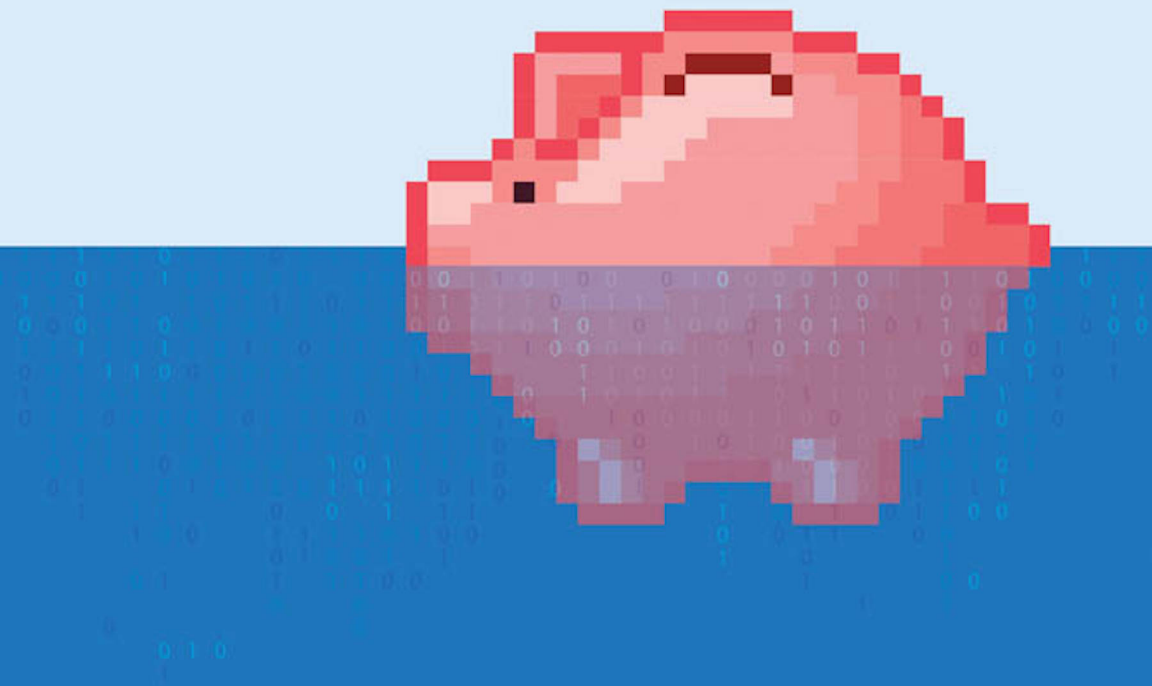
Technical Debt in Practice

How to Find It and Fix It



Neil Ernst, Rick Kazman, and Julien Delange

Technical Debt in Practice

Technical Debt in Practice

How to Find It and Fix It

Neil Ernst, Rick Kazman, and Julien Delange

The MIT Press
Cambridge, Massachusetts
London, England

The MIT Press would like to thank the anonymous peer reviewers who provided comments on drafts of this book. The generous work of academic experts is essential for establishing the authority and quality of our publications. We acknowledge with gratitude the contributions of these otherwise uncredited readers.

This book was set in Stone Serif by Westchester Publishing Services.

Library of Congress Cataloging-in-Publication Data

Names: Ernst, Neil, author. | Kazman, Rick, author. | Delange, Julien, author.
Title: Technical debt in practice : how to find it and fix it / Neil Ernst, Rick Kazman, and Julien Delange.
Description: Cambridge, Massachusetts : The MIT Press, 2021. | Includes bibliographical references and index.
Identifiers: LCCN 2020041281 | ISBN 9780262542111 (paperback)
Subjects: LCSH: Computer software—Development—Quality control. | Software failures—Prevention. | Project management. | Maintainability (Engineering)
Classification: LCC QA76.76.Q35 E76 2021 | DDC 005.3028/7—dc23
LC record available at https://lccn.loc.gov/2020041281

—For Kieran, Elliott, and Kambria, who toured our old hometown while we labored on this. N.E.

