate = 0.01 calls/second  
  1-minute rate = 0.01 calls/second
```

```
5-minute rate = 0.00 calls/second
15-minute rate = 0.00 calls/second
    min = 222.94 milliseconds
    max = 222.94 milliseconds
    mean = 222.94 milliseconds
    stddev = 0.00 milliseconds
    median = 222.94 milliseconds
    75% <= 222.94 milliseconds
    95% <= 222.94 milliseconds
    98% <= 222.94 milliseconds
    99% <= 222.94 milliseconds
    99.9% <= 222.94 milliseconds
...
com.example.swaggerize.resources.ServicesApi.putService.frequency
    count = 1
    mean rate = 0.01 events/second
    1-minute rate = 0.00 events/second
    5-minute rate = 0.00 events/second
    15-minute rate = 0.00 events/second
...
com.example.swaggerize.resources.ServicesApi.putService.exceptions
    count = 0
    mean rate = 0.00 events/second
    1-minute rate = 0.00 events/second
    5-minute rate = 0.00 events/second
    15-minute rate = 0.00 events/second
```

This is the result after calling once the endpoint to register a service. Every metric has multiple attributes. The `count` is the raw counter, while the others are summary statistics. For example, the 95% percentile is named as `p95`. Attributes, similarly to metrics, may be selectively included/excluded in the configuration file.

9.7 Centralized Logging: Problem

The QA department had noticed that some services are not registered with Swaggerize after startup. In the absence of logs, they are not able to figure out what went wrong. Therefore, they demand from us an upgrade of the service to produce logs that can be analyzed from a central location.

9.8 Centralized Logging: Resolution

Our first duty is to establish a mutual understanding about *logs*. Dumping messages onto standard output or error streams or simply letting `e.printStackTrace()` to do its job cannot be classified as logs. Below are some of the reasons why they aren't logs:

- The messages are unstructured and require a human being to interpret them. Sometimes, they are so obscure that only the developer who created them may unscramble their content.
- The messages are lost if they aren't explicitly redirected into a file using OS-level constructs.
- There is no way to prioritize messages based upon a severity level. All `System.out.println` or `System.err.println` calls result in messages of seemingly equal importance.
- There is no way to filter or search messages based upon some attributes. All messages are lumped together without an ability to categorize them by their source.
- `System.out.println` and `System.err.println` calls are temporary in the code base, and many are removed, depending on the execution mode (production or development). There is no way to selectively turn them off inside a configuration file. Every change in the amount of logging necessitates a recompilation of the code base.

For us, the log is a *well-structured, human-readable, machine-processable* message, which may be *persisted, searched, visualized, and correlated* with other such messages. This definition has three sections: message structure, message handling, and message correspondence. We will use JavaScript Object Notation (JSON) as a format for our log messages. This is both human readable and machine parsable. The processing of messages will be done by a proper logging tool. Figure 9.2 depicts the constituent parts of the ELK stack and how it satisfies the previous definition. Finally, the message correlation will be implemented by using a special correlation identifier for all log messages. Figure 9.3 shows the importance of interrelating log messages in a distributed system.

Our definition blurs the difference between pure application logs and business events (logs) [7]. From our log message perspective, there are no distinctions. Both contain valuable enterprise information worthwhile to be analyzed.

Dropwizard has superb built-in support for logging messages and uses the Logback framework (visit logback.qos.ch) for its logging back end. The nice thing is that Dropwizard will route logs from various logging APIs including `java.util.logging`. Therefore, we don't need to change our source code to accommodate Logback. We will use the standard Java's logging facility (for more details, see www.dropwizard.io/1.1.4/docs/manual/core.html#logging).

The production and development log configuration should be different. During development, we just need the simplest console logging that records everything from all loggers. In production, we would like to be more selective. Here is the

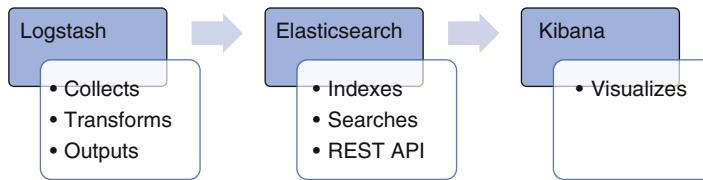


Fig. 9.2 ELK allows logs to be stored in a central location, efficiently retrieved either by using the Elasticsearch's level 2 REST API and visualized by Kibana within a Web browser. The Elasticsearch may run as a cluster for performance reasons since it is the core engine that indexes and searches logs

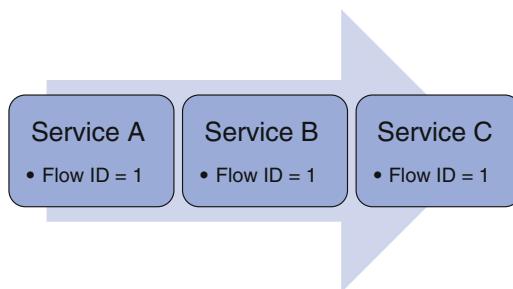


Fig. 9.3 The workflow instance that impacts multiple services may be associated with a correlation (flow) identifier. This enables grouping of log messages belonging to the matching flow

production configuration setting for our Swaggerize service (we will add the Syslog appender in the next chapter):

```

logging:
  # The default level to all loggers.
  level: INFO

  loggers:
    # Permit informational messages for our service.
    com.example.swaggerize: INFO

  appenders:
    # Log warnings and errors to stderr
    - type: console
      threshold: WARN
      target: stderr

