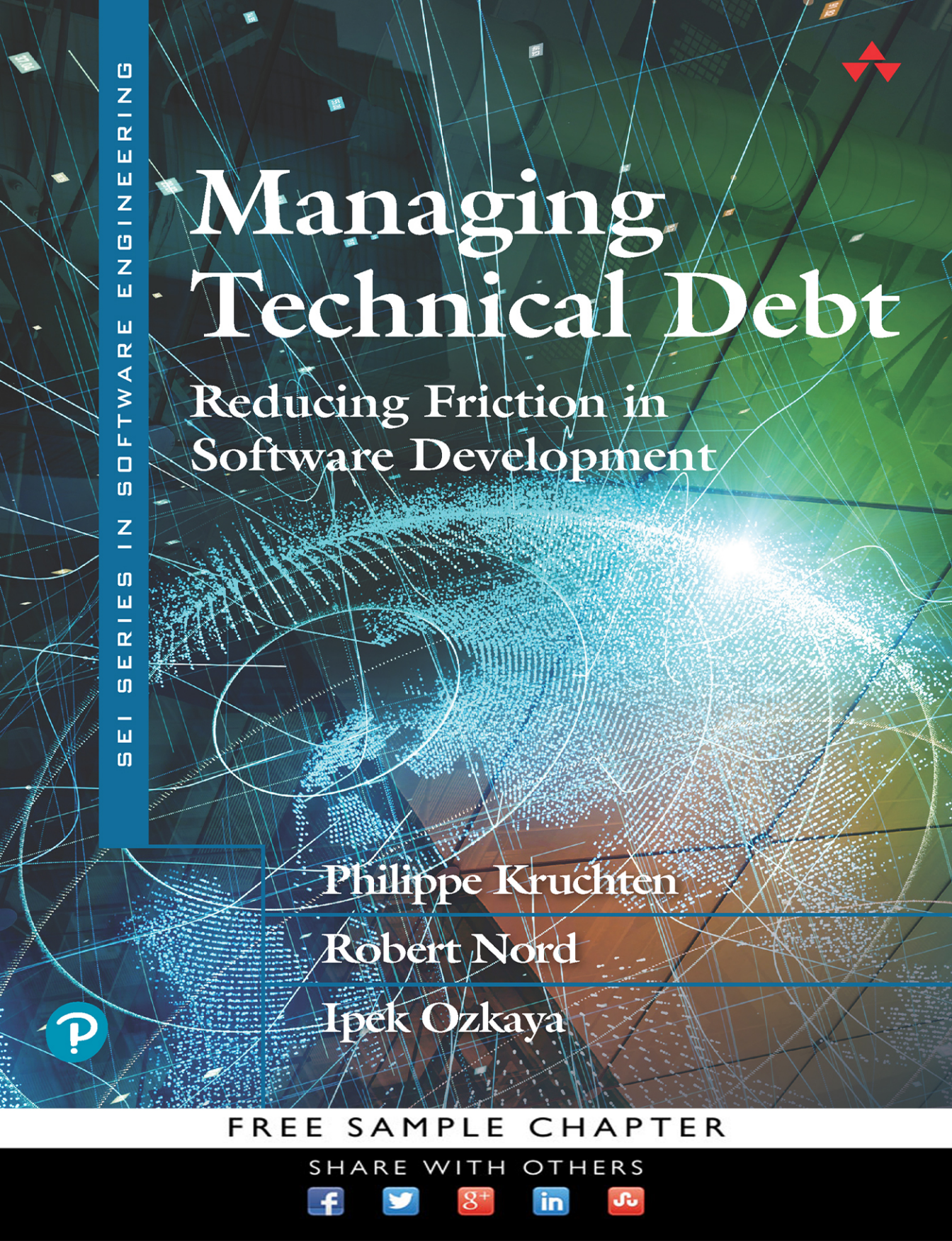

SEI SERIES IN SOFTWARE ENGINEERING
Managing Technical Debt
Reducing Friction in Software Development
Philippe Kruchten
Robert Nord
Ipek Ozkaya
FREE SAMPLE CHAPTER
SHARE WITH OTHERS

Managing Technical Debt

Managing Technical Debt

Reducing Friction in Software Development

Philippe Kruchten
Robert Nord
Ipek Ozkaya

Software Engineering Institute
Carnegie Mellon

Software Engineering Institute
Carnegie Mellon

The SEI Series in Software Engineering

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact intlcs@pearson.com.