—For Hong-Mei, who humored our rants and cheered us on. R.K.

—For Alejandra and Chewie. J.D.

Contents

Acknowledgments

This book would not have happened without the help and inspiration of a lot of different people. The three of us met while working at the Carnegie Mellon University Software Engineering Institute in Pittsburgh. Thanks to all our coworkers at the SEI, including Robert Nord, Peter Feiler, Ipek Ozkaya, James Ivers, Felix Bachmann, John Klein, Stephanie Bellomo, Phil Bianco, and Chuck Weinstock. Linda Northrop remains an inspiration.

Rick Kazman would especially like to thank his research collaborators, including Yuanfang Cai, Damian Tamburri, Humberto Cervantes, and the group at SoftServe, including Serge Haziyev and Andriy Shapochka.

Julien would like to thank Julien Danjou and Nicolas Devillard for their time.

Neil would like to thank his collaborators, as well as the anonymous reviewers of this manuscript, who although causing more late nights, doubtless improved the final version.

1 Introduction

There is scarcely anything that drags a person down like debt.
—P. T. Barnum

1.1 What Is Technical Debt?

If you have a credit card or a mortgage or a car loan, you already know a bit about debt. You are also probably familiar with the many metaphors that surround the ways we think about and talk about debt: crushing debt, drowning in debt, buried in debt. We talk of debt in relationships, or the social debt that you feel after your neighbor invited you to dinner and you have not reciprocated. Metaphors are pervasive in our lives. So it should come as no surprise that we also apply this metaphor to the technical world. The fact that you cracked the cover of this book suggests that this metaphor is already speaking to you. The metaphor of technical debt is something that every software developer has at least heard of. But metaphors only take you so far. Our purpose in writing this book is to move us all beyond just the metaphor into actionable insights, methods, and tools that allow us to deal with technical debt as software engineers.

Every organization that creates nontrivial software systems has technical debt, and the software development world is becoming increasingly conscious of this debt. Take Facebook, for example: the company originally used the PHP language to prototype and deliver their product quickly as a young company, but then as they grew they had to face the limitations of their early technical decisions. PHP simply was not able to scale and provide the kind of performance that they needed to support their ever-growing userbase, and so Facebook had to find solutions to pay this debt back; in

their case, the solution was to create a PHP transpiler. Almost all companies face such issues, and this book provides many concrete examples of technical debt—from Boeing and Airbus to Twitter and many others. But we do not just provide horror stories (although there are plenty). We provide a concrete set of practical solutions—methods, tools, and techniques—for dealing with technical debt. An important message of this book is not that you should never incur debt; it is that you should not *inadvertently* assume a large debt, at least not without knowing when and how you will pay it back. And we provide you the techniques to do just that: to identify debt, to manage the debt that you have identified, and to avoid debt where possible.

The metaphor likening the creation of software to a process of going into debt was invented by Ward Cunningham in 1992. Cunningham was inspired by the seminal work of the linguist and philosopher George Lakoff on the power of metaphor. Lakoff argued that metaphors are central to the development of ideas and that humans are almost unable to talk and reason without using metaphors.

Cunningham, inspired by this insight, created the debt metaphor as a way of explaining his software development actions and decisions to his boss. (Actually, Cunningham did not call it "technical debt"; others coined that term later. Cunningham just called it "debt" in his early writings.) His boss came from the finance side, and Cunningham knew that his best chance of getting his boss to understand *what* he was doing and *why* was by using a metaphor to link with something that he felt his boss would be able to relate to: going into debt. Cunningham noted that, in the world of business and in our personal lives, we often borrow money to achieve a goal more quickly (like buying a car or a home). But then we have to live with the "penalty" of this decision: we pay interest on that loan.

Cunningham thought that going into debt could be a good thing for the eventual state of the software product. The world agrees with him. As a society we love our mortgages, car loans, and credit cards. In the world of software, we try to get a product to market as quickly as possible, even though this product may be flawed, even though we may not fully understand our market and hence the features that we should be providing, and even though, in our hurry to get the product out the door, we inevitably take shortcuts and make some ill-considered decisions.