    # Log informational messages to stdout
    - type: console
  
```

```
threshold: INFO
target: stdout

# Log info, warnings and errors to our service's main log.
# Rolled over daily and retained for 10 days.
- type: file
  threshold: INFO
  logFormat:
    "{\"service\": \"%X{serviceName}\", \"createdAt\": \"%d{yyyy-MM-
dd'T'HH:mm:ss.SSS}\", \"level\": \"%p\", \"class\": \"%c\", \"thread
\": \"%t\", \"correlationId\": \"%X{correlationId}\", \"message\": \"%
m\"}\n"
  currentLogFilename: ./logs/swaggerize.log
  archivedLogFilenamePattern: ./logs/swaggerize-%d.log.gz
  archivedFileCount: 10
  bufferSize: 256KB

# Log debug messages, info, warnings and errors to our service's
# debug log. Rolled over hourly and retained for 24 hours
- type: file
  threshold: TRACE
  queueSize: 512
  logFormat:
    "{\"service\": \"%X{serviceName}\", \"createdAt\": \"%d{yyyy-MM-
dd'T'HH:mm:ss.SSS}\", \"level\": \"%p\", \"class\": \"%c\", \"thread
\": \"%t\", \"correlationId\": \"%X{correlationId}\", \"message\": \"%
m\"}\n"
  currentLogFilename: ./logs/swaggerize-debug.log
  archivedLogFilenamePattern: ./logs/swaggerize-debug-%d{yyyy-
MM-dd-HH}.log.gz
  archivedFileCount: 24
  neverBlock: true
  bufferSize: 256KB
```

We are overriding the global WARN level for our service to be INFO. There are three explicit appenders: a console for redirecting warnings and errors into the `stderr` device, a rolling file appender for regular messages, and a rolling file appender for debug messages. File appenders store JSON-formatted messages. The `serviceName` and `correlationId` are custom properties that will be set inside the application. It is important to increase the default buffer size for file appenders for performance reasons. Also, we don't want to block on debug messages, so the `neverBlock` is set to `true`. As the volume of debug messages is large, the queue size is set to 512 (the default is 256). We are left with two tasks: to set the correlation identifier to a Universally Unique IDentifier (UUID) if it isn't provided and to set the previously mentioned custom properties from the application (see Exercise 3).

Logging into file or sending logs over the network is a security concern, especially if it contains business intelligence data or system-related messages that may uncover some weaknesses. Encryption doesn't matter if you only care about data at rest inside your database. Your logs must be protected too. An astute attacker will always choose the path of least resistance to grab sensitive information.

The handy way to process the custom HTTP request header X-LogCorrelationId is to install a request filter. It will add that header if it is missing. Here is the listing of our custom filter:

Listing of the LogCorrelationIdFilter Class

```
/**  
 * This filter checks for the optional log correlation id header,  
 * and adds it as necessary.  
 */  
@Provider  
public final class LogCorrelationIdFilter implements  
    ContainerRequestFilter {  
    private static final Logger LOG =  
        Logger.getLogger(LogCorrelationIdFilter.class.getName());  
    private final String headerName;  
  
    public LogCorrelationIdFilter(String headerName) {  
        this.headerName = headerName;  
    }  
  
    @Override  
    public void filter(ContainerRequestContext requestContext)  
        throws IOException {  
        String logCorrelationId =  
            requestContext.getHeaderString(headerName);  
        if (logCorrelationId == null) {  
            logCorrelationId = UUID.randomUUID().toString();  
            LOG.log(  
                Level.FINE,  
                "Adding correlation id: " + logCorrelationId);  
            requestContext  
                .getHeaders()  
                .add(headerName, logCorrelationId);  
        }  
    }  
}
```

This filter must be registered at startup, so the following line is needed inside the main method of our application class (we also need to add the corresponding check in our system test):

```
environment.jersey().register(new LogCorrelationIdFilter(  
    SwaggerizeConfiguration.LOG_CORRELATION_ID_HNAME));
```

Finally, here is the listing of the partial patch to our resource class that accepts the log correlation identifier and sets the custom logging properties:

Listing of the Partial Patch to the ServicesApi Class

```
public Response deleteService(  
    @HeaderParam(  
        SwaggerizeConfiguration.LOG_CORRELATION_ID_HNAME)  
    @NotEmpty  
    String logCorrelationId,  
    @PathParam("serviceName")  
    @NotEmpty  
    String serviceName) {  
  
    MDC.put(  
        SwaggerizeConfiguration.SERVICE_NAME_MDC,  
        SwaggerizeConfiguration.SERVICE_NAME);  
  
    MDC.put(  
        SwaggerizeConfiguration.LOG_CORRELATION_ID_MDC,  
        logCorrelationId);  
  
    LOG.entering(CLASS_NAME, "deleteService",  
        new Object[] { serviceName });  
  
    Response response = null;  
    try {  
        serviceEntryDAO.deleteByName(serviceName);  
        response = Response.noContent().build();  
    } catch (Exception e) {  
        throw new ApiException(  
            ApiException.ErrorDescriptor.INTERNAL_ERROR, e);  
    } finally {  
        MDC.remove(SwaggerizeConfiguration.LOG_CORRELATION_ID_MDC);  
        MDC.remove(SwaggerizeConfiguration.SERVICE_NAME_MDC);  
        LOG.exiting(CLASS_NAME, "deleteService", response);  
    }  
    return response;  
}
```

The other methods are amended in the same way (see Exercise 4). We must also update our integration tests (see the book’s source code) to set the log correlation identifier header (our filter will not run in those tests). The MDC stores the properties associated with the current thread.

During execution, the logging levels may be dynamically altered (this is a built-in feature of Dropwizard). For example, to set the logging level for our service to TRACE, you should execute the following command:

```
curl -X POST -d "logger=com.example.swaggerize&level=TRACE" \
http://localhost:8081/tasks/log-level
```

We have completed our job regarding logging, and with the previous metrics extension, we can release the new 2.1.0 version of the Swaggerize service (see Exercises 5 and 6).

9.9 Summary

Monitoring and logging equips us with ability to see what is happening with our infrastructure and applications. We cannot run a production environment blindly. It isn’t solely about protecting our system against failures or autoscaling the system based upon demand. As we have witnessed, data is precious also from the business intelligence viewpoint. Metrics and logs contain all sorts of useful data: how our system is used, when is the peak activity of users, what are the most popular sections of the API, etc. These may drive further evolution and expand the business. With such data, we can base evolution on facts, rather than pure guesses. You can always be one step ahead of your users, without waiting for them to ask for new features. You would know what they want in advance. The key takeaways from this chapter are listed below:

- Monitoring and logging is required to both keep our system running as well as proactively govern our system’s evolution.
- The collected data from the running system has a long-term value. It also enables postmortem debugging sessions when you must analyze what had happened only by looking at the acquired data [8].
- Metrics and logs must support the principal use cases of the system and entail a holistic treatment.
- The architecture must enable efficient monitoring and logging, as these are architecturally significant nonfunctional requirements.

9.10 Exercises

1. (Self-study) Read about the *Rhine paradox* that is also a very common trap when working with data.

2. (Self-study) A typical computer system embodies various clocks. The situation is even more exaggerated by multicore CPUs and/or distributed computer systems. The latter must even take care to synchronize the clocks over multiple machines, although this feat cannot be perfectly achieved. Serious errors may originate from lack of knowledge about the intricacies of existing clocks. For example, judging about ordering of events in an asynchronous distributed system cannot rely upon bare timing information. Likewise, more precise per-core timers on a computer may drift from each other and become unreliable for measuring long-running tasks. Read [3–4] to get an insight about the difficulties in creating trustworthy time series data.
3. Extend the service descriptor with a global optional HTTP request parameter `X-LogCorrelationId`. The type should be `string` with a comment that it is an UUID (you may want to supply the regular expression constraint to only permit properly formatted UUIDs).
4. Refactor the code of the `ServicesApi` class to avoid duplicating the logic to set and remove properties from the MDC storage. Moreover, add the same support into the `IndexPage` resource class.
5. Expand the level of logging emitted from the `ServicesApi` resource class for the methods that are also used by other services. These would be informational messages recording the registration/deregistration events.
6. (Project assignment) You have seen so far two ways to alter the logging level: setting it in the configuration file and summoning a POST HTTP request. It may seem that the latter is all that we need since we can change the logging level remotely. However, it cannot help to catch lower-level logs when an issue appears. It would be best if we could somehow log a couple of earlier debug messages before logging a warning or error. This is possible with the usage of the `JDK MemoryHandler` class that persists logs inside a circular buffer in memory. It allows us to set a threshold that triggers the push of stored messages. So, it can accumulate the last N debug messages and when an error comes in, emit those stored messages with the error message. In this fashion, we can gain more information on what preceded the error condition. Implement this logging approach, and incorporate it into our Swaggerize service. You may also want to offer a new Dropwizard task called `log-threshold-level` to set this triggering value remotely.

References

Further Reading

1. Turnbull J (2016) The art of monitoring. www.artofmonitoring.com. Accessed 20 Sep 2017. This book gives you an overview about monitoring and logging, and introduces you to Riemann, Graphite, and the ELK stack. It covers both topics: collecting data on clients as well as processing them on the central location

Regular Bibliographic References

2. Leskovec J, Rajaraman A, Ullman DJ (2014) Mining of massive datasets, 2nd edn. Cambridge University Press, Cambridge
3. Neil NG (2016) Time is an illusion: lunchtime doubly so. queue.acm.org/detail.cfm?id=2878574. Accessed 22 Sep 2017
4. Ratzel R, Greenstreet R (2012) Toward higher precision. Commun ACM 55(10):38–47
5. Brewer B, Zeman M, Souders S (2015) Creating meaningful metrics that get your users to do the things you want. O'Reilly, Sebastopol, CA
6. Tseitlin A (2013) The Antifragile organization. Commun ACM 56(8):40–44
7. Kreps J (2014) I ❤ logs. O'Reilly Media, Sebastopol, CA
8. Pacheco D (2011) Postmortem debugging in dynamic environments. Commun ACM 54(12): 44–51

The previous chapters of this book have talked about the prerequisites to attain DevOps. DevOps is an abyss-crossing movement, like test-driven development (TDD) regarding testability, where deployability and observability are treated as architecturally significant quality attributes. This entails connecting construction with deployment and operational aspects of the produced system. As TDD demands sophisticated tools and frameworks, the same applies for DevOps. Nonetheless, the tools are only helpful if the whole system is built around the DevOps paradigm. This chapter introduces some key technologies and tools to scale the deployment of a system that is developed using the DevOps architectural style.

DevOps requires a holistic software engineering standpoint, where product and project management, development, and operational actions are fully coordinated and driven by an architecture of the system [1–2]. It also mandates a carefully chosen set of tools [3] since without them you cannot have DevOps.

As a positive side effect of DevOps and automation is the focus on quality. In other words, you cannot apply continuous deployment/delivery without continuous quality (see Exercise 1). The adoption of DevOps practices also works in backward direction and has a positive influence on quality assurance [4]. From this standpoint, DevOps may be regarded as a proactive risk management method. Many omissions happen due to human errors caused by conditions that are risky. Instead of trying to put more burden on humans, we can minimize the risk of errors by altering the conditions under which people work. In this manner, we reduce the chance of having problems during deployment. Here are some those pesky aspects of manual labor pertaining to deployment:

- Repetitive mundane work lessens the attention level of a human operator. This opens the possibility of inducing a careless error by a person that wouldn't occur otherwise.
- Massive documents about the deployment process may contain errors and are hard to fully comprehend. Testing the accuracy of a Word text cannot be automated nor left to a machine. Also, it is hard to keep the documents in sync

with the latest changes in the software. Many details could be stale, leading even to serious security and stability issues.

- Under heavy pressure, the risk of erring increases considerably. Therefore, passive document-based processes are erratic in the most critical moments.
- The manual deployment process isn't scalable in volume (measured in the number of deployments over some period) and is usually targeted to a fixed environment. Replicating the production system on your laptop is nearly impossible to accomplish by following the documents. Furthermore, the document-based approach doesn't scale across the organization either. You will usually need specialized personnel to do the deployment.