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2019931698

ISBN-13: 978-0-13-564593-2
ISBN-10: 0-13-564593-X

1 19

Executive Editor
Kim Spenceley

Development Editor
Kiran Kumar Panigrahi

Managing Editor
Sandra Schroeder

Senior Project Editor
Lori Lyons

Copy Editor
Catherine D. Wilson

Indexer
Ken Johnson

Proofreader
Abigail Manheim

Cover Designer
Chuti Prasertsith

Compositor
codeMantra

To Sylvie, Nicolas, Alice, Zoé, Harmonie,
Claire, and Henri
—PK

To Victoria and Richard
—RN

To Ibrahim, Zeynep, and Zehra
—IO

Contents at a Glance

This page intentionally left blank

Contents

Foreword

In the late 1500s, a road was built encircling the island on which I now live. Well, not a road exactly, but more of a modest walking path, serving to connect the many small farming and fishing villages that flourished at that time. But, times change, and with the arrival of the whaling boats and the missionaries and the plantation owners in the 1800s, there was a clear economic incentive to reduce the friction of travel and to increase the capacity of transport. As such, using that original path as its architectural foundation, a wider road was built to accommodate horses and trains and the emerging motor car. Times changed yet again, and World War II necessitated yet wider and stronger roads, but—not surprisingly—corners were cut owing to the expediency of conflict. After the war, when the whalers, missionaries, plantation owners, and sailors were but an historical memory, that road remained, but now served to accommodate the cars of visitors who were arriving in alarmingly increasing numbers. Money for infrastructure being what it is, a new road was planned, but only partly built. The cost of maintaining the old parts of the road cut into the funds for building the new parts; but then, this is the nature of all systems. Even now, times change, and this time it is climate change, manifesting itself in the rise of the ocean and projected to reach three feet within the century. Already the ocean is encroaching on that ancient path and beginning to inundate the road in ways that make its replacement inevitable and urgent.

Software-intensive systems are a lot like that: Foundations are laid, corners are cut for any number of reasons that seem defensible at the time; but in the fullness of time, the relentless accretion of code over months, years, and even decades quickly turns every successful project into a legacy one. It is fascinating to watch young companies that grew quickly, unfettered by legacy code, suddenly wake up one day and realize that developing long-lived, quality software-intensive systems is hard.

What you have before you is an incredibly wise and useful book. Philippe, Ipek, Robert, and the other contributors have considerable real-world experience in delivering quality systems that matter, and their expertise shines through in these pages. Here you will learn what technical debt is, what is it not, how to manage it, and how to pay it down in responsible ways.

This is a book I wish I had when I was just beginning my career; but then, it couldn't have been written until now. The authors present a myriad of case studies, born from years of their experience, and offer a multitude of actionable insights for how to apply it to your project. Read this book carefully. Read it again. There's useful information on every page which, quite honestly, will change the way you approach technical debt in good and proper ways.

—Grady Booch
IBM Fellow
January 2019

Preface

Philippe: I ran into technical debt long before I had a name for it. In 1980, I was working at Alcatel on some peripheral device, and the code had to fit in 8 kilobytes (kB) of ROM (Read-Only Memory). With the deadline to "burn" the ROMs approaching, we did a lot of damage to the code to make it fit, thinking, "Oh, for the next release we'll have 16 kB available, we'll make it right…" We did get 16 kB of ROM for the next release, but we never, ever fixed all the abominable things we had to do to the source code because the deadline for the next product was, again, too close. New programmers coming on board would say, "Wow, this is ugly, brain-damaged, awful. How did you end up writing such bad code?" Colleagues would reply, "Oh, yes, go ask Philippe, he'll explain why it's like that. At least, on the bright side, it does the job and passes all the tests. So, fix that code at your own risk."

Robert: With the advent of agile practice, I was interested in hearing stories from developers about how it scales. Two projects in different organizations at the time were adopting agile and had recognized the importance of an end-to-end performance requirement. The demos for the minimal viable product were an unquestionable success. It just so happened that in each case, the demo sparked a new high-volume bandwidth requirement. One project was able to take the new requirement in stride while the other project "hit the wall," as Philippe would say. The architecture and supporting processes were not sufficiently flexible to allow the project to quickly adapt. This got me thinking about the choices that developers make to produce more features or to invest in architecture and infrastructure.

Ipek: I believe software engineering is first an economic activity. While in principle budget, schedule, and other business concerns should drive your design choices, that has not been my experience in many of the systems I worked on. A package routing system, let us call it the GIS-X, is a canonical example. I was part of the team that conducted an architectural evaluation of the system in 2007. The development team was tasked to incorporate advanced geographic information processing to GIS-X to optimize driving routes. As the schedule realities started to take priority, each of the five development teams working on the project started diverging from the design. Among several other technical issues, one key mistake the organization made was not assigning an architecture owner to keep the design, business, and resource constraints in check.