Despite all these negatives, Cunningham still maintained that "going into debt" was a *good* thing. For Cunningham, it was an essential part of

building software iteratively, naturally occurring as your theory of the software diverged from what you had actually written. Paying down the debt meant realigning the implementation with the theory. Just as in real life, these loans need to be repaid. Debt is simply a means to an end. In real life, if you do not repay what you borrow, more and more of your income goes to servicing your debt. Your credit score drops, collection agencies appear at your door, and your ability to borrow further is greatly decreased. In the world of software, if we do not repay the debt by refactoring—repairing and improving the hasty and ill-considered code that we first delivered—then our debt may, and often does, overwhelm us. We will then spend all of our effort fixing bugs and never get to add new features, and those new features are what our customers really value, after all. (If you are an experienced developer you are probably saying "been there, done that" at this point.)

In software engineering, the perfect is usually the enemy of the good. If we wait until we have a perfect understanding of our requirements and a beautiful, polished design, we will have likely missed our window of opportunity to capture market share. We may have alienated our users and lost the support of management. A better way is to develop something, present it to users, get feedback, and iterate, gradually and incrementally improving the product as we learn more about the problem domain. So debt is not, by itself, the problem. The problem is not identifying and acknowledging the debt, not measuring the debt, and not paying the debt back. We must, in fact, move beyond the metaphor. We must not only acknowledge our debt, but measure it and manage it. But typically we do neither; we simply ignore it. Why does this happen?

This happens because, in the early stages of a product, the interest payments are not too large. The code base is growing, but often it is still comprehensible by its creators. We are adding features without too much trouble. We are happy and our customers are happy. We are getting positive feedback, which reinforces our behavior and our motivation: get the next feature out the door.

Over time, of course, complexity increases. We might keep up for a while: some people can keep a few thousand lines of code in their head—they can digest and assimilate all of this information and operate on this knowledge very efficiently. Some exceptional programmers might be able to keep 20,000 or 50,000 or even 100,000 lines of code in their head. But ask yourself: What is the largest system allowed by law? That is a stupid

question of course. No laws constrain artificial systems; they have no inherent bounds. That is why the Chromium project has nearly 20 million lines of code and the Linux kernel is almost as big. Clearly no human can digest 20 million lines of code. If you could read one line of code per second and did this nonstop, twenty-four hours a day, it would take you nearly eight months to read the entirety of Chromium.

It is a fact: system complexity, if left unmanaged, will always overwhelm us. Thus, a system that starts off as comprehensible becomes incomprehensible over time, despite our best efforts. We can fool ourselves in the early stages of development that everything is OK, and that we can ignore the growing complexity—the mounting debt—because we *can* keep it all in our heads, but in the end it overwhelms us. That is, it overwhelms us unless we pay down some of this debt, unless we consciously attempt to manage the complexity, unless we forego some of the instant gratification that worked so well for us in the early stages of the project (delivering features) to do the less visible but critically important task of re-envisioning, restructuring, and refactoring our technical artifacts (e.g., code, tests, documentation). Part of this book is a series of case studies that explain exactly this process of moving through early to late phases of development.

Slowly, as the field of technical debt matured, people started to realize that there were many forms of debt and many nuances regarding how, when, why, and if one should take on debt. Martin Fowler created a nice taxonomy of technical debt back in 2009, which we have adapted slightly, as shown below in figure 1.1.

In this figure, Fowler distinguishes two dimensions of debt. Dimension 1 (which we call "risk") explores this question: Is the debt reckless or prudent? If one simply codes without ever considering design, that would be reckless. That would be equivalent to constructing a building without ever having a civil engineer or architect design or analyze it. This is unthinkable (and illegal) in the construction industry. On the other hand, debt may be strategic and prudent. For example, if we are facing a hard deadline, if we need to prototype and produce *something*, so that we can get feedback and figure out what we really need to build, these are cases where taking on debt is prudent. Or, to turn the argument around, spending a large amount of effort on design in these instances would be a waste, since we have more important goals (such as the deadline) or little confidence that we would be building the right product (which is why we prototype). There is no point in building the wrong

