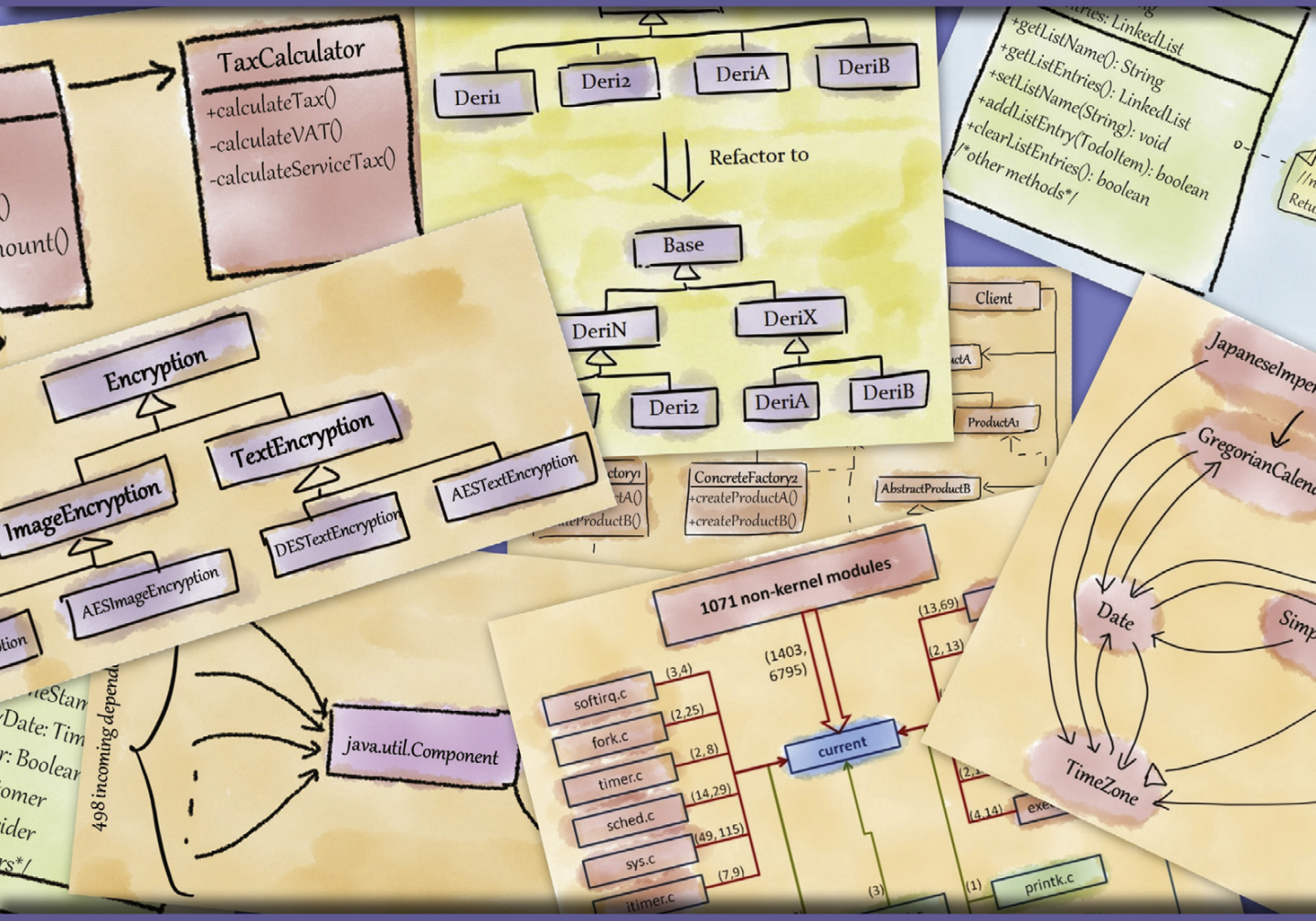
# Refactoring for Software Design Smells

## Managing Technical Debt





Girish Suryanarayana,
Ganesh Samarthyam, Tushar Sharma


Forewords by Grady Booch and Stéphane Ducasse


MORGAN KAUFMANN

# Refactoring for Software Design Smells

This page intentionally left blank

# Refactoring for Software Design Smells

## Managing Technical Debt

**Girish Suryanarayana**

**Ganesh Samarthyam**

**Tushar Sharma**

ELSEVIER

AMSTERDAM • BOSTON • HEIDELBERG • LONDON
NEW YORK • OXFORD • PARIS • SAN DIEGO
SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO

Morgan Kaufmann is an imprint of Elsevier

MK
MORGAN KAUFMANN

**Acquiring Editor:** Todd Green
**Editorial Project Manager:** Lindsay Lawrence
**Project Manager:** Punithavathy Govindaradjane
**Designer:** Mark Rogers

*Morgan Kaufmann* is an imprint of Elsevier
225 Wyman Street, Waltham, MA, 02451, USA



**Notices**

Knowledge and best practice in this field are constantly changing. As new research and experience broaden our understanding, changes in research methods, professional practices, or medical treatment may become necessary.

Practitioners and researchers must always rely on their own experience and knowledge in evaluating and using any information, methods, compounds, or experiments described herein. In using such information or methods they should be mindful of their own safety and the safety of others, including parties for whom they have a professional responsibility.

To the fullest extent of the law, neither the Publisher nor the authors, contributors, or editors, assume any liability for any injury and/or damage to persons or property as a matter of products liability, negligence or otherwise, or from any use or operation of any methods, products, instructions, or ideas contained in the material herein.

**Library of Congress Cataloging-in-Publication Data**
Suryanarayana, Girish
Refactoring for software design smells: managing technical debt/Girish Suryanarayana, Ganesh Samarthyam, Tushar Sharma.
pages cm
Includes bibliographical references and index.
ISBN: 978-0-12-801397-7 (paperback)
1. Software refactoring. 2. Software failures. I. Samarthyam, Ganesh. II. Sharma, Tushar. III. Title.

QA76.76.R42S86 2015
005.1′6–dc23

2014029955

**British Library Cataloguing-in-Publication Data**
A catalogue record for this book is available from the British Library.

ISBN: 978-0-12-801397-7

*To Amma, Anna, Vibha, and Sai Saadhvi*
*-Girish*

*To my little princess Tanmaye*
*-Ganesh*

*To Papa, Mummy, Pratibha, and Agastya*
*-Tushar*

This page intentionally left blank

# Contents

# Foreword by Grady Booch

There is a wonderful book by Anique Hommels called *Unbuilding Cities: Obduracy in Urban Sociotechnical Change* that speaks of the technical, social, economic, and political issues of urban renewal. Cities grow, cities evolve, cities have parts that simply die while other parts flourish; each city has to be renewed in order to meet the needs of its populace. From time to time cities are intentionally refactored (many cities in Europe were devastated in World War II and so had to be rebuilt); most times cities are refactored in bits (the Canary Warf renewal in London comes to mind), but sometimes the literal and figurative debt is so great a once great city falls into despair (present day Detroit, for example).

Software-intensive systems are like that. They grow, they evolve, sometimes they wither away, and sometimes they flourish. The concept of technical debt is central to understanding the forces that weigh upon systems, for it often explains where, how, and why a system is stressed. In cities, repairs on infrastructure are often delayed and incremental changes are made rather than bold ones. So it is again in software-intensive systems. Users suffer the consequences of capricious complexity, delayed improvements, and insufficient incremental change; the developers who evolve such systems suffer the slings and arrows of never being able to write quality code because they are always trying to catch up.

What delights me about this present book is its focus on technical debt and refactoring as the actionable means to attend to it.

When you have got a nagging tiny gas leak on a city block, a literal smell will lead you to the underlying cause. Software-intensive systems are like that as well, although the smells one may observe therein are far more subtle and invisible to all the senses save to the clever cognitive ones. As you read on in this book, I think you will find one of the more interesting and complex expositions of software smells you will ever find. Indeed, you have in your hand a veritable field guide of smells, sort of like the very real *Birds of the West Indies*, except about software, not sparrows.

Abstraction, encapsulation, modularization, and hierarchy: these are all elements fundamental to software engineering best practices, and you will see these highlights explained as well. In all, I think you will find this a delightful, engaging, actionable read.

Grady Booch
*IBM Fellow and Chief Scientist for*
*Software Engineering, IBM Research*

This page intentionally left blank

# Foreword by Dr. Stéphane Ducasse

Working with real and large object-oriented applications made me and my coauthors (O. Nierstrasz and S. Demeyer) think about object-oriented reengineering patterns. Our idea was to record good approaches to handle these systems at the design level but also with a bit of process. This is how Object-Oriented Reengineering Patterns (http://scg.unibe.ch/download/oorp/) came to life after 3 years of intensive work. In the same period I reviewed the book "Refactoring: Improving the Design of Existing Code" by Martin Fowler because we were working on language-independent refactorings. To me, books like these are milestones. They create a foundation for our engineering science.

When as JOT associate editor, I saw the authors' article describing the work presented in this book, I became an immediate supporter. I immediately asked if they would write a book and if I could be a reviewer. Why? Because a good book on Smells was still missing. And we daily face code smells. "Refactoring for Software Design Smells" is an excellent book. It is another milestone that professionals will use. It captures the deep knowledge experts have when facing real code. It puts in shape and word the complex knowledge that experts acquire after years of experience. I daily program either Pharo runtime and libraries or the Moose software and data analysis platform, and there is not a single day where I'm not challenged by design. Why? Because design, real design, is not simple. Even with a lot of practice and experience design is challenging. This is why having good abstractions to play with is important. Patterns and smells are really effective ways to capture such experience and reflection about the field.

I like the idea that I have access to excellent books that I can suggest to my students, colleagues, and friends. I like books that fill you up and make you think about your own practices. There are few of such ones and they are like gems. And "Refactoring for Software Design Smells" is one of those.

It is the book I would have loved to write (but I probably would not have been able to) as a complement to the Fowler's refactoring book, and my Object-Oriented Reengineering Patterns book. I'm sure that you will learn a lot from it and that you will enjoy it. I want to thank Girish, Ganesh, and Tushar (the authors of this book) to have spent their time to write it. I know that the process is long and tedious but this is an important part of our culture and knowledge that they capture and offer to you. I also want to thank them for their invitation to write this foreword.

Stéphane Ducasse

(Dr. Stéphane Ducasse is an expert in and fond of object design, language design, and reflective programming. He has contributed to Traits which got introduced in Pharo, Perl-6, PHP 5.4, and Squeak and influenced Fortress and Scala. He is one of the lead developers of Pharo (http://www.pharo.project.org/), an open-source live programming language, IDE and platform. He is an expert in software analysis and reengineering. He is one of the developers of the Moose analysis platform http://www.moosetechnology.org and he recently cocreated http://www.synectique.eu, a company delivering advanced tools adapted to client problems.)

This page intentionally left blank

# Preface

*As a program is evolved its complexity increases unless work is done to maintain or reduce it.*

**Lehman's law of Increasing Complexity [1]**

## WHAT IS THIS BOOK ABOUT?

Change is inevitable and difficult! This is true not only about life but also about software. Software is expected to evolve continuously to meet the ever-increasing demands of its users. At the same time, the intangible nature of software makes it difficult to manage this continuous change. What typically results is poor software quality[1] and a huge technical debt.

This book tells you how to improve software quality and reduce technical debt by discovering and addressing smells in your design. Borrowing a phrase from the health care domain "a good doctor is one who knows the medicines but a great doctor is one who knows the disease," our approach is grounded on the philosophy that "a good designer is one who knows about design solutions but a great designer is one who understands the problems (or smells) in the design, how they are caused, and how they can be addressed by applying proven and sound design principles." The goal of this book is, therefore, to guide you into becoming a better designer—one who can recognize and understand the "disease" in his design, and can treat it properly, thereby, improving the quality of the software and keeping technical debt under control.

## WHAT DOES THIS BOOK COVER?

This book presents a catalog of 25 structural design smells and their corresponding refactoring towards managing technical debt. We believe that applying software design principles is the key to developing high-quality software. We have, therefore, organized our smell catalog around four basic design principles. Smells are named after the specific principle they violate. The description of each smell reveals the design principle that the smell violates, discusses some factors that can cause that smell to occur, and lists the key quality attributes that are impacted by the smell. This allows the reader to get an idea of the technical debt incurred by the design.

[1] Capers Jones [3] finds that poor software quality costs more than US $150 billion per year in the United States and greater than US $500 billion per year worldwide.

Each smell description also includes real-world examples as well as anecdotes based on experience with industrial projects. Accompanying each example are potential refactoring solutions that help address the particular instance of the smell. We believe that the impact of a smell can only be judged based on the design context in which it occurs. Therefore, we also explicitly consider situations wherein a smell may be purposely introduced either due to constraints (such as language or platform limitations) or to realize an overarching purpose in the design.

Smells can be found at different levels of granularity, including architecture and code. Similarly, smells can be either structural or behavioral in nature. Rather than surveying a wide range of smells pertaining to different levels of granularity and nature, we focus only on "structural" and "design-level" smells in this book. Further, the book discusses only those smells that are commonly found in real-world projects and have been documented in literature.

## WHO SHOULD READ THIS BOOK?

**Software Architects and Designers**—If you are a practicing software architect or a designer, you will get the most out of this book. This book will benefit you in multiple ways. As an architect and designer, your primary responsibility is the software design and its quality, and you (more than anyone else) are striving to realize quality requirements in your software. The knowledge of design smells and the suggested refactorings covered in this book will certainly help you in this regard; *they offer you immediately usable ideas for improving the quality of your design*. Since this book explicitly describes the impact of smells on key quality attributes, you will gain a deeper appreciation of how even the smallest design decision has the potential to significantly impact the quality of your software. Through the real-world anecdotes and case studies, you will become aware of what factors you should be careful about and plan for in order to avoid smells.

**Software Developers**—We have observed in practice that often developers take shortcuts that seemingly get the work done but compromise on the design quality. We have also observed that sometimes when the design does not explicitly address certain aspects, it is the developer who ends up making key design decisions while coding. In such a case, if design principles are incorrectly applied or not applied at all, smells will arise. We believe that reading this book will help developers realize how their seemingly insignificant design decisions or shortcuts can have an immense impact on software quality. This realization will help them transform themselves into better developers who can detect and address problems in the design and code.

**Project Managers**—Project managers constantly worry about the possible schedule slippages and cost overruns of their projects. There is an increased awareness today that often the primary reason for this is technical debt, and many project managers are, therefore, always on the look out for solutions to reduce technical

debt. Such project managers will benefit from reading this book; they will gain a better understanding of the kinds of problems that manifest in the design and have an increased appreciation for refactoring.

**Students**—This book is very relevant to courses in computer science or software engineering that discuss software design. A central focus of any software design course is the fundamental principles that guide the modeling and design of high-quality software. One effective way to learn about these principles is to first study the effects (i.e., smells) of wrong application or misapplication of design principles and then learn about how to apply them properly (i.e., refactoring). We, therefore, believe that reading this book will help students appreciate the value of following good design principles and practices and prepare them to realize high-quality design in real-world projects post completion of their studies.

## WHAT ARE THE PREREQUISITES FOR READING THIS BOOK?

We expect you to have basic knowledge of object-oriented programming and design and be familiar with at least one object-oriented language (such as C++, Java, or C#). We also expect you to have knowledge of object-oriented concepts such as class, abstract class, and interface (which can all be embraced under the umbrella terms "abstraction" or "type"), inheritance, delegation, composition, and polymorphism.

## HOW TO READ THIS BOOK?

This book is logically structured into the following three parts:

- **Chapters 1 and 2** set the context for this book. Chapter 1 introduces the concept of technical debt, the factors that contribute to it, and its impact on software projects. Chapter 2 introduces design smells and describes the principle-based classification scheme that we have used to categorize and name design smells in this book.
- **Chapters 3–6** present the catalog of 25 design smells. The catalog is divided into four chapters that correspond to the four fundamental principles (abstraction, encapsulation, modularization, and hierarchy) that are violated by the smells. For each design smell, we provide illustrative examples, describe the impact of the smells on key quality attributes, and discuss the possible refactoring for that smell in the context of the given examples.
- **Chapters 7 and 8** present a reflection of our work and give you an idea about how to repay technical debt in your projects. Chapter 7 revisits the catalog to pick up examples and highlight the interplay between smells and the rest of the design in which the smells occur. Chapter 8 offers practical guidance and tips on how to approach refactoring of smells to manage technical debt in real-world projects.

The appendices contain a listing of design principles referenced in this book, a list of tools that can help detect and address design smells, a brief summary of the UML-like notations that we have used in this book, and a suggested reading list to help augment your knowledge of software design.

Given the fact that Java is the most widely used object-oriented language in the world today, we have provided coding examples in Java. However, the key take-away is in the context of design principles, and is therefore applicable to other object-oriented languages such as C++ and C# as well. Further, we have used simple UML-like class diagrams in this book which are explained in Appendix C.

Many of the examples discussed in this book are from JDK version 7.0. We have interpreted the smells in JDK based on the limited information we could glean by analyzing the source code and the comments. It is therefore possible that there may be perfectly acceptable reasons for the presence of these smells in JDK.

We have shared numerous anecdotes and case studies throughout this book, which have been collected from the following different sources:

- Experiences reported by participants of the online *Smells Forum*, which we established to collect smell stories from the community.
- Incidents and stories that have been shared with us by fellow attendees at conferences, participants during guest lectures and trainings, and via community forums.
- Books, journals, magazines, and other online publications.
- Experiences of one of the authors who works as an independent consultant in the area of design assessment and refactoring.

We want to emphasize that our goal is *not* to point out at any particular software organization and the smells in their design through our anecdotes, case studies, or examples. Rather, our sole goal is to educate software engineers about the potential problems in software design. Therefore, it should be noted that the details in the anecdotes and case studies reported in this book have been modified suitably so that confidential details such as the name of the organization, project, or product are not revealed.

This is a kind of book where you do not have to read from first page to last page—take a look at the table of contents and jump to the chapters that interest you.

## WHERE CAN I FIND MORE INFORMATION?

You can get more details about the book on the Elsevier Store at http://www.store.elsevier.com./product.jsp?isbn=9780128013977. You can find more information, supplementary material, and resources at http://www.designsmells.com. For any suggestions and feedback, please contact us at designsmells@gmail.com.

## WHY DID WE WRITE THIS BOOK?

Software design is an inherently complex activity and requires software engineers to have a thorough knowledge of design principles backed with years of experience and loads of skill. It requires careful thought and analysis and a deep understanding of the requirements. However, today, software engineers are expected to build really complex software within a short time frame in an environment where requirements are continuously changing. Needless to say, it is a huge challenge to maintain the quality of the design and overall software in such a context.

We, therefore, set out on a task to help software engineers improve the quality of their design and software. Our initial efforts were geared toward creating a method for assessing the design quality of industrial software (published as a paper [4]). We surveyed a number of software engineers and tried to understand the challenges they faced during design in their projects. We realized that while many of them possessed a decent theoretical overview of the key design principles, they lacked the knowledge of how to apply those principles in practice. This led to smells in their design.

Our survey also revealed a key insight—design smells could be leveraged to understand the mistakes that software engineers make while applying design principles. So, we embarked on a long journey to study and understand different kinds of smells that manifest in a piece of design. During this journey, we came across smells scattered across numerous sources including books, papers, theses, and tools and started documenting them. At the end of our journey, we had a huge collection of 530 smells!

We had hoped that our study would help us understand design smells better. It did, but now we were faced with the humongous task of making sense of this huge collection of smells. We, therefore, decided to focus only on structural design smells. We also decided to limit our focus to smells that are commonly found in real-world projects. Next, we set out on a mission to meaningfully classify this reduced collection of smells. We experimented with several classification schemes but were dissatisfied with all of them. In the midst of this struggle, it became clear to us that if we wanted to organize this collection so that we could share it in a beneficial manner with fellow architects, designers, and developers, our classification scheme should be linked to something fundamental, i.e., design principles. From this emerged the following insight:

> *When we view every smell as a violation of one or more underlying design principle(s), we get a deeper understanding of that smell; but perhaps more importantly, it also naturally directs us toward a potential refactoring approach for that smell.*

We built upon this illuminating insight and adopted Booch's fundamental design principles (abstraction, encapsulation, modularization, and hierarchy) as the basis for our classification framework. We categorized and aggregated the smells based on which of the four design principles they primarily violated, and created an initial

catalog of 27 design smells. We published this initial work as a paper [5] and were encouraged when some of the reviewers of our paper suggested that it would be a good idea to expand this work into a book. We also started delivering corporate training on the topic of design smells and observed that software practitioners found our smell catalog really useful. Buoyed by the positive feedback from both the reviewers of the paper and software practitioners, we decided to develop our initial work into this book.

Our objective, through this book, is to provide a framework for understanding how smells occur as a violation of design principles and how they can be refactored to manage technical debt. In our experience, the key to transforming into a better designer is to *use smells as the basis to understand how to apply design principles effectively in practice*. We hope that this core message reaches you through this book.

We have thoroughly enjoyed researching design smells and writing this book and hope you enjoy reading it!

# Acknowledgments

There are a number of people who have been instrumental in making this book see the light of day. We take this opportunity to sincerely acknowledge their help and support.

First, we want to thank those who inspired us to write this book. These include pioneers in the area of software design and refactoring, including Grady Booch, David Parnas, Martin Fowler, Bob Martin, Stéphane Ducasse, Erich Gamma, Ward Cunningham, and Kent Beck. Reading their books, articles, and blogs helped us better understand the concepts of software design.

Next, we want to thank all those who helped us write this book. In this context, we want to deeply appreciate the people who shared their experiences and war stories that made their way into the book. These include the attendees of our training sessions, participants of the online Smells Forum that we had set up, and the numerous people that we talked to at various conferences and events.

We would also like to thank the technical reviewers of this book: Grady Booch, Stéphane Ducasse, and Michael Feathers. Their critical analyses and careful reviews played a key role in improving the quality of this book. We are also thankful to Venkat Subramaniam for providing valuable review comments and suggestions. In addition, we would like to thank our friends Vishal Biyani, Nandini Rajagopalan, and Sricharan Pamudurthi for sparing their time and providing thoughtful feedback.

We are deeply indebted to Grady Booch and Stéphane Duccase for believing in our work and writing the forewords for our book.

This book would not have been possible without the overwhelming help and support we received from the whole team at Morgan Kaufmann/Elsevier. A special thanks to Todd Green who believed in our idea and provided excellent support from the time of inception of the project to its realization as a book that you hold in your hands. We also would like to convey our sincere thanks to Lindsay Lawrence for supporting us throughout the book publishing process. Further, we thank Punithavathy Govindaradjane and her team for their help during the production process Finally, our thanks to Mark Rogers for his invaluable contribution as the designer.

Finally, we want to express our gratitude toward those who supported us during the writing process. Our special thanks to K Ravikanth and PVR Murthy for their insightful discussions which helped improve the book. Girish and Tushar would like to thank Leny Thangiah, Rohit Karanth, Mukul Saxena, and Ramesh Vishwanathan (from Siemens Research and Technology Center, India) and Raghu Nambiar and Gerd Hoefner (from Siemens Technologies and Services Pvt. Ltd., India) for their constant support and encouragement. Ganesh would like to thank Ajith Narayan and Hari Krishnan (from ZineMind Technologies Pvt. Ltd., India) for their support.

We spent countless weekends and holidays working on the book instead of playing with our kids or going out for shopping. We thank our families for their constant love, patience, and support despite the myriad long conferencing sessions that we had during the course of this book project.

This page intentionally left blank

CHAPTER

# 1 Technical Debt

The first and most fundamental question to ask before commencing on this journey of refactoring for design smells is: What are design smells and why is it important to refactor the design to remove the smells?

Fred Brooks, in his book *The Mythical Man Month*, [6] describes how the inherent properties of software (i.e., complexity, conformity, changeability, and invisibility) make its design an "essential" difficulty. Good design practices are fundamental requisites to address this difficulty. One such practice is that a software designer should be aware of and address *design smells* that can manifest as a result of design decisions. This is the topic we cover in this book.

So, what are design smells?

> *Design smells are certain structures in the design that indicate violation of fundamental design principles and negatively impact design quality.*

In other words, a design smell indicates a potential problem in the design structure. The medical domain provides a good analogy for our work on smells. The symptoms of a patient can be likened to a "smell," and the underlying disease can be likened to the concrete "design problem."

This analogy can be extended to the process of diagnosis as well. For instance, a physician analyzes the symptoms, determines the disease at the root of the symptoms, and then suggests a treatment. Similarly, a designer has to analyze the smells found in a design, determine the problem(s) underlying the smells, and then identify the required refactoring to address the problem(s).

Having introduced design smells, let us ask why it is important to refactor[1] the design to remove the smells.

The answer to this question lies in *technical debt*—a term that has been receiving considerable attention from the software development community for the past few years. It is important to acquire an overview of technical debt so that software developers can understand the far-reaching implications of the design decisions that they make on a daily basis in their projects. Therefore, we devote the discussion in the rest of this chapter to technical debt.

[1] In this book, we use the term *refactoring* to mean "behavior preserving program transformations" [13].

**Refactoring for Software Design Smells.** http://dx.doi.org/10.1016/B978-0-12-801397-7.00001-1

## 1.1 WHAT IS TECHNICAL DEBT?

*Technical debt is the debt that accrues when you knowingly or unknowingly make wrong or non-optimal design decisions.*

Technical debt is a metaphor coined by Ward Cunningham in a 1992 report [44]. Technical debt is analogous to financial debt. When a person takes a loan (or uses his credit card), he incurs debt. If he regularly pays the installments (or the credit card bill) then the created debt is repaid and does not create further problems. However, if the person does not pay his installment (or bill), a penalty in the form of interest is applicable and it mounts every time he misses the payment. In case the person is not able to pay the installments (or bill) for a long time, the accrued interest can make the total debt so ominously large that the person may have to declare bankruptcy.

Along the same lines, when a software developer opts for a quick fix rather than a proper well-designed solution, he introduces technical debt. It is okay if the developer pays back the debt on time. However, if the developer chooses not to pay or forgets about the debt created, the accrued interest on the technical debt piles up, just like financial debt, increasing the overall technical debt. The debt keeps increasing over time with each change to the software; thus, the later the developer pays off the debt, the more expensive it is to pay off. If the debt is not paid at all, then eventually the pile-up becomes so huge that it becomes immensely difficult to change the software. In extreme cases, the accumulated technical debt is so huge that it cannot be paid off anymore and the product has to be abandoned. Such a situation is called *technical bankruptcy*.

## 1.2 WHAT CONSTITUTES TECHNICAL DEBT?

There are multiple sources of technical debt (Figure 1.1). Some of the well-known dimensions of technical debt include (with examples):

- **Code debt:** Static analysis tool violations and inconsistent coding style.
- **Design debt:** Design smells and violations of design rules.
- **Test debt:** Lack of tests, inadequate test coverage, and improper test design.
- **Documentation debt:** No documentation for important concerns, poor documentation, and outdated documentation.

In this book, we are primarily concerned with the design aspects of technical debt, i.e., design debt. In other words, when we refer to technical debt in this book, we imply design debt.

To better understand design debt, let us take the case of a medium-sized organization that develops software products. To be able to compete with other organizations

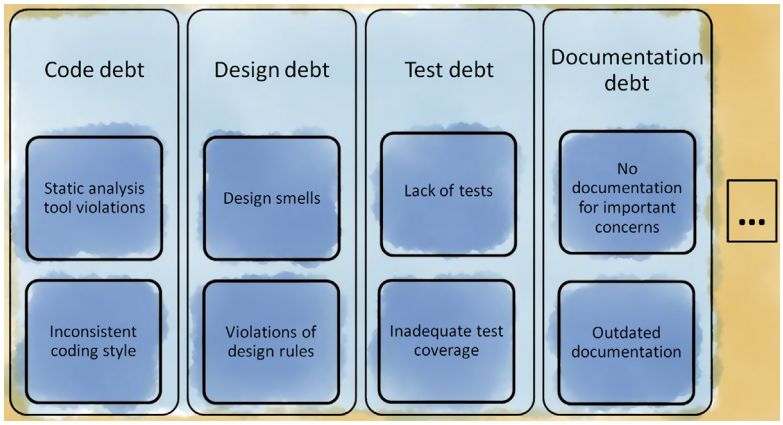



**FIGURE 1.1**

Dimensions of technical debt.

in the market, this organization obviously wants to get newer products on the market faster and at reduced costs. But how does this impact its software development process? As one can imagine, its software developers are expected to implement features faster. In such a case, the developers may not have the opportunity or time to properly assess the impact of their design decisions. As a result, over time, such a collection of individual localized design decisions starts to degrade the structural quality of the software products, thereby contributing to the accumulation of design debt.

If such a product were to be developed just once and then no longer maintained, the structural quality would not matter. However, most products are in the market for a long time period and therefore have an extended development and maintenance life cycle. In such a case, the poor structural quality of the software will significantly increase the effort and time required to understand and maintain the software. This will eventually hurt the organization's interests. Thus, it is extremely important for organizations to monitor and address the structural quality of the software. The work that needs to be invested in the future to address the current structural quality issues in the software is design debt.

An interesting question in the context of what constitutes technical debt is whether defects/bugs are a part of this debt. Some argue that defects (at least some of them) originate due to technical debt, thus are part of technical debt. There are others who support this viewpoint and argue that if managers decide to release a software version despite it having many known yet-to-be-fixed defects, these defects are a part of technical debt that has been incurred.

However, there are others in the community who argue that defects do not constitute technical debt. They argue that the main difference between defects and technical debt is that defects are visible to the users while technical debt is largely invisible. We support this stance. In our experience, defects are rarely ignored by the organization and receive much attention from the development

teams. On the other hand, issues leading to technical debt are mostly invisible and tend to receive little or no attention from the development teams. Why does this happen?

This happens because defects directly impact external quality attributes of the software that are directly visible to the end users. Technical debt, on the other hand, impacts internal quality of the software system, and is not directly perceivable by the end users of the software. Organizations value their end users and cannot afford to lose them; thus, defects get the utmost attention while issues related to "invisible" technical debt are usually deferred or ignored. Thus, from a practical viewpoint, it is better to leave defects out of the umbrella of technical debt, so that they can be dealt with separately; otherwise, one would fix defects and mistakenly think that technical debt has been addressed.

## 1.3 WHAT IS THE IMPACT OF TECHNICAL DEBT?

Why is it important for a software practitioner to be aware of technical debt and keep it under control? To understand this, let us first understand the components of technical debt. Technical debt is a result of the principal (the original hack or shortcut), and the accumulated interest incurred when the principal is not fixed. The interest component is compounding in nature; the more you ignore it or postpone it, the bigger the debt becomes over time. Thus, it is the interest component that makes technical debt a significant problem.

Why is the interest compounding in nature for technical debt? One major reason is that often new changes introduced in the software become interwoven with the debt-ridden design structure, further increasing the debt. Further, when the original debt remains unpaid, it encourages or even forces developers to use "hacks" while making changes, which further compounds the debt.

Jim Highsmith [45] describes how Cost of Change (CoC) varies with technical debt. A well-maintained software system's actual CoC is near to the optimal CoC; however, with the increase in technical debt, the actual CoC also increases. As previously mentioned, in extreme cases, the CoC can become prohibitively high leading to "technical bankruptcy."

Apart from technical challenges, technical debt also impacts the morale and motivation of the development team. As technical debt mounts, it becomes difficult to introduce changes and the team involved with development starts to feel frustrated and annoyed. Their frustration is further compounded because the alternative—i.e., repaying the whole technical debt—is not a trivial task that can be accomplished overnight.

It is purported that technical debt is the reason behind software faults in a number of applications across domains, including financing. In fact, a BBC report clearly mentions technical debt as the main reason behind the computer-controlled trading error at U.S. market-maker Knight Capital that decimated its balance sheet [46].

## CASE STUDY

To understand the impact that technical debt has on an organization, we present the case of a medium-sized organization and its flagship product. This product has been on the market for about 12 years and has a niche set of loyal customers who have been using the same product for a number of years.

Due to market pressures, the organization that owns the product decides to develop an upgraded version of the product. To be on par with its competitors, the organization wants to launch this product at the earliest. The organization marks this project as extremely important to the organization's growth, and a team of experienced software architects from outside the organization are called in to help in the design of the upgraded software.

As the team of architects starts to study the existing architecture and design to understand the product, they realize pretty soon that there are major problems plaguing the software. First and foremost, there is a huge technical debt that the software has incurred during its long maintenance phase. Specifically, the software has a monolithic design. Although the software consists of two logical components—client and server—there is just a single code base that is being modified (using appropriate parameters) to either work as a client or as a server. This lack of separation of client and server concerns makes it extremely difficult for the architects to understand the working of the software. When understanding is so difficult, it is not hard to imagine the uncertainty and risk involved in trying to change this software.

The architects realize that the technical debt needs to be repaid before extending the software to support new features. So they execute a number of code analyzers, generate relevant metrics, formulate a plan to refactor the existing software based on those metrics, and present this to the management. However, there is resistance against the idea of refactoring. The managers are very concerned about the impact of change. They are worried that the refactoring will not only break the existing code but also introduce delays in the eventual release of the software. They refer the matter to the development team. The development team seems to be aware of the difficulty in extending the existing code base. However, surprisingly, they seem to accept the quality problems as something natural to a long-lived project. When the architects probe this issue further, they realize that the development team is in fact unaware of the concept of technical debt and the impact that it can have on the software. In fact, some developers are even questioning what refactoring will bring to the project. If the developers had been aware of technical debt and its impact, they could have taken measures to monitor and address the technical debt at regular intervals.

The managers have another concern with the suggestion for refactoring. It turns out that more than 60% of the original developers have left the project, and new people are being hired to replace them. Hence, the managers are understandably extremely reluctant to let the new hires touch the 12-year old code base. Since the existing code base has been successfully running for the last decade, the managers are fearful of allowing the existing code to be restructured.

So, the architects begin to communicate to the development team the adverse impacts of technical debt. Soon, many team members become aware of the cause behind the problems plaguing the software and become convinced that there is a vital need for refactoring before the software can be extended. Slowly, the team starts dividing into pro-refactoring and anti-refactoring groups. The anti-refactoring group is not against refactoring per se, but does not want the focus of the current release to be on refactoring. The pro-refactoring group argues that further development would be difficult and error-prone unless some amount of refactoring and restructuring is first carried out.

Eventually, it is decided to stagger the refactoring effort across releases. So, for the current release, it is decided to refactor only one critical portion of the system. Developers are also encouraged to restructure bits of code that they touch during new feature development. On paper, it seems like a good strategy and appears likely to succeed.

However, the extent of the incurred technical debt has been highly underestimated. The design is very tightly coupled. Interfaces for components have not been defined. Multiple responsibilities

*Continued*