TDD and DevOps interconnect domains and prevent silos in the organization. DevOps establishes a bidirectional communication between development and operations teams. I had seen a perfect example of the opposite direction of DevOps (operations → development) while visiting the U-Boat Museum in Hamburg, Germany. The Russian submarine captain had participated in all phases of the construction work and must have known everything about the functioning of the submarine. In the modern DevOps world, operations personnel provide valuable input to development, without necessarily watching the whole development phase.

DevOps is tightly associated with automation of operational aspects of the system (including deployment), and this must be done in a prudent manner. We want to diminish the probabilities of having risks turn into problems in production, and this can be achieved in three harmonizing ways:

- Automating aspects of deployment and operations regarded as “monkey business.” You have already seen this in action in Chap. 6. The automatic database schema/data migration is a fine example for this item. Moreover, by letting the service take care of database schema tasks, you don't need to worry about administering the target database or contacting the right person. If database admins would need to perform all those tasks manually, they would quickly become a bottleneck.
- Leaving enough wiggle room for leveraging human intelligence in critical moments. This is crucial to gain a decent level of system resiliency. Most complex systems do possess some emergent properties that can only be recognized by people. Out-of-the-box problem solving capabilities cannot be automated, unless artificial intelligence techniques become competitive with humans. One-click software delivery is an attractive goal, but it shouldn't enforce a zero-click human intervention during the production run.
- Eliminating backseat drivers from the process. More automation means that decisions made by professionals are protected against lay people, who may override an otherwise sound action. This philosophy also applies to configuration settings, where you must minimize the number of publicly accessible parameters and keep the rest protected for privileged users.

10.1 Multi-phased Continuous Deployment/Delivery

Continuous deployment has a similar goal as delivery, with a distinction that the former doesn't automatically push the software to the customer. At any rate, they both try to provide the following benefits:

- Increased productivity by focusing only on new features instead of technicalities regarding how to deliver those
- Improved stability by letting a machine detect any deployment issues (instead of manually performing checks)
- Executable “documentation” by letting the deployment scripts speak for themselves
- Enhanced testability by allowing each delivery stage (phase) to be independently verified through automated tests
- Elevated reuse potential, as each phase may be leveraged by different teams in various environments

Figure 10.1 shows the idea behind the multiphased automation. Each phase is reminiscent of a software layer, as both introduce delineations between different abstraction levels and the lower phase (layer) is independent from upper phases (layers). One phase is an input to the other. Phasing helps quality control as well as boosts agility. A single node may not need to run all deployment steps every time, if the process is staged. Decoupling the phases supports various use cases: deploying from scratch, updating only one component of the infrastructure, installing a new application, etc.

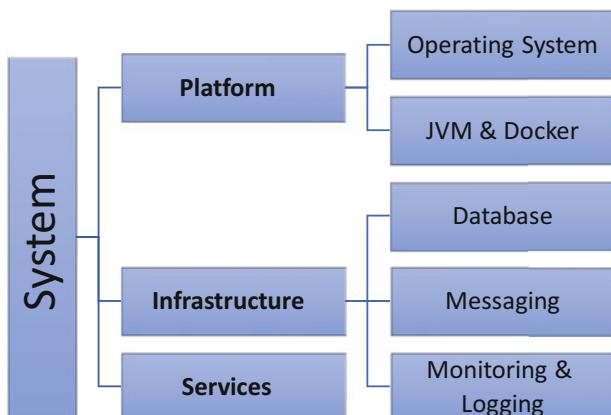
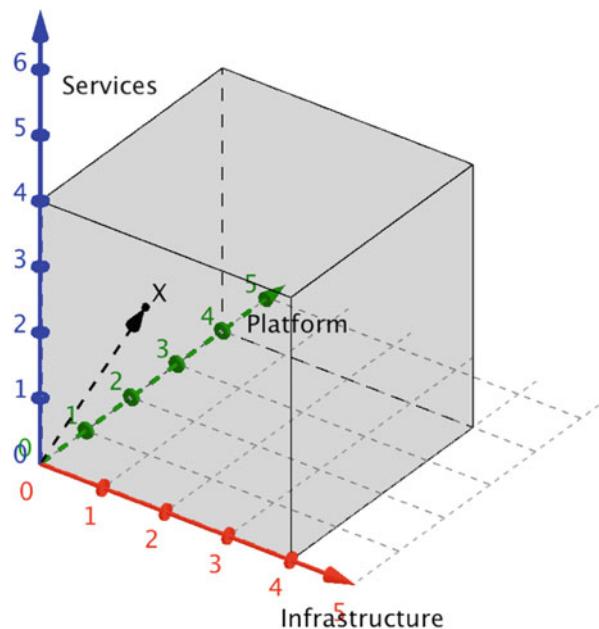


Fig. 10.1 The multiphased deployment/delivery automation approach shown upside down. As we move from top to bottom, the phases transition from more stable and versatile to more volatile and specific variants. Correspondingly, the speed of deployment/delivery increases as we move toward the higher levels. It should be much faster to upgrade a single service instead of the underlying infrastructure

Fig. 10.2 The automation cube shows your current position and highlights the nonautomated area. The point X marks the current level of automation across dimensions



We don't need to automate all phases (directions) in the same amount. For example, maybe you want to leave the infrastructure part less automated than the services layer. This brings us to the idea of an automation cube (or polyhedra in case of more phases). Figure 10.2 shows the cube for our three phases: platform, infrastructure, and services. Each directional axis is divided into units of automation (this should be established inside a company). Afterward, a team should decide how much to automate in each direction, which sets the target goal of the continuous deployment/delivery process. The cube (or especially the polyhedra) is useful to visualize what areas are weakly automated. Each direction should be supported by tools that must be assembled to comprise an automation ecosystem.

It is important to realize that you may implement phases using various technologies. The platform can be crafted as a base image of a virtual machine (VM) or a base image of a Docker container. Usually, you want to handle platform and infrastructure inside VMs and let services be packaged as containers. Sure, in this case the platform must include the corresponding container technology toolset (e.g., Docker Engine, Docker Compose, etc.). The Java Virtual Machine (JVM) is handy to be part of the platform when some infrastructure components are better run without any additional virtualization wrapper (e.g., Zookeeper should be run directly on the node on top of the JVM).

10.2 Infrastructure as a Code: Problem

The QA team is complaining that it takes too much time to setup a new node from scratch. They would like to have a good starting point and immediately jump to deploying application services.

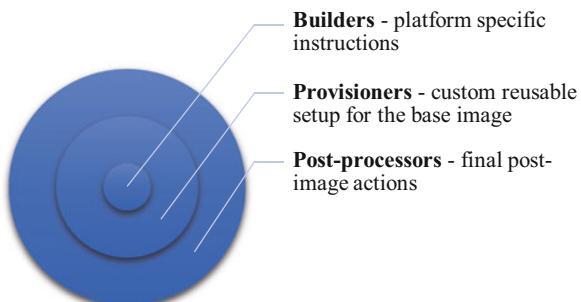
10.3 Infrastructure as a Code: Resolution

We will use Packer (visit packer.io) to automate the platform/infrastructure base image creation. In lighter automation scenarios, you may bundle platform and infrastructure components inside the same base image. At any rate, this will allow the QA team to have a ready VM with all necessary base products installed. The first thing you should do is to install Packer onto your machine. On Mac OS X you may use the Homebrew package manager. At the time of this writing, the latest version was 1.1.0.

Figure 10.3 shows the three major structural components of Packer. The purpose of such arrangement is to augment reuse. For example, if you want to target another platform, like Amazon, then you just need to introduce a new builder component. The provisioners can remain the same. The main benefits of using Packer are as follows:

- Fast platform/infrastructure deployment, since new nodes may be launched instantaneously without waiting heavy components to install.
- Multiprovider portability by creating identical images for multiple platforms. This prevents mismatches between environments, which is especially important to faithfully reproduce a production system on your local machine.
- Improved stability, since Packer prepares images based upon your scripts (executable deployment “documentation”). Any error is caught early and automatically.
- Greater testability by smoke testing the image after it is built.

Fig. 10.3 Packer’s layered organization. The innermost layer is the entry point into the image creation process. Each layer is a unit of reuse. Additional image production customizations may be carried out by Packer plugins



We will create a new project called `vm-base-image` that will contain all the deployment scripts. Keeping these artifacts inside a version control system is the basic step toward realizing the concept called *infrastructure as code*.

The fastest way to get started is to clone the github.com/chef/bento project onto your local machine by issuing:

```
git clone https://github.com/chef/bento vm-base-image
```

Bento is a community project with Packer templates for building minimal Vagrant (visit www.vagrantup.com) base boxes. We will use Vagrant version 2.0.0 to control the VMs. We are interested in to keep the `ubuntu/ubuntu-17.04-amd64.json` Packer template and associated shell scripts. All the rest can be deleted. We will rename the previous template by adding the `-acme` suffix. In this chapter, we are going to focus only on VirtualBox (visit virtualbox.org) virtualization provider. The following additions will be incorporated into the base image using the shell provisioner (each component is installed via the matching shell script):

- Java SE 8 Runtime Environment
- Docker CE and Docker Compose
- MySQL (as a Docker image)
- Graphite (as a Docker image)

The shell scripts are situated inside the `ubuntu/scripts` folder. These are added to the list of executables inside the `provisioners/scripts` section (consider the book's source code). Here is the patch to the original Bento Packer template file (there is an additional modification not shown here to increase the compression level for the `vagrant` post-processor):

Listing of the Partial Patch to the Original Bento Packer Template File

```
"builders": [
  {
    ...
    "export_opts": [
      [
        "--manifest",
        "--vsys", "0",
        "--description", "{{user 'description'}}",
        "--version", "{{user 'version'}}"
      ]
    }
  }
]
"scripts": [
  ...
]
```

```
"scripts/docker.sh",
"scripts/mysql.sh",
"scripts/graphite.sh",
"scripts/cleanup.sh",
"scripts/jre8.sh",
...
],
"variables": {
  "box_basename": "ubuntu-17.04-amd64-acme",
  "build_timestamp": "{{isotime \"20170926200000\"}}",
  "cpus": "2",
...
  "headless": "true",
...
  "memory": "4096",
...
  "template": "ubuntu-17.04-amd64-acme",
  "version": "1.0.0",
  "description": "Base infrastructure node image."
}
```

It is useful to validate the template file after each change. This can be done by running `packer validate ubuntu-17.04-amd64-acme.json`. Before running the next command from the `ubuntu` subfolder (see Exercise 3), you must have VirtualBox installed on your machine:

```
packer build -only=virtualbox-iso ubuntu-17.04-amd64-acme.json
```

Depending on the speed of your network connection, this may take 30 min to finish. Once this process completes, you will have a Vagrant base box ready inside the `builds` folder (see Exercise 4). At this moment, you should create a tag in the repository and name it as `<image name>-1.0.0` (the suffix is the actual version number of the image). The next step is to bootstrap Vagrant by registering this box with the following command (issue it from the same `ubuntu` subfolder):

```
vagrant box add --name acme/ubuntu-17.04 ../builds/ubuntu-17.04-
amd64-acme.virtualbox.box
```

We will now create a new project `infrastructure-setup` for setting up the infrastructure using our base image. Vagrant abstracts away the peculiarities in handling virtual machines for different platforms. In other words, you may use the same facility to control VMs in VMware, VirtualBox, AWS, or any other supported platform. Here is the Vagrant script to create two VMs using VirtualBox and start MySQL, Graphite, and the rsyslog server on those nodes. Here is the listing of the matching Vagrant script named as `Vagrantfile-acme-infrastructure`:

Listing of the Vagrant Script to Construct and Set Up the Infrastructure Nodes Locally

```
# -*- mode: ruby -*-
# vi: set ft=ruby :

Vagrant.require_version '>= 2.0.0'
ENV["LC_ALL"] = "en_US.UTF-8"

Vagrant.configure(2) do |config|
  config.vm.box = 'acme/ubuntu-17.04'

  config.vm.provision 'bootstrap', type: 'shell' do |sh|
    sh.path = 'scripts/bootstrap-node.sh'
  end

  # Test the assumptions regarding the base image.
  config.vm.provision :serverspec do |spec|
    spec.pattern = 'tests/*_spec.rb'
  end

  config.vm.network "private_network", type: "dhcp"

  # This is a group of 2 VMs for local (currently only VirtualBox)
  # execution.
  (1..2).each do |i|
    config.vm.define "infrastructure-vm-#{i}" do |node|
      node.vm.hostname = "infrastructure-vm-#{i}"
      node.vm.synced_folder '.', '/vagrant', disabled: true

      node.vm.provider 'virtualbox' do |vb|
        vb.name = "ACME Infrastructure VM #{i}"
        vb.memory = 2048

        # Uncomment the next line to switch VirtualBox into GUI mode.
        # This is useful for debugging purposes.
        # vb.gui = true
      end

      if i == 1 then
        node.vm.provision 'database', type: 'shell' do |sh|
          sh.name = 'Setting up the relational database...',
          sh.path = 'scripts/setup-database.sh',
        end
      end
    end
  end
end
```

```
    sh.privileged = false,
    sh.args = "#{ENV['DB_USERNAME']} #{ENV['DB_PASSWORD']}
              #{ENV['DB_SCHEMA']}"
  end
  elsif i == 2 then
    node.vm.provision 'monitoring/logging', type: 'shell' do |sh|
      sh.name = 'Setting up the monitoring and logging...',
      sh.path = 'scripts/setup-monitoring-logging.sh',
      sh.privileged = true
    end

    node.vm.post_up_message =
      %q(Execute 'vagrant ssh <VM name>' and then 'ifconfig' to
          find the IP address of the VM.)
  end
end
end
end
end
```

We are reducing the memory footprint of a single VM to 2GB and are running appropriate shell scripts to provision the VM (see the book's source code). The above script also uses the Serverspec framework (visit serverspec.org) to smoke test the base image. The required Vagrant plugin can be installed by executing `vagrant plugin install vagrant-serverspec` (see Exercise 5). Here is the test specification (see the book's source code for the helper script), which is very readable even by nondevelopers:

Listing of the Test Specification to Smoke Test the Base Image

```
require_relative 'spec_helper'

# Check that Docker is properly setup.
describe service('docker') do
  it { should be_enabled }
  it { should be_running }
end
describe group('docker') do
  it { should exist }
end

# Check that we have pulled all required Docker images.
describe docker_image('mysql:5.7.19') do
  it { should exist }
```

```

end
describe docker_image('graphiteapp/graphite-statsd') do
  it { should exist }
end

# Check the other installed packages and daemons.
describe command('docker-compose -version') do
  its(:stdout) {
    should contain('1.16.1').after('docker-compose version')
  }
end
describe command('java -version') do
  its(:stderr) { should contain('1.8').after('openjdk version \''')}
}
end
describe command('curl --version') do
  its(:stdout) { should contain('7.52.1').after('curl') }
end

```

We can start our VMs by issuing the following commands (the `export` statements are only required once per command line window session):

```

export VAGRANT_VAGRANTFILE=Vagrantfile-acme-infrastructure
export DB_USERNAME=admin
export DB_PASSWORD='<your password for admin>'
export DB_SCHEMA=Swaggerize_DB
vagrant up

```

At this moment, we have 2 VMs up and running. As with the base image, you may want to tag the matching commit in your version control system. We are left with expanding the configuration file of our Swaggerize service to send metrics to Graphite and logs to the Syslog server (see Exercise 6). We will set the new version number of our service to 2.2.0. Here is the patch to the configuration file:

Listing of the Patch to the Swaggerize Service's Configuration File

```

logging:
  appenders:
  ...
    # Send logs to a syslog server.
    - type: syslog
      host: ${SYSLOG_HOST}
      port: 514
      threshold: ALL
      logFormat:

```

```
"{\\"service\": \"%X{serviceName}\", \"createdAt\": \"%d{yyyy-MM-  
dd'T'HH:mm:ss.SSS}\", \"level\": \"%p\", \"class\": \"%c\", \"thread  
\": \"%t\", \"correlationId\": \"%X{correlationId}\", \"message\":  
\": \"%m\"}%n"  
  
metrics:  
  reporters:  
    ...  
    - type: graphite  
      host: ${GRAPHITE_HOST}  
      port: 2003  
      prefix: swaggerize
```

We also need to add the following dependency to the build file:

```
<dependency>  
  <groupId>io.dropwizard</groupId>  
  <artifactId>dropwizard-metrics-graphite</artifactId>  
</dependency>
```

After we have started up the infrastructure and recorded the IP addresses of the machines, we can start our Swaggerize service. Don't forget to set all environment variables properly and perform an initial database migration. All in all, we have finalized the first assignment in this chapter.

10.4 Deployment: Problem

The QA team would like a more compact and easier way to deploy and use our Swaggerize service. They would also like to use it inside available container environments.

10.5 Deployment: Resolution

Let us package our service as a Docker image. For this, we will need to do two things: define the Dockerfile and boost our build process to create a Docker image automatically. The Dockerfile contains instructions on what should the image contain and how to start it up. It is very important to understand the layered structure of the container's filesystem. During the image build, Docker executes the statements from the Dockerfile. Each composite action comprises one logical unit of change in the filesystem. When you alter the Dockerfile or inherit a base image, Docker will only run the commands that differ from the most recent reusable state (layer). This also dictates how big is the delta between various versions of an image (this can have a huge impact on the download time). Therefore, you should follow the next two simple rules (you will see all this in action soon):

- Starts from the most stable commands in your Dockerfile and moves toward the most volatile
- Group statements inside a composite that you judge makes no sense to be split

Here is the Dockerfile for creating the image for our service:

Listing of the Swaggerize Service's Dockerfile

```
FROM phusion/baseimage:0.9.22
LABEL maintainer="Ervin Varga ervin.varga@exploit.rs"

CMD ["/sbin/my_init"]

# Install JRE 8+
RUN apt-get -y update \
    && apt-get install -y --no-install-recommends default-jre

RUN mkdir /etc/service/swaggerize
COPY docker/swaggerize.sh /etc/service/swaggerize/run
RUN chmod +x /etc/service/swaggerize/run

WORKDIR /usr/share/swaggerize
COPY target/swaggerize.jar swaggerize.jar
COPY config.yml default-conf/config.yml

RUN apt-get clean && rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/*
EXPOSE 8080
EXPOSE 8081
```

We put the installation of the JRE 8+ before our application stuff, as it is more stable. Also, the whole command is comprised from two subcommands and is kept as a single unit. We use the phusion/baseimage Docker image as a starting point (visit github.com/phusion/baseimage-docker). During the build, we include the `swaggerize.sh` shell script as part of final image, whose execution is monitored by the `runit` tool. This script is shown below:

Listing of the `swaggerize.sh` Script

```
#!/bin/bash

set -e
cd /usr/share/swaggerize
```

```
# Copy over the default configuration into the target
# configuration folder. Don't copy if the user has overridden
# the default settings.
mkdir -p conf
cp -u --backup=simple default-conf/*.* conf/

# Migrate the database, as needed.
java -jar swaggerize.jar db migrate conf/config.yml

# Start the service.
java -server -jar swaggerize.jar server conf/config.yml
```

The Swaggerize service is now fully encapsulated and only exposes the minimal set of configuration options (mostly the environment variables in the configuration file). If someone wants more, then he may mount a Docker volume and override the complete configuration file.

There is one additional benefit of packing services as Docker images. Each image is identified by a unique identifier that may be used for regulatory purposes. This identifier encompasses the whole bundle, and you can be sure that even a tiny change in the build would drastically alter the generated identifier. In some sense, this is like digitally marking the image.

For creating the actual image in the usual mvn clean package manner, we need to add the following dependency to the build file:

```
<plugin>
    <groupId>com.spotify</groupId>
    <artifactId>dockerfile-maven-plugin</artifactId>
    <version>1.3.6</version>
    <executions>
        <execution>
            <id>default</id>
            <goals>
                <goal>build</goal>
            </goals>
        </execution>
    </executions>
    <configuration>
        <repository>
            ${enterprise.docker.registry}/qa/swaggerize
        </repository>
        <tag>${project.version}</tag>
    </configuration>
</plugin>
```

You may redefine the `enterprise.docker.registry` property to point to your own private Docker registry (see Exercise 7). The Swaggerize service can be started with the following command:

```
docker run -d\  
  --name swaggerize\  
  --restart=always\  
  -e GRAPHITE_HOST=<read the IP of the infrastructure VM 2>\  
  -e SYSLOG_HOST=<read the IP of the infrastructure VM 2>\  
  -e DB_HOST=<read the IP of the infrastructure VM 1>\  
  -e DB_USERNAME=admin  
  -e DB_PASSWORD='<your admin password>'\  
  -e DB_PORT=3306\  
  -e DB_SCHEMA=Swaggerize_DB\  
  -p 8080:8080 -p 8081:8081\  
  --net=host\  
  com.example.docker_registry/qa/swaggerize:2.2.0
```

If you want to get rid of the leading URL (and only use the standard Docker Hub), then you should move the slash into the property and define the image name as `${enterprise.docker.registry}qa/swaggerize`. Now, if you keep the property's default value as empty, then your image name would become simply `qa/swaggerize:2.2.0`. My default is to use a personal Docker registry, unless I decide to make something public.

10.6 Summary

This chapter has explained what is behind the concept of DevOps and has illustrated the automation in two stages. The ideas presented here are purely scratching the surface of available possibilities. The next logical step could be to control the deployment/delivery cycle via a build server (Jenkins is one popular open-source choice). The key takeaways from this chapter are as follows:

- DevOps is a paradigm that brings forward deployability and observability as architecturally significant quality attributes.
- Tool support is excruciatingly important to gain pragmatic benefits from good ideas. There is a flourishing ecosystem of tools around DevOps, and we have demonstrated a couple of them.
- There are lots of misconceptions surrounding the term DevOps (as is the case with TDD), and it isn't about vehemently trying to automate everything. DevOps is about quality assurance, where we want to proactively prevent faults from producing a dependable product.

10.7 Exercises

1. Covering your code base with automated tests is one of the crucial preconditions toward further automation. Introduce a code coverage analyzer into Swaggerize, and evaluate the quality of existing tests. For sectors of the code that are improperly covered, write more tests. It is well known that untested code is more error prone. There is no better way to automate your system than by reducing the number of faults in the shipped product.
2. (Project assignment) The correlation identifier in the logs may also be used to find out the source of a request (another possibility is to use the User-Agent request header in logs). For example, if an agent calls the Swaggerize service through a Web browser, then the matching correlation identifier will be an independent random value. However, if another service calls Swaggerize to register itself, then the correlation identifier in the logs will have the same value as set by the calling party. Now, try to set alarms for warnings and errors if they are caused by software services (you may utilize your favorite logging tool). Such events may indicate a bug in those services regarding how they use the Swaggerize service's API.
3. Try out the other virtualization providers like VMWare. Notice that you just need to alter the build command and specify `vmware-iso`. Of course, if you omit the `-only` flag, then Packer will summon the build for all supported providers (see the template in the book's source code).
4. Add a post-processor to the template to upload the generated box file into your private AWS S3 bucket (unless you decide to offer it publicly). You would then be able to download it from anywhere with an Internet connection. You can also set up Vagrant to do this for automatically.
5. It is possible to use a secret Vagrant feature for auto-installing the required plugins. Extend the `infrastructure-setup` project to use this facility (see dustingram.com/articles/2015/09/16/vagrant-secret-feature).
6. (Project assignment) Deploy the ELK stack in a separate VM (it would be the third one), and configure Logstash to accept Syslog events. You would want to update the base image to pull the ELK Docker images.
7. (Project assignment) Set up a private Docker registry (it is already available as an official Docker image called `registry`), and push the Swaggerize image there. You may do that by simply invoking `mvn dockerfile:push` after the image had been built.

References

Further Reading

1. Daniels K, Davis J (2016) Effective DevOps. O'Reilly Media, Sebastopol, CA. Explains the context surrounding DevOps, and how to organize product and project management activities to successfully interconnect development with operations. This book together with [2] accentuates

- the magnitude of software engineering elements that must be in-place to attain the goals of DevOps
- 2. Bass L, Weber I, Zhu L (2015) DevOps: a software architect's perspective. Addison-Wesley Professional, Old Tappan, NJ. This book connects DevOps with architecture, and emphasizes the importance to treat deployability and observability as key quality attributes of the system. The book gives you both a theoretical background, and practical case studies to fully appreciate DevOps as a true paradigm, rather than just a set of cool technologies and tools
 - 3. Humble J, Farley D (2010) Continuous delivery: reliable software releases through build, test, and deployment automation, Video Enhanced Edition. Addison-Wesley Professional, Upper Saddle River, NJ. This book complements [1–2] by guiding you through the ecosystem of tools to realize the vision of DevOps. It nicely illustrates how a balanced set of tools are indispensable to make DevOps a reality

Regular Bibliographic References

- 4. Roche J (2013) Adopting DevOps practices in quality assurance. Commun ACM 56(11):38–43

Index

A

- Abstract data type, 79
- ACME, 74
- Adaptive
 - maintenance category, 5

B

- Bag of tasks, 283
- Behavior-driven development, 258
- Bonferroni's principle, 301

C

- Cause/effect diagram, 61
- Code of ethics, 78
- Cohesion, 218
- Composition over inheritance, 142
- Consumer-driven contracts, 83
- Context, 10
- Correction
 - maintenance category, 5
- Corrective
 - maintenance category, 5
- Correlation *vs.* causation, 301
- Cross-origin resource sharing, 166
- Cyclomatic complexity, 19, 33

D

- Davis's principles, 39
- Defensive programming, 49, 91
- De Morgan's law, 55
- Dependency injection, 61
- Deviation principle, 26
- Don't repeat yourself, 120

E

- Encapsulation, 79
- Enhancement
 - maintenance category, 5
- Epistemic complexity, 9
- Essence, 4
- E type system, 6
- Exploratory test, 211

F

- Factory method, 49, 121
- Fail-fast, 91
- Failures, 5
- Faults, 5
- Feasibility study, 166
- Fishbone, *see* Cause/effect diagram
- Framework, 54

G

- God class, 248

H

- Half sync-half async pattern, 283
- Heisenbug, 100

I

- Information hiding, 79
- Infrastructure as code, 324
- Intentional programming, 36
- Inversion of control, 60
- Is-a relationship, 79
- Ishikawa, *see* Cause/effect diagram

K

Kano model, 163

L

Layered approach, 41

Learning path, 10

Liskov substitution principle, 79

M

Maintainability, 4

Maintenance activity, 5

Maintenance phase, 5

Model-view-controller pattern, 219

N

N-version programming, 250

O

Optimistic locking, 97

P

Parkinson's law, 7

Perfective

 maintenance category, 5

Pessimistic locking, 97

Preventive

 maintenance category, 5

Primal-dual method, 261

P type system, 6

Q

Quality, 243

 assurance, 12

 attributes, 11

 control, 12

R

Read-time convenience, 53, 80

Reuse, 54

S

Safety argument, 32

Separation of concerns, 62, 218

Service table, 167

Single level of abstraction, 34

Smoke testing, 74

Software architecture, 40

Software engineering method and theory
(SEMAT), 4

Strategy design pattern, 144

Stratified (layered) design, 35

S type system, 6

Sustainability, 247

System requirements, 159

T

Technical debt, 6

Template design pattern, 144

Test-driven development (TDD), 41

Total cost of ownership, 6

U

UML use case diagram, 164

Usability, 219

User requirements, 159

V

Validation, 32

Verification, 32

W

Warning suppression, 80

Write-time convenience, 80

See also Read-time convenience