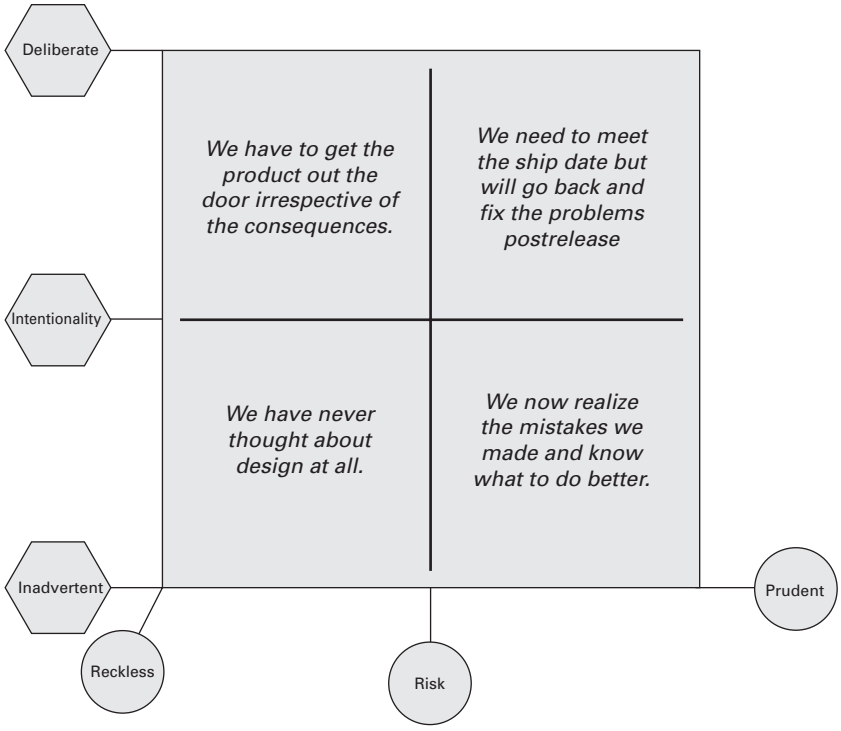


Figure 1.1
Technical debt quadrants.

product right, or in delivering a perfect product after the deadline has been missed. So taking on debt in such cases is perfectly reasonable if (and this is the hard part) we commit to analyzing and repaying the debt in the future.

Fowler's dimension 2—which we call "intentionality"—explores a second question: Is the debt deliberate or inadvertent? There are times when a debt is both deliberate and reckless, as scary as that sounds. This is often the result of management decisions: agreeing to unrealistic deadlines, understaffing, not hiring the right staff, or not training staff adequately. In such cases, a product is often thrown together while developers don't realize they are even taking on debt; perhaps this notion has never even occurred to them. They are simply in survival mode, trying to meet their deadlines and goals. However, there are also cases where the debt is deliberate. In this case, the team knows that they are taking shortcuts and they do so as a calculated investment. The argument goes like this: We need to make some

suboptimal decisions now, to meet our near-term needs, but we fully intend to pay back the costs of these decisions. Debt that is prudent and deliberate is good debt; this is why we obtain a mortgage on a home, for example. Debt that is reckless and deliberate is often a symptom of a project that is in trouble. The analogy in financial terms would be excessive credit card debt, payday loans, or borrowing money from a loan shark. You really cannot afford it, but you do it anyway. Debt that is reckless and inadvertent is often the result of inadequate knowledge and training. While this likely does not apply to the readers of *this* book, the vast majority of people in the world have little knowledge of even the most rudimentary financial concepts.

Steve McConnell, another early pioneer in the field of technical debt, further refined and categorized the concepts introduced by Cunningham. He wrote about short-term vs. long-term debt. As we just discussed, sometimes we take on debt deliberately, for example to preserve startup capital or to reduce time to market. Such debt should, however, be short-term with the intention to pay it off once we have better funding or once we have our first product release in the marketplace. On the other hand, sometimes we take on long-term debt for strategic reasons. For example, if we know that a system is nearing the end of its lifetime, but we can still squeeze a few more years and a few more dollars out of it, then it likely makes sense to just make the minimal changes needed to keep it going. But note that in both cases, the decisions about taking on (and repaying) debt are intentional. Unintentional debt typically reflects poor knowledge, poor discipline, or bad programming practices, as we will discuss throughout the book. Sadly, most people are not adept at managing debt, and programmers are people (see the Financial Literacy Test sidebar in box 1.1 if you believe otherwise).

Box 1.1
Financial Literacy Test

About a decade ago Olivia Mitchell, a professor at the George Washington University, devised three simple questions to assess basic financial literacy. These were:

1. "Suppose you had $100 in a savings account and the interest rate was 2 percent per year. After five years, how much do you think you would have in the account if you left the money to grow?"

   a. More than $102
   b. Exactly $102

c. Less than $102
d. Don't know
e. Refuse to answer

2. "Imagine that the interest rate on your savings account was 1% per year and inflation was 2% per year. After one year, with the money in this account, would you be able to buy . . . "

a. More than today
b. Exactly the same as today
c. Less than today
d. Don't know
e. Refuse to answer

3. "Do you think the following statement is true or false?

Buying a single company stock usually provides a safer return than a stock mutual fund."

a. True
b. False
c. Don't know
d. Refuse to answer

Shockingly, just one third of Americans age fifty and older were able to correctly answer all three of these questions. And these results held up in further testing done in many other countries. While the answers improved slightly as the level of the respondents' education went up, even these results were still depressing. Only 44.3% of respondents with a college degree answered all three questions correctly.

What is the point of this story? It is quite simple. If we humans are so bad at thinking about financial debt, we are even more disadvantaged when thinking about technical debt. If even the simplest, most basic knowledge of financial literacy is uncommon, how likely is it that the average programmer, architect, project manager, or nontechnical manager would understand concepts of debt as they apply to the more abstract world of code and design, where the consequences of debt are often not felt for years? Thus, the point of this aside is to remind you that understanding debt and its consequences is likely not going to be intuitively obvious to most members of your organization. You are going to have to do some work to build a business case for monitoring and, in some cases, paying back, your accumulated technical debt.

Oh, and by the way, the answers to the financial literacy quiz are A, C, and B. If you got any of these wrong, you need to fix your financial literacy!

1.2 Moving Beyond the Metaphor

While technical debt began as a metaphor, it is slowly morphing into something that can be managed. If we want to believe that, as software engineers, we are truly engineers and not merely hackers, we need to measure, track, and reason about our software. In her landmark article "Prospects for an Engineering Discipline of Software," Mary Shaw outlined what it means to be an engineering discipline. She defined engineering as: "Creating cost-effective solutions to practical problems by applying scientific knowledge building things in the service of mankind." Engineering is based on a scientific foundation. To properly manage software engineering processes and artifacts, they must be measured. As any management scientist will tell you, if you can't measure it, you can't manage it. And yet the vast majority of software built today is not designed, not measured, and not managed. Because the demand for software is ever-growing and because software is so malleable, we have (mostly) been able to get away with this cavalier attitude. But make no mistake: this is not engineering. Software development, the way it is practiced in most projects is, at best, a craft. And a craft depends on the skill of the engineer.

As we will show in this book, there is hope: we can move beyond the metaphor of technical debt. We can move beyond software engineering as a craft. But doing so requires a change of habits, it requires a change of tooling, and it requires discipline.

The technical debt metaphor is growing to encompass more aspects of a modern software project, beyond source code. For example, there are now tools that can measure and analyze social debt and design debt. Let's consider social debt. There are numerous examples of projects that have been driven into the ground not because their technology was inferior but because their organizational structure was pathological. Just as researchers have identified code smells—duplicate code, god classes, inappropriate intimacy, and so on—a number of organizational smells have been identified. For example, Tamburri and colleagues have identified organizational smells—social debt—such as:

- *Cognitive distance*: the distance developers perceive among peers with considerable (educational, experiential, cultural) background differences.
- *Informality excess*: excessive informality due to the lack of information management and control protocols.

- *Disengagement*: thinking the product is mature and deploying it even though it might not be ready.
- *Institutional isomorphism*: excessive similarity of the processes and structures of one subgroup to those of others.

Companies are beginning to invest in monitoring and managing technical debt. For example, in one company we analyzed eight projects and applied three complementary analysis techniques to measure and monitor their technical debt. In chapter 4, we describe an analysis of the code bases, issue-tracking systems, and version control systems of these eight projects to assess the quality of their software architectures. The outputs of the analyses were (1) architecture-wide coupling measures (decoupling level [DL], and propagation cost [PC]) that the company could use to compare their systems to industry benchmarks, (2) a set of design flaws found in each project, and (3) a set of design "hotspots" in each project—groups of files that architects and developers consider to be worth refactoring.

By measuring and tracking these eight projects, and by quantifying the costs incurred by the accumulated debt, we were able to convince management to refactor six of them. Prior to our collecting this data, the project members were vaguely aware of their debt—they of course experienced it every day—but they had no way to measure it and hence no way to convince their managers to invest the time and effort to pay down the debt. A connected metaphor comes (initially) from the world of automotive engineering. *Lean thinking* suggests that one important practice in manufacturing and producing software is to reduce the amount of waste. Waste is any practice that is not directly adding value to the product. Technical debt is waste, either in the sense of extra work to deal with the effects of the debt or as required rework to remove the debt that we should never have incurred in the first place (and, ultimately, have financial and productivity impacts).

Thus, this book is not a depressing tale but rather a reason for hope. The good news is that we should, and we *can*, explicitly monitor and manage technical debt.

1.3 Summary

Our message is that software development and technical debt can be controlled and managed. But this process of managing must begin with measurement.

This is the message of the remainder of this book: technical debt is real, and it has enormous consequences on project success. For this reason, a prudent project manager or technical lead needs to plan for how to avoid, detect, monitor, manage, and pay down debt.

In the rest of this book, we will show how you can have debt in your design, in your code, in your test suites, and in your documentation. And we will provide practical strategies in each case for avoiding this debt. But that is the ideal case. In many projects debt is already there, or it is unavoidable due to schedule pressures. To help resolve existing debts, we also provide strategies for identifying and quantifying this debt and for making the business case to pay down the debt.

1.4 Book Outline

The majority of the book is dedicated to the details of dealing with technical debt in the software development lifecycle. We include chapters on requirements debt, implementation debt, testing debt, architecture debt, documentation debt, deployment debt, and social debt—broadening the horizons of what has been typically thought of as technical debt. In each of these chapters we make the case for why each of these kinds of debt is worrisome. Figure 1.2 briefly highlights how these different lifecycle activities can cause technical debt. This is a big-picture view and as such is incomplete, but it is suggestive. We examine every major project activity—requirements, design and architecture, testing, deployment and delivery, and documentation—and their consequences for technical debt. We also introduce the little-studied but pernicious topic of social debt.

This characterization of technical debt may be different from what you have read in other books or articles. Most of the existing literature and most technical debt detection tools focus on *code* as the carrier of debt. While we agree wholeheartedly that code may contain considerable debt, you can "go into debt" in other dimensions of your software project just as easily. After all, debt is incurred whenever you took a shortcut, or whenever you did not understand or manage an aspect of your project lifecycle as well as you could have. For example, if you have a large team and subparts of the team are not communicating with each other, this might be a problem for your project. It is a debt because you might have avoided this by paying more attention to mentoring new project members or fostering better

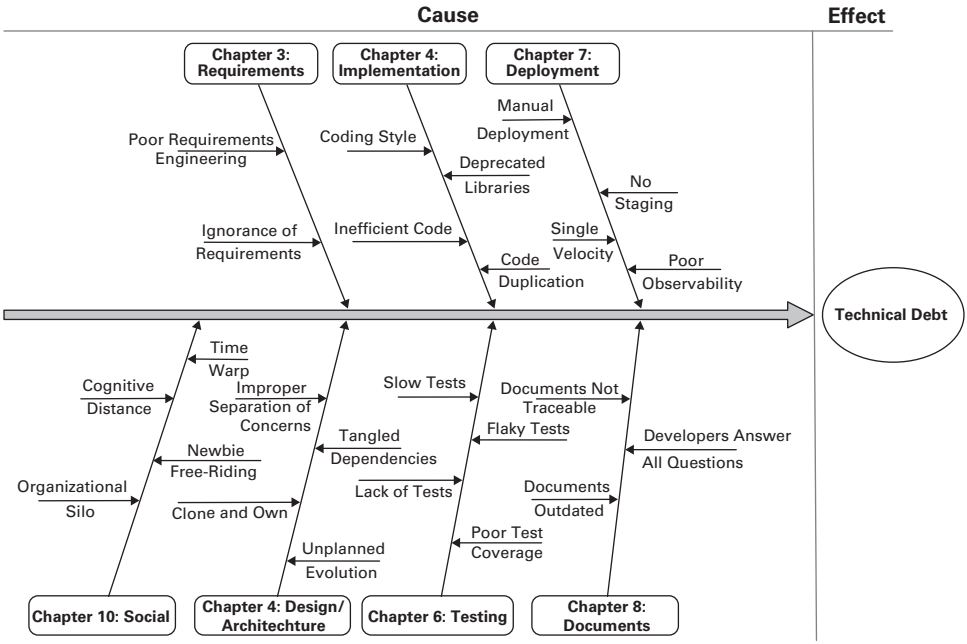


Figure 1.2
Fishbone diagram of the causes of technical debt, per software lifecycle phase. A fishbone (or Ishikawa) diagram represents a problem on the right (head) and causes of that problem on the left.

communication channels and practices. But if you, instead, just let everyone loose on the project and hoped for the best, you have incurred what we call social debt.

Let's consider another example: Perhaps your project started off as a simple codebase with a small team of developers who all knew each other well and worked closely together. As the project grew, however, a great many new developers were added, some of whom were working in remote locations. The old way of doing things did not require anyone to document anything; you could always just walk over to your buddy or chat over lunch and share information informally. However, as the project progressed and grew this was no longer feasible, and many people were spending a lot of time reinventing the wheel. In this case, you may have incurred documentation debt. If you had just spent some time documenting the rationale behind the system, coding standards, architectural assumptions, and

design constraints you might have avoided this debt. But in the early days of the project this simply was not needed and was not a priority.

The many types of technical debt and their associated causes are not independent of one another, of course. Requirements debt leads to a poor understanding of what the software should be doing. That makes it hard to know what the architecture ought to be. And it makes it unlikely that anyone would bother to document these shifting sands. If the architecture is poorly understood, then the implementation will be prone to problems—and we will certainly have trouble writing tests if we don't know what the original requirements were. Finally, if we are incurring debt in all these phases, it is highly likely we will also have social debt that either causes or is caused by all of the other problems.

Thus we see the potential for debt in every aspect of the software development lifecycle, and we have organized our book accordingly, with chapters on requirements debt, implementation debt, architecture debt, testing debt, documentation debt, and deployment debt. Within each of these chapters we have adopted a common structure. First, we discuss how, in each lifecycle phase, technical debt can be **identified:** how the causes in figure 1.2 can be discovered, using techniques and tools and metrics. We then explain how to **manage** these problems, once they have been identified, and how they can be mitigated. Finally, to **avoid** the problems in the future, we discuss strategies to improve the conduct of that phase of the lifecycle so that technical debt will not reoccur. At the end of each chapter, we conclude with a section on further reading, where the sources for our discussion are elaborated.

We also examine the notion of technical debt in machine learning systems (chapter 9). As these systems are relatively new—compared with traditional forms of software development—we felt that it was important to evaluate the kinds of debt found there and the ways in which these new forms of debt can be identified and managed.

In the chapters on the social and managerial aspects of technical debt (chapters 10 and 11), we delve into more details about how projects, and project managers and architects, can deal with the inevitability of technical and social debt. We talk about how to measure code-level and programmer-level activities as indicators of such debt types. Social debt is the notion that many technical debt issues are caused by organizational problems. The technical debt metaphor lends itself to financial reasoning, and we discuss a few project management techniques for measuring and managing debt.

Throughout the book we intersperse our discussion of the various kinds of debt with a set of interviews with practitioners—detailing their experiences with various kinds of debts—and case studies from real projects, which were (or are still) affected by technical debt. We involve these practitioners in boxes we call "Voice of the Practitioner," where we distill lessons on managing technical debt from a practical perspective. We present the full interviews at the end of the book in an appendix. From each interview we summarize the purpose of the interview and some key technical debt takeaways. We do the same thing with our case studies, explaining how the case study explains and reveals key issues with technical debt.

In every chapter we provide concrete advice on how technical debt can be identified, avoided, and managed to deliver a higher-quality product faster than before. We give real-world examples of how teams have benefitted from debt removal and how they have been able to address long-term issues, such as performance problems, functional regressions, release delays, or design decay.

Further Reading

For a fascinating discussion of the use and power of metaphors, see: George Lakoff, Mark Johnson, *Metaphors We Live By* (University of Chicago Press, 1980).

The term "technical debt" (or, originally, just "debt") was coined by Ward Cunningham, in "The WyCash Portfolio Management System," http://c2.com/doc/oopsla92.html, 1992. This was later refined and elaborated by many others, such as Martin Fowler, where he described his "TechnicalDebtQuadrant," https://martinfowler.com/bliki/TechnicalDebtQuadrant.html, 2009; and Steve McConnell, in his essay "Technical Debt," http://www.construx.com/10x_Software_Development/Technical_Debt/, 2007. George Fairbanks summarizes the iterative nature of debt in his IEEE Software article "Ur-Technical Debt," https://www.georgefairbanks.com/ieee-software-v32-n4-july-2020-ur-technical-debt.

Annamaria Lusardi and Olivia Mitchell have discussed their extensive research and studies on financial literacy in "The Economic Importance of Financial Literacy: Theory and Evidence," *Journal of Economic Literature* 52, no. 1 (March 2014): 5–44.

Mary Shaw has written and spoken extensively on the progress of software engineering as a true engineering discipline. This was first presented

in the technical report: "Prospects for an Engineering Discipline of Software," CMU/SEI-90-TR-20 (September 1990). The quote is found on page 2.

A discussion of social debt and how an architect may detect it and defend against it can be found in Damian Tamburri, Rick Kazman, Hamed Fahimi, "The Architect's Role in Community Shepherding," *IEEE Software* 33, no. 6 (November–December 2016): 70–79.

Lean thinking was introduced to North American audiences in Eliyahu Goldratt and Jeff Cox, *The Goal: A Process of Ongoing Improvement* (North River Press, 2014). Donald Reinertsen has pioneered product development and the concept of cost of delay in "Principles of Product Development Flow" (Celeritas, 2009). The notion of applying Lean to software was introduced in Mary Poppendieck and Tom Poppendieck, *Lean Software Development: An Agile Toolkit* (Addison-Wesley Professional, 2003). Expanding beyond software development, Eric Ries's book *Lean Startup* (Crown Publishing, 2001) has been influential in understanding how to experiment and pivot an entire company.

Finally, Cai and colleagues have substantial experience in automatically analyzing architectures for design problems that lead to technical debt. The foundation for this work can be found in Yuanfang Cai, Lu Xiao, Rick Kazman, and Ran Mo, Qiong Feng, "Design Rule Spaces: A New Model for Representing and Analyzing Software Architecture," *IEEE Transactions on Software Engineering*, January 2018. They outline a set of tools for automatically detecting design degradation and design flaws, and they provide empirical results showing the consequences of those flaws in Ran Mo, Yuanfang Cai, Rick Kazman, Lu Xiao, and Qiong Feng, "Decoupling Level: A New Metric for Architectural Maintenance Complexity," *Proceedings of the International Conference on Software Engineering (ICSE) 2016*, Austin, TX, May 2016; and in Ran Mo, Yuanfang Cai, Rick Kazman, Lu Xiao, "Hotspot Patterns: The Formal Definition and Automatic Detection of Architecture Smells," *Proceedings of the 12th Working IEEE/IFIP Conference on Software Architecture (WICSA 2015)*, Montreal, Canada, May 2015.