Around 2005–2008 the concept of technical debt started to emerge, in the form of myriads of blog entries, mostly in the agile process community. We realized that developers understood technical debt very well, even when they were not calling it that, but the business side of their organizations had little insight and saw it as very similar to defects. The three of us met several times around that time, and we initially worked on developing a little game about hard choices to help software teams get a better feeling for what technical debt is about. As we found more people both in industry and academia willing to understand more about this strange concept that did not fit very well in any software engineering narrative, we started in 2010 organizing a series of workshops on Managing Technical Debt, initially sponsored by the Software Engineering Institute (SEI), to explore more thoroughly the concept. We've had one workshop a year since. They have grown in importance and are now a series of annual TechDebt conferences.

The three of us wrote papers together and made presentations—short ones, long ones—to diverse audiences all around the world. Our varied views started to converge in 2015, and this is when we thought of writing a book about technical debt. It proved to be still a bit of a moving target.

We interacted with many people over the past eight years or so, and the book you have in hand is the result of these collaborations with hundreds of people. With their help, we made great strides in understanding the phenomenon behind the simple metaphor of technical debt. We think we now better understand where technical debt comes from, what consequences it has on software-intensive development projects, and what form this technical debt actually takes. We now say with certainty that all systems have technical debt, and managing technical debt is a key software engineering practice to master for any software endeavor to succeed. We've heard how different organizations cope with it. We looked at and tried tools promising to perform miracles with technical debt. We also understood the limits of the simple financial metaphor: We realize now that technical debt is not quite like your house mortgage.

This book is intended for the many practitioners who've heard the term and those who think that it may have some relevance in their context. Hopefully it will give you tools to analyze your own situation and put names on events and artifacts you are confronted with.

This is not a scientific treatise, full of data and statistics. There are other venues for this. But we will give you concrete examples that you can relate to. It is also illustrated with stories that some of our friends from our industry have contributed, telling you their experience of technical debt in their own words.

Philippe: I now see that my 1980s story about 8 kB of ROM is a very clear-cut case of technical debt, triggered by pure schedule pressure, with severe consequences on the maintainability of this small piece of code. I attended the 1992 OOPSLA

conference in Vancouver where Ward Cunningham used the term "technical debt" for the first time. At last I had a name for it.

Robert: Reflecting on the two projects adopting agile, I first approached the problem thinking that architecture infrastructure needed to be equally visible as features in the product backlog. That gave me some, but not all, the tools I needed to understand the choice in selecting one or the other. I now see that adding technical debt items to the backlog brings visibility to the long-term consequences of the choices as they are made together with more needed tools to strategically plan and monitor those choices as technical debt.

Ipek: A few months ago in one of the software architecture courses I teach at the Software Engineering Institute (SEI), an attendee approached me to ask if I had ever worked on the GIS-X system. He happened to be one of the engineering managers on the team. He recalled our recommendations and in reflection reassured me that while at the time we did not phrase our findings using the words, we were spot on that the technical debt they had resulted in the project being canceled. A full circle moment.

It does not stop here. Now you will have to share with us and the community *your* stories about technical debt. This book is not the end…only a start.

Philippe Kruchten, Vancouver
Robert Nord, Pittsburgh
Ipek Ozkaya, Pittsburgh

This page intentionally left blank

Acknowledgments

Many colleagues attended the Managing Technical Debt (MTD) workshops over the years that provided an opportunity to exchange ideas and improve practice. The idea of the technical debt landscape grew out of a working session at the Third International Workshop on Technical Debt at the International Conference on Software Engineering (ICSE) in Zurich in 2012. A week-long Dagstuhl Seminar on Managing Technical Debt in Software Engineering in 2016 produced a consensus definition for technical debt, a draft conceptual model, and a research roadmap. Paris Avgeriou and Carolyn Seaman, early pioneers in managing technical debt, joined us in organizing events and guiding the community. Tom Zimmermann provided generous support from ICSE as the MTD workshop series transformed into a conference. He helped make the inaugural edition of the TechDebt Conference in 2018 a success where researchers, practitioners, and tool vendors could explore theoretical and practical techniques that manage technical debt.

We are grateful to Robert Eisenberg, Michael Keeling, Ben Northrop, Linda Northrop, Eltjo Poort, and Eoin Woods, who shared their experience and wisdom in the form of sidebars. We also appreciate the software engineers, developers, project managers, and people on the business side of the organization for sharing their stories and practices from the trenches.

Special thanks to Len Bass and Hasan Yasar, who contributed their expertise to the chapter on technical debt and production. Kevin Sullivan presented the net present value (NPV) and real options example at our very first workshop on technical debt in 2010, and Steve McConnell refined it in subsequent discussions.

Thanks go to the experts for their thorough and helpful reviews of different drafts of the manuscript that helped make this a better book. These include Paris Avgeriou, Felix Bachmann, Len Bass, Stephany Bellomo, Robert Eisenberg, Neil Ernst, George Fairbanks, Shane Hastie, James Ivers, Clemente Izurieta, Rick Kazman, Nicolas Kruchten, Jean-Louis Letouzey, Ben Northrop, Linda Northrop, Eltjo Poort, Chris Richardson, Walker Royce, Carolyn Seaman, Eoin Woods, and Hasan Yasar.

