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

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 sparkles 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