O'REILLY®

Java Edition

Building Maintainable Software

TEN GUIDELINES FOR FUTURE-PROOF CODE

Free Sampler

Joost Visser

Building Maintainable Software: Ten Guidelines for Future-Proof Code
by Joost Visser

Copyright © 2016 Software Improvement Group, B.V. All rights reserved.

Printed in the United States of America.

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

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

Acquisitions Editor: Rachel Roumeliotis
Editor: Nan Barber
Production Editor: Matthew Hacker
Copyeditor: Rachel Monaghan
Proofreader: Marta Justak
Indexer: WordCo Indexing Services, Inc.
Interior Designer: David Futato
Cover Designer: Randy Comer
Illustrator: Rebecca Demarest

February 2016: First Edition

Revision History for the First Edition
2016-01-25: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491940662* for release details.

978-1-4919-5352-5

[LSI]

Table of Contents

CHAPTER 1
Introduction

Who wrote this piece of code?? I can't work like this!!

—Any programmer

Being a software developer is great. When someone gives you a problem and requirements, you are able to come up with a solution and translate that solution into a language that a computer understands. These are challenging and rewarding endeavors. Being a software developer can also be a painstaking job. If you regularly have to change source code written by others (or even by yourself), you know that it can be either really easy or really difficult. Sometimes, you can quickly identify the lines of code to change. The change is nicely isolated, and tests confirm that it works as intended. At other times, the only solution is to use a hack that creates more problems than it solves.

The ease or difficulty with which a software system can be modified is known as its *maintainability*. The maintainability of a software system is determined by properties of its source code. This book discusses these properties and presents 10 guidelines to help you write source code that is easy to modify.

In this chapter, we explain what we mean when we speak about maintainability. After that, we discuss why maintainability is important. This sets the stage to introduce the main topic of this book: how to build software that is maintainable from the start. At the end of this introduction we discuss common misunderstandings about maintainability and introduce the principles behind the 10 guidelines presented in this book.

1.1 What Is Maintainability?

Imagine two different software systems that have exactly the same functionality. Given the same input, both compute exactly the same output. One of these two systems is fast and user-friendly, and its source code is easy to modify. The other system is slow and difficult to use, and its source code is nearly impossible to understand, let alone modify. Even though both systems have the same functionality, their quality clearly differs.

Maintainability (how easily a system can be modified) is one characteristic of software quality. Performance (how slow or fast a system produces its output) is another.

The international standard ISO/IEC 25010:2011 (which we simply call ISO 25010 in this book[1]) breaks down software quality into eight characteristics: *maintainability*, *functional suitability*, *performance efficiency*, *compatibility*, *usability*, *reliability*, *security*, and *portability*. This book focuses exclusively on maintainability.

Even though ISO 25010 does not describe how to measure software quality, that does not mean you cannot measure it. In Appendix A, we present how we measure software quality at the Software Improvement Group (SIG) in accordance with ISO 25010.

The Four Types of Software Maintenance

Software maintenance is not about fixing wear and tear. Software is not physical, and therefore it does not degrade by itself the way physical things do. Yet most software systems are modified all the time after they have been delivered. This is what software maintenance is about. Four types of software maintenance can be distinguished:

- Bugs are discovered and have to be fixed (this is called *corrective maintenance*).
- The system has to be adapted to changes in the environment in which it operates—for example, upgrades of the operating system or technologies (this is called *adaptive maintenance*).
- Users of the system (and/or other stakeholders) have new or changed requirements (this is called *perfective maintenance*).
- Ways are identified to increase quality or prevent future bugs from occurring (this is called *preventive maintenance*).

1 Full title: *International Standard ISO/IEC 25010. Systems and Software Engineering -- Systems and software Quality Requirements and Evaluation (SQuaRE) -- System and Software Quality Models.* First Edition, 2011-03-01.

1.2 Why Is Maintainability Important?

As you have learned, maintainability is only one of the eight characteristics of software product quality identified in ISO 25010. So why is maintainability so important that it warrants its own, dedicated book? There are two angles to this question:

- Maintainability, or lack thereof, has significant business impact.
- Maintainability is an enabler for other quality characteristics.

Both angles are discussed in the next two sections.

Maintainability Has Significant Business Impact

In software development, the maintenance phase of a software system often spans 10 years or more. During most of this time, there is a continuous stream of issues that need to be resolved (corrective and adaptive maintenance) and enhancement requests that have to be met (perfective maintenance). The efficiency and effectiveness with which issues can be resolved and enhancements can be realized is therefore important for stakeholders.

Maintenance efforts are reduced when issue resolution and enhancements can be performed quickly and easily. If efficient maintenance leads to less maintenance personnel (developers), it also lowers maintenance costs. When the number of developers stays the same, with efficient maintenance they have more time for other tasks, such as building new functionality. Fast enhancements mean shorter time-to-market of new products and services supported by the system. For both issue resolution and enhancements, it holds that if they are slow and troublesome, deadlines may not be met or the system may become unusable.

SIG has collected empirical evidence that issue resolution and enhancements are twice as fast in systems with above-average maintainability than in systems with below-average maintainability. A factor of two is a significant quantity in the practice of enterprise systems. The time it takes to resolve issues and make an enhancement is on the order of days or weeks. It is not the difference between fixing 5 bugs or 10 in an hour; it is the difference between being the first one to the market with a new product, or seeing your competitor months ahead of you.

And that is just the difference between above-average and below-average maintainability. At SIG we have seen newly built systems for which the maintainability was so low that it was no longer possible to effectively modify them—even before the systems went into production. Modifications introduced more bugs than they solved. Development took so long that the business environment (and therefore, user requirements) had already changed. More modifications were needed, which introduced yet more bugs. More often than not, such systems are written off before they ever see a 1.0 release.

Maintainability Is an Enabler for Other Quality Characteristics

Another reason why maintainability is a special aspect of software quality is that it acts as an enabler for other quality characteristics. When a system has high maintainability, it is easier to make improvements in the other quality areas, such as fixing a security bug. More generally speaking, optimizing a software system requires modifications to its source code, whether for performance, functional suitability, security, or any other of the seven nonmaintainability characteristics defined by ISO 25010.

Sometimes they are small, local modifications. Sometimes they involve more invasive restructuring. All modifications require finding a specific piece of source code and analyzing it, understanding its inner logic and its position in the business process that the system facilitates, analyzing dependencies between different pieces of code and testing them, and pushing them through the development pipeline. In any case, in a more maintainable system, these modifications are easier to make, allowing you to implement quality optimizations faster and more effectively. For example, highly maintainable code is more stable than unmaintainable code: changes in a highly maintainable system have fewer unexpected side effects than changes in an entangled system that is hard to analyze and test.

1.3 Three Principles of the Guidelines in This Book

If maintainability is so important, how can you improve maintainability of the code that you write? This book presents 10 guidelines that, if followed, lead to code that is highly maintainable. In the following chapters, each guideline is presented and discussed. In the current chapter, we introduce the principles behind these guidelines:

1. Maintainability benefits most from adhering to simple guidelines.
2. Maintainability is not an afterthought, but should be addressed from the very beginning of a development project. Every individual contribution counts.
3. Some violations are worse than others. The more a software system complies with the guidelines, the more maintainable it is.

These principles are explained next.

Principle 1: Maintainability Benefits Most from Simple Guidelines

People may think that maintainability requires a “silver bullet”: one technology or principle that solves maintainability once and for all, *automagically*. Our principle is the very opposite: maintainability requires following simple guidelines that are not sophisticated at all. These guidelines guarantee sufficient maintainability, not perfect maintainability (whatever that may be). Source code that complies with these guidelines can still be made more maintainable. At some point, the additional gains in maintainability become smaller and smaller, while the costs become higher and higher.

Principle 2: Maintainability Is Not an Afterthought, and Every Contribution Counts

Maintainability needs to be addressed from the very start of a development project. We understand that it is hard to see whether an individual "violation" of the guidelines in this book influences the overall maintainability of the system. That is why *all* developers must be disciplined and follow the guidelines to achieve a system that is maintainable overall. Therefore, your individual contribution is of great importance to the whole.

Following the guidelines in this book not only results in more maintainable code, but also sets the right example for your fellow developers. This avoids the "broken windows effect" in which other developers temporarily relax their discipline and take shortcuts. Setting the right example is not necessarily about being the most skilled engineer, but more about retaining discipline during development.

Remember that you are writing code not just for yourself, but also for less-experienced developers that come after you. This thought helps you to simplify the solution you are programming.

Principle 3: Some Violations Are Worse Than Others

The guidelines in this book present metric thresholds as an absolute rule. For instance, in Chapter 2, we tell you to never write methods that have more than 15 lines of code. We are fully aware that in practice, almost always there will be exceptions to the guideline. What if a fragment of source code violates one or more of these guidelines? Many types of tooling for software quality assume that each and every violation is bad. The hidden assumption is that all violations should be resolved. In practice, resolving all violations is neither necessary nor profitable. This all-or-nothing view on violations may lead developers to ignore the violations altogether.

We take a different approach. To keep the metrics simple but also practical, we determine the quality of a complete codebase not by the code's number of violations but by its *quality profiles*. A quality profile divides metrics into distinct categories, ranging from fully compliant code to severe violations. By using quality profiles, we can distinguish moderate violations (for example, a method with 20 lines of code) from severe violations (for example, a method with 200 lines of code). After the next section, which discusses common misunderstandings about maintainability, we explain how quality profiles are used to measure the maintainability of a system.

1.4 Misunderstandings About Maintainability

In this section, we discuss some misunderstandings about maintainability that are encountered in practice.

Misunderstanding: Maintainability Is Language-Dependent

"Our system uses a state-of-the-art programming language. Therefore, it is at least as maintainable as any other system."

The data we have at SIG does not indicate that the technology (programming language) chosen for a system is the dominant determining factor of maintainability. Our dataset contains Java systems that are among the most maintainable, but also, that are among the least maintainable. The average maintainability of all Java systems in our benchmark is itself average, and the same holds for C#. This shows us that it is possible to make very maintainable systems in Java (and in C#), but using either of these languages does not guarantee a system's maintainability. Apparently, there are other factors that determine maintainability.

For consistency, we are using Java code snippets throughout the book. However, the guidelines described in this book are not specific to Java. In fact, SIG has benchmarked systems in over a hundred programming languages based on the guidelines and metrics in this book.

Misunderstanding: Maintainability Is Industry-Dependent

"My team makes embedded software for the car industry. Maintainability is different there."

We believe that the guidelines presented in this book are applicable to all forms of software development: embedded software, games, scientific software, software components such as compilers and database engines, and administrative software. Of course, there are differences between these domains. For example, scientific software often uses a special-purpose programming language, such as R, for statistical analysis. Yet, in R, it is a good idea to keep units short and simple. Embedded software has to operate in an environment where performance predictability is essential and resources are constrained. So whenever a compromise has to be made between performance and maintainability, the former wins over the latter. But no matter the domain, the characteristics defined in ISO 25010 still apply.

Misunderstanding: Maintainability Is the Same as the Absence of Bugs

"You said the system has above-average maintainability. However, it turns out it is full of bugs!"

According to the ISO 25010 definitions, a system can be highly maintainable and still be lacking in other quality characteristics. Consequently, a system may have above-average maintainability and still suffer from problems regarding functional suitability, performance, reliability, and more. Above-average maintainability means nothing more than that the modifications needed to reduce the number of bugs can be made at a high degree of efficiency and effectiveness.

Misunderstanding: Maintainability Is a Binary Quantity

"My team repeatedly has been able to fix bugs in this system. Therefore, it has been proven that it ***is*** *maintainable."*

This distinction is important. "Maintain-Ability" is literally the ability to maintain. According to its definition in ISO 25010, source code maintainability is not a binary quantity. Instead, maintainability is the degree to which changes can be made efficiently and effectively. So the right question to ask is not whether changes (such as bug fixes) have been made, but rather, how much effort did fixing the bug take (efficiency), and was the bug fixed correctly (effectiveness)?

Given the ISO 25010 definition of maintainability, one could say that a software system is never perfectly maintainable nor perfectly *un*maintainable. In practice, we at SIG have encountered systems that can be considered unmaintainable. These systems had such a low degree of modification efficiency and effectiveness that the system owner could not afford to maintain it.

1.5 Rating Maintainability

We know now that maintainability is a quality characteristic on a scale. It signifies different degrees of being able to maintain a system. But what is "easy to maintain" and what is "hard to maintain"? Clearly, a complex system is easier to maintain by an expert than by a less experienced developer. By benchmarking, at SIG we let the metrics in the software industry answer this question. If software metrics for a system score below average, it is harder than average to maintain. The benchmark is recalibrated yearly. As the industry learns to code more efficiently (e.g., with the help of new technologies), the average for metrics tends to improve over time. What was the norm in software engineering a few years back, may be subpar now. The benchmark thus reflects the state of the art in software engineering.

SIG divides the systems in the benchmark by star rating, ranging from 1 star (hardest to maintain) to 5 stars (easiest to maintain). The distribution of these star ratings among systems from 1 to 5 stars is 5%-30%-30%-30%-5%. Thus, in the benchmark the systems that are among the top 5% are rated 5 stars. In these systems, there are still violations to the guidelines, but much fewer than in systems rated below.

The star ratings serve as a predictor for actual system maintainability. SIG has collected empirical evidence that issue resolution and enhancements are twice as fast in systems with 4 stars than in systems with 2 stars.

The systems in the benchmark are ranked based on their metric quality profiles. Figure 1-1 shows three examples of unit size quality profiles (print readers can view full-color figures for this and the other quality profiles that follow in our repository for this book).

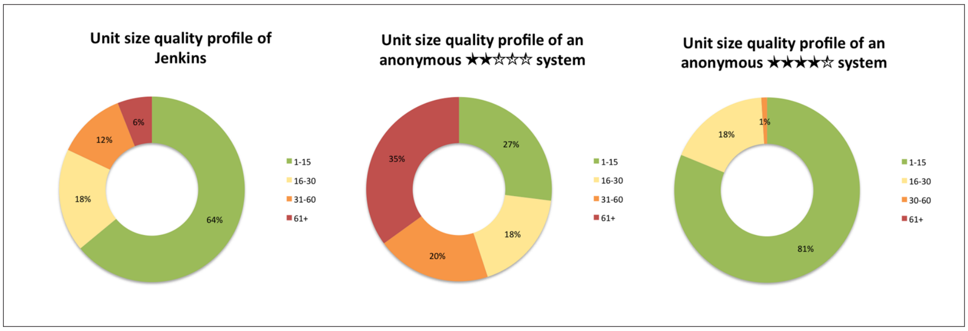

Figure 1-1. Example of three quality profiles

The first chart in Figure 1-1 is a quality profile for unit size based on the source code of Jenkins version 1.625, a popular open source continuous integration server. The quality profile tells us that the Jenkins codebase has 64% of its code in methods that are no longer than 15 lines of code (compliant with the guideline). The profile also shows that 18% of all the code in the codebase is in methods between 16 and 30 lines of code, and 12% is in methods between 31 and 60 lines of code. The Jenkins codebase is not perfect. It has severe unit size violations: 6% of the codebase is in very long units (more than 60 lines of code).

The second chart in Figure 1-1 shows the quality profile of a 2-star system. Notice that over one-third of the codebase is in units that are over 60 lines of code. Doing maintenance on this system is a very painstaking job.

Finally, the third chart in Figure 1-1 shows the unit size cutoff points for 4 stars. Compare this chart to the first one. You can tell that Jenkins complies to the unit size guideline for 4 stars (although not for 5 stars), since the percentages of code in each category are lower than the 4-star cutoffs.

In a sidebar at the end of each guideline chapter, we present the quality profile categories for that guideline as we use them at SIG to rate maintainability. Specifically, for each guideline, we present the cutoff points and the maximum percentage of code in each category for a rating of 4 stars or higher (top 35% of the benchmark).

1.6 An Overview of the Maintainability Guidelines

In the following chapters, we will present the guidelines one by one, but here we list all 10 guidelines together to give you a quick overview. We advise you to read this book starting with Chapter 2 and work your way through sequentially.

Write short units of code (Chapter 2)
: Shorter units (that is, methods and constructors) are easier to analyze, test, and reuse.

Write simple units of code (Chapter 3)
: Units with fewer decision points are easier to analyze and test.

Write code once (Chapter 4)
: Duplication of source code should be avoided at all times, since changes will need to be made in each copy. Duplication is also a source of regression bugs.

Keep unit interfaces small (Chapter 5)
: Units (methods and constructors) with fewer parameters are easier to test and reuse.

Separate concerns in modules (Chapter 6)
: Modules (classes) that are loosely coupled are easier to modify and lead to a more modular system.

Couple architecture components loosely (Chapter 7)
: Top-level components of a system that are more loosely coupled are easier to modify and lead to a more modular system.

Keep architecture components balanced (Chapter 8)
: A well-balanced architecture, with not too many and not too few components, of uniform size, is the most modular and enables easy modification through separation of concerns.

Keep your codebase small (Chapter 9)
: A large system is difficult to maintain, because more code needs to be analyzed, changed, and tested. Also, maintenance productivity *per line of code* is lower in a large system than in a small system.

Automate development pipeline and tests (Chapter 10)
: Automated tests (that is, tests that can be executed without manual intervention) enable near-instantaneous feedback on the effectiveness of modifications. Manual tests do not scale.

Write clean code (Chapter 11)
: Having irrelevant artifacts such as TODOs and dead code in your codebase makes it more difficult for new team members to become productive. Therefore, it makes maintenance less efficient.