At the SEI, James Ivers, head of the SEI's Architecture Practices initiative, provided steady and persistent support for this effort. The SEI has been involved in technical debt research for many years, and the work of our colleagues helped shape our thinking on the topic with contributions from Felix Bachmann, Stephany Bellomo, Nanette Brown, Neil Ernst, Ian Gorton, Rick Kazman, Zach Kurtz, and Forrest

Shull. Linda Northrop led the SEI program that was instrumental in the development of the field of software architecture and in influencing our ideas about architecture in the technical debt landscape. She was also our mentor throughout the journey. Jim Over, Anita Carleton, and Paul Nielsen supported transitioning the work in managing technical debt to practice, including this book. Thanks to Kurt Hess for working with us to transform many of the concepts into the figures that illustrate the book. Tamara Marshall-Keim was invaluable in helping us untangle and clearly communicate complex concepts. Her knowledge of the domain and editing expertise made significant improvements to the content of the book.

At the University of British Columbia, we thank graduate students Erin Lim, Ke Dai, and Jen Tsu Hsu, who went boldly into the wild world of software and system development and investigated what technical debt actually looked like. And more recently another student, Mike Marinescu, helped us with the book production.

At Pearson Education, Kim Spenceley and Chris Guzikowski provided guidance and support. Our thanks also go to our copy editor, Kitty Wilson, production editor, Lori Lyons, and the team of production professionals.

Finally, we thank our families and friends for their encouragement and support.

About the Authors

Philippe Kruchten is a professor of software engineering at the University of British Columbia in Vancouver, Canada. He joined academia in 2004, after a 30+-year career in industry, where he worked mostly with large software-intensive systems design in the domains of telecommunication, defense, aerospace, and transportation. Some of his experience in software development is embodied in the Rational Unified Process (RUP), whose development he directed from 1995 until 2003. He's the author or co-author of *Rational Unified Process: An Introduction* (Addison-Wesley, 1998), *RUP Made Easy: A Practitioner's Guide* (Addison-Wesley, 2003), and *Software Engineering with UPEDU* (Addison-Wesley, 2003), as well as earlier books about programming in Pascal and Ada. He received a doctoral degree in information systems (1986) and a mechanical engineering degree (1975) from French engineering schools.

Robert Nord is a principal researcher at the Carnegie Mellon University Software Engineering Institute, where he works to develop and communicate effective methods and practices for agile at scale, software architecture, and managing technical debt. He is coauthor of the practitioner-oriented books *Applied Software Architecture* (Addison-Wesley, 2000) and *Documenting Software Architectures: Views and Beyond* (Addison-Wesley, 2011) and lectures on architecture-centric approaches. He received a PhD in computer science from Carnegie Mellon University and is a distinguished member of the ACM.

Ipek Ozkaya is a principal researcher at the Carnegie Mellon University Software Engineering Institute. Her primary work includes developing techniques for improving software development efficiency and system evolution, with an emphasis on software architecture practices, software economics, agile development, and managing technical debt in complex, large-scale software-intensive systems. In addition, as part of her responsibilities, she works with government and industry organizations to improve their software architecture practices. She received a PhD in Computational Design from Carnegie Mellon University. Ozkaya is a senior member of IEEE and the 2019–2021 editor-in-chief of *IEEE Software* magazine.

This page intentionally left blank

About the Contributors

Robert Eisenberg is a retired Lockheed Martin Fellow with more than 30 years of experience in the full lifecycle development of large-scale software systems. His areas of expertise include software methodologies and processes, schedule and earned value management, agile transformation, and technical debt management. He led the Lockheed Martin corporate initiative on the development of practices and methods for managing technical debt and assisted many programs in their application. Robert also led the Lockheed Martin Space Systems business area initiative to develop and implement new business models and practices based on lean and agile principles. He has presented at multiple workshops and conferences on both technical debt management and agile methods, practices, and transformation. Robert received an MS in computer science from the University of Virginia and a BS in computer science from the University of Delaware.

Michael Keeling is a professional software engineer and the author of *Design It! From Programmer to Software Architect* (Pragmatic Bookshelf, 2017). Keeling currently works at LendingHome and has also worked at IBM, Vivisimo, BuzzHoney, and Black Knight Technology. Keeling has an MS degree in software engineering from Carnegie Mellon University and a BS degree in computer science from the College of William and Mary. Contact him via Twitter @michaelkeeling or his website, https://www.neverletdown.net.

Ben Northrop is the founder of Highline Solutions, a Pittsburgh-based digital consultancy focused on the architecture, development, and deployment of large-scale custom software systems. In his 20 years of experience, Ben has helped to build dozens of systems across a number of industries, including transportation, finance, telecommunications, higher education, and retail. He holds two degrees from Carnegie Mellon University: a BS in Information and Decision Systems and an MS in Logic, Computation, and Methodology. His writing can be found at www.bennorthrop.com.

Linda Northrop has more than 45 years of experience in the software development field as a practitioner, researcher, manager, consultant, author, speaker, and educator. She is a Fellow at Carnegie Mellon University's Software Engineering Institute

(SEI). Under her leadership, the SEI developed software architecture and product line methods and a series of highly acclaimed books and courses that are used worldwide. Northrop also co-authored *Software Product Lines: Practices and Patterns* (Addison-Wesley, 2002). She led a cross-disciplinary, national research group on ultra-large-scale systems that resulted in the book *Ultra-Large-Scale Systems: The Software Challenge of the Future*. Her current professional interests are software architecture, ultra-large-scale systems, and software innovations to aid children with different abilities. Find Linda Northrop at http://www.sei.cmu.edu/about/people/profile.cfm?id=northrop_13182.

Eltjo R. Poort leads the architecture practice at CGI in the Netherlands. In his 30-year career in the software industry, he has fulfilled many engineering and project management roles. In the 1990s, he oversaw the implementation of the first SMS text messaging systems in the United States. In the past decade, he has produced various publications on improving architecting practices, including his PhD thesis in 2012. Eltjo is best known for his work on risk- and cost-driven architecture, a set of principles and practices for agile solution architecting, for which he received the Linda Northrop Software Architecture Award in 2016. His solution architecture blog can be found at eltjopoort.nl. In his spare time, Eltjo plays the violin in Symfonieorkest Nijmegen. Eltjo is a member of the IFIP Working Group 2.11 on Software Architecture.

Eoin Woods is the CTO of Endava, a technology company that delivers projects in the areas of digital, agile, and automation. Prior to joining Endava, Eoin worked in the software engineering industry for 20 years, developing system software products and complex applications in the capital markets domain. His main technical interests are software architecture, distributed systems, and computer security. He is editor of the *IEEE Software* "Pragmatic Architect" column, co-authored the well-known software architecture book *Software Systems Architecture* (Addison-Wesley, 2011), and received the 2018 Linda M. Northrop Award for Software Architecture, awarded by the SEI at Carnegie Mellon University. Eoin can be contacted via his website, at www.eoinwoods.info.

Acronyms

5W	Five Ws: Who, What, Where, When, Why
A2DAM	Agile Alliance Debt Analysis Method
AADL	Architecture Analysis and Design Language
ADL	Architecture Description Language
ALM	Application Lifecycle Management
API	Application Programming Interface
ATAM	Architecture Tradeoff Analysis Method
CISQ	Consortium for IT Software Quality
CVE	Common Vulnerabilities and Exposures
CWE	Common Weakness Enumeration
DB	Database
FLOSS	Free, Libre, Open-Source Software
FTE	Full-time Equivalent
I18N	Internationalization
IRAD	Independent Research and Development
ISO	International Organization for Standardization
L10N	Localization
MVP	Minimum Viable Product
NPV	Net Present Value
OMG	Object Management Group
ROI	Return On Investment
SaaS	Software as a Service
SAFe®	Scaled Agile Framework®
SLOC	Source Lines of Code
SOA	Service-Oriented Architecture
SQALE	Software Quality Assessment based on Lifecycle Expectations
SysML	Systems Modeling Language
UML	Unified Modeling Language
UX	User Experience

This page intentionally left blank

SEI Figures for Managing Technical Debt

Special permission to reproduce portions of the following texts and images was granted by the Software Engineering Institute:

Chapter	Page Number	Figure Number	Description
1	Page 14	P1-1	Principle 1: Technical debt reifies an abstract concept
	Page 15	F1-1	Major concepts of technical debt
2	Page 20	F2-1	Technical Debt Landscape
	Page 24	C2-A	Solution U is cheaper than V
	Page 25	C2-B	W over V is cheaper than W over U
	Page 26	C2-C	Pay interest, or repay the principal
	Page 27	C2-D	Pay more interest, or repay the higher principal
	Page 32	P2-2	Principle 2: If you do not incur any form of interest, then you probably do not have actual technical debt
	Page 33	2-2	Technical Debt Timeline
3	Page 37	F3-1	"It depends": The many factors of context
	Page 45	P3-3	Principle 3: All Systems Have Technical Debt
4	Page 53	F4-1	Timeline: Reaching the awareness point
	Page 55	P4-4	Technical debt must trace to the system
	Page 60	F4-2	Identifying technical debt items
	Page 63	F4-3	The four things to do in development product backlog
5	Page 66	F5-1	Results of the code analysis for Phoebe
	Page 67	P5-5	Technical debt is not synonymous with bad quality

Part I

Exploring the Technical Debt Landscape

This page intentionally left blank

Chapter 1

Friction in Software Development

There is still much friction in the process of crafting complex software; the goal of creating quality software in a repeatable and sustainable manner remains elusive to many organizations, especially those who are driven to develop in Internet time.

—Grady Booch

Is the productivity of your software organization going down? Is your code base harder and harder to evolve every week? Is the morale of your team declining? As with many other successful software endeavors, you are probably suffering from the inability to manage friction in your software development and may have a pervasive case of technical debt.

Why should you care about technical debt? How does it manifest itself? How is it different from software quality? In this chapter, we introduce the metaphor of technical debt and present typical situations where it exists.

The Promise of Managing Technical Debt

Understanding and managing technical debt is an attractive goal for many organizations. Proactively managing technical debt promises to give organizations the ability to control the cost of change in a way that integrates technical decision making and software economics seamlessly with software engineering delivery.

The term *technical debt* is not new. Ward Cunningham introduced it in 1992 to communicate the delicate balance between speed and rework in pursuit of delivering functioning quality software. And the concepts it encompasses are not new either.

Ever since we started creating software products, we have been grappling with this issue under other names: software maintenance, software evolution, software aging, software decay, software system reengineering, and so on.

You can think of technical debt as an analogy with friction in mechanical devices; the more friction a device experiences due to wear and tear, lack of lubrication, or bad design, the harder it is to move the device, and the more energy you have to apply to get the original effect. At the same time, friction is a necessary condition of mechanical parts working together. You cannot eliminate it completely; you can only reduce its impact.

Slowly, over the past ten years, many large companies whose livelihoods depend on software have realized that technical debt, under this or any other name, is very real and crippling their ability to satisfy customer desires. Technical debt has started to translate into financial impact. At some point in the past, companies may have made a trade-off to take on technical debt to deliver quickly or scale quickly, threw more people at the problem when the debt mounted, and never reduced or managed the debt. It is not a proper debt, from an accounting perspective, but the specter of huge costs somewhere on the path ahead will negatively affect the company's financial bottom line. Government organizations that are large buyers of software also now realize that focusing only on initial development cost obscures the full cost of the software; they have begun to demand justification of all lifecycle costs from the software industry.

Technical debt is pervasive: It affects all aspects of software engineering, from requirements handling to design, code writing, the tools used for analyzing and modifying code, and deployment to the user base. The friction caused by technical debt is even apparent in the management of software development organizations, in the social aspect of software engineering. Technical debt is the mirror image of software technical sustainability; Becker and colleagues (2015) described technical debt as "the longevity of information, systems, and infrastructure and their adequate evolution with changing surrounding conditions. It includes maintenance, innovation, obsolescence, data integrity, etc." And it relates to the wider concern of sustainability in the software industry—not only in the environmental sense but also in the social and technical senses.

Progress on managing technical debt has been piecewise, and the workforce tends to devalue this type of debt. So it remains a problem. Why do we think that understanding and managing the problem as technical debt will have a different outcome? Software engineering as a discipline is at a unique point at which several subdisciplines have matured to be part of the answer to the technical debt question. For example, program analysis techniques, although not new, have recently become sophisticated enough to be useful in industrial development environments. So, they're positioned to play a role in identifying technical debt in a way they weren't a few years ago. DevOps tooling environments that incorporate operations and development further allow developers to analyze their code, locate issues before they

become debt, and implement a faster development lifecycle. Developers also now have the vocabulary to talk about technical debt as part of their software development process and practices.

The technical debt concept resonates well with developers, as they look for a well-defined approach to help understand the complex dependencies between software artifacts, development teams, and decision makers and how to balance short-term needs to keep the software product running with long-term changes to keep the product viable for decades. In this way, technical debt can also be seen as a kind of strategic investment and a way to mitigate risk.

Technical Debt A-B-C

Many practitioners today see *technical debt* as a somewhat evasive term to designate poor internal code quality. This is only partly true. In this book, we will show that technical debt may often have less to do with intrinsic code quality than with design strategy implemented over time. Technical debt may accrue at the level of overall system design or system architecture, even in systems with great code quality. It may also result from external events not under the control of the designers and implementers of the system.

This book is dedicated to defining principles and practices for managing technical debt—defining it, dissecting it, providing examples to study it from various angles, and suggesting techniques to manage it. Our definition of technical debt is as follows:

> In software-intensive systems, technical debt consists of design or implementation constructs that are expedient in the short term but that set up a technical context that can make a future change more costly or impossible. Technical debt is a contingent liability whose impact is limited to internal system qualities—primarily, but not only, maintainability and evolvability.

We like this definition because it does not fall into the trap of considering only the financial metaphor implied by the term *debt*. Although the metaphor carries an interesting financial analogy, technical debt in software is not quite like a variable-rate mortgage or an auto loan. It begins and accumulates in development artifacts such as design decisions and code.

Technical debt also has a contingent aspect that depends on something else that might or might not happen: How much technical debt you need to worry about depends on how you want the system to evolve. We like that this definition does not include defects in functionality (faults and failures) or external quality deficiencies (serviceability), as lumping together defects and technical debt muddies the water. System qualities, or quality attributes, are properties of a system used to indicate

how well the system satisfies the needs of its stakeholders. The focus on internal quality is the lens through which these deficiencies are seen from the viewpoint of the cost of change. Technical debt makes the system less maintainable and more difficult to evolve.

Technical debt is not a new concept. It is related to what practitioners have for decades been calling *software evolution* and *software maintenance*, and it has plagued the industry ever since developers first produced valuable software that they did not plan to throw away or replace with new software but instead wanted to evolve or simply maintain over time. The difference today is the increasing awareness that technical debt, if not managed well, will bankrupt the software development industry. Practitioners today have no choice but to treat technical debt management as one of the core software engineering practices.

While technical debt can have dire consequences, it is not always as ominous as it may sound. You can look at it as part of an overall investment strategy, a strategic software design choice. If you find yourself spending all your time dealing with debt or you reach the point where you cannot repay it, you have incurred bad debt. When you borrow or leverage time and effort that you can and will repay in the future, you may have incurred good debt. If the software product is successful, this strategy can provide you with greater returns than if you had remained debt free. In addition, you might also have the option to simply walk away from your debt if the software is not successful. This dual nature of technical debt—both good and bad—makes grappling with it a bit confusing for many practitioners.

We will return to the financial metaphor later to investigate whether there are some software equivalencies to the financial ideas of principal, interest, repayment, and even bankruptcy.

Examples of Technical Debt

To illustrate our definition, we offer a few stories about technical debt in software development projects. You will see organizations struggling with their technical debt and software development teams failing to strategize about it.

Quick-and-Dirty if-then-else

A company in Canada developed a good product for its local customers. Based on local success, the company decided to extend the market to the rest of Canada and immediately faced a new challenge: addressing the 20% of Canada that uses the French language in most aspects of life. The developers labored for a week to produce a French version of the product, planting a global flag for French = Yes or No as well as hundreds of if-then-else statements all over the code. A product demo went smoothly, and they got the sale!

> Then, a month later, on a trip to Japan, a salesperson proudly boasted that the software was multilingual, returned to Canada with a potential order, and assumed that a Japanese version was only one week of work away. Now the decision not to use a more sophisticated strategy—such as externalizing all the text strings and using an internationalization package—was badly hurting the developers. They would not only have to select and implement a scalable and maintainable strategy but also have to undo all the quick-and-dirty if-then-else statements.

For the Canadian company, the decision to use if-then-else statements spread the change throughout the code, but it was a necessary quick-and-dirty solution from a business perspective to get a quick sale. Doing the right thing at that stage would have postponed the delivery of the system and likely lost them the deal. So even though the resulting code was ugly—as well as hard to modify and evolve—it was the right decision. Now, would you continue down that path and add another layer of if-then-else for each language? Or would you rethink the strategy and decide to repay the original technical debt? Inserting the Japanese version of the quick fix, with its issues of character sets and vertical text, would be too much of a burden and a subsequent maintenance issue. You may argue that a good designer would have set up provisions for internationalization and localization right at the outset, but this is easy to say in hindsight; the demands and constraints at the beginning of development for this small venture were quite different, focused on the main features, and didn't foresee the need for a multilingual feature.

Hitting the Wall

> Two large global financial institutions merged. As a result, two IT systems essential to their business had to merge. The management of the new company determined that a duct-tape and rubber-band system, mixing the two systems in some kind of chimera, would not work. They decided to build a support system from scratch, using more recent technologies and, in some ways, walking away from years of accumulated technical debt in the original systems.
>
> The company organized a team to build the new replacement system. They progressed rapidly because the first major release was to provide an exact replacement of the existing systems. In a few months, they accumulated a lot of code that performed well in demos for each one-week "sprint" (or iteration). But nobody thought about the architecture of the system; everyone focused on creating more and more features for the demo. Finally, some harder issues of scalability, data management, distribution of the system, and security began to surface, and the team discovered that refactoring the mass of code already produced to address these issues was rapidly leading them to a complete stop. They hit the wall, as marathon runners would say. They had lots of code but no explicit architecture. In six months, the organization had accumulated a massive amount of technical debt that brought them to a standstill.

The situation here is very different from the first case. This was not an issue of code quality. It was an issue of foresight. The development team neglected to consider architectural and technology selection issues or learn from the two existing systems at

appropriate times during development; the team did not need to do all of that up front, but it needed to do it early enough not to burden the project downstream. Refactoring is valuable, but it has limits. The development team had to throw away large portions of the existing code weeks after its original production. Although the organization hoped to eliminate technical debt when it decided to implement a brand-new system after the merger, it failed to incorporate eliminating technical debt into the project management strategy for the new system. Ignorance is bliss—but only for a while.

Crumbling Under the Load

A successful company in the maritime equipment industry successfully evolved its products for 16 years, in the process amassing 3 million lines of code. Over these 16 years, the company launched many different products, all under warranty or maintenance contracts; new technologies evolved; staff turned over; and new competitors entered the industry.

The company's products were hard to evolve. Small changes or additions led to large amounts of work in regression testing with the existing products, and much of the testing had to be done manually, over several days per release. Small changes often broke the code, for reasons unsuspected by the new members of the development team, because many of the design and program choices were not documented.

In the case of the maritime equipment company, there was no single cause of technical debt. There were hundreds of causes: code imperfections, tricks, and workarounds, compounded by no usable documentation and little automated testing. While the development team dreams of a complete rewrite, the economic situation does not allow delaying new releases or new products or abandoning support for older products. Some intermediate strategy must be implemented.

Death by a Thousand Cuts

One IT-service organization landed several major contracts. Some of this new business allowed the organization to grow its offshore development businesses and enter emerging software development markets. For several years, the organization experienced a hiring boom.

The IT-service projects were similar in nature, and the organization assumed that its new developers were interchangeable across projects. The project managers thought, "The task is customization of the same or similar software, so how different could it be?" But in some cases, the new employees lacked the right skills or knowledge about the packages used. In other cases, time and revenue-growth pressures pushed them to skip testing the code thoroughly or fail to think through their designs. They also did not put in the time to create common application programming interfaces (APIs). The hiring boom created unstable teams, with new members introduced almost every month. It even became an internal joke: "Get a bunch of online Java and Microsoft certifications, and you are a senior developer here." In no time, the project managers lost control of the schedule as well as the number of defects introduced into the system.

This IT-service organization provides another example in which there is no single source of technical debt. We call this "death by a thousand cuts" because a pervasive lack of competence can result in many small, avoidable coding issues that are never caught. Lack of organizational competency—as in the case of this IT-service organization—easily activates a number of cascading effects. The unplanned and unmanaged hiring boom, the missed opportunity to enforce commonality across the products, and the limited testing all contributed to the accumulating technical debt.

Tactical Investment

> A five-person company developed a web application in the urban transportation domain, targeted at users of buses and trains. In this relatively new and rapidly evolving domain, the targeted users could not really tell the company what they would need. "I'll know it when I see it" was the general response. So, the company developed a "minimum viable product" (MVP) with some core functionality and little underlying sophistication. Members of the company beta-tested it with about 100 users in one city. They had to "pivot" several times until they found their niche, at which point they invested heavily in building the right infrastructure for a product that would be able to support millions of simultaneous users and adapt to dozens of situations and cities.

The initial shortcuts that members of this small company took and the high-level rudimentary infrastructure they initially developed are examples of technical debt wisely assumed. The company borrowed the time it would have spent on the complete definition and implementation of the infrastructure to deliver early. This allowed it to complete an MVP months earlier than traditional development practices, which put the infrastructure first, would have allowed. Moreover, the company learned useful lessons about the key issues (which did not necessarily match its initial assumptions) of reliability, fault tolerance, adaptability, and portability. Building in these quality attributes up front would have created massive rework once the developers understood more completely what their users needed.

All along, members of this company were aware of the deliberate shortcuts they were taking and their consequences on future development. From the perspective of their angel investors, these were good strategies for risk management; if the company found no traction in the market, the developers could stop development early and minimize cost before the company made massive financial investments. Management also made it very clear to everyone, internal and external, that the shortcuts were temporary solutions so that no one would be tempted to keep them, painfully patched, as part of the permanent solution. In this manner, taking on technical debt was a wise investment that paid off. The company repaid the "borrowed time," but it could also have walked away from the project.

In all these examples, the current state of the software carries code that works, but it makes further evolutions harder. The debt was induced by lack of foresight, time constraints, significant changes in requirements, or changes in the business context.

Software Crisis Redux

You have likely seen the symptoms and heard stories of technical debt similar to those just shared: teams spending almost all of their time fixing defects and continuously slipping on deadlines for shipping new technology; teams discovering incompatibilities despite continuous integration efforts and spending time on out-of-cycle rework; recurring user complaints about functionality that appears to be already fixed several times; outdated technology and platforms requiring convoluted workarounds and presenting challenges for upgrading; and a team admitting that the solution it had a year ago to make the system work is not good enough anymore. For organizations that want to sustain continuous growth and revenue, these are problems. And for some companies, these problems look like an impending new software crisis.

Ever since the famous 1969 NATO Software Engineering Conference heralded the birth of software engineering, the industry has been in a constant state of crisis. In his 1972 ACM Turing Award Lecture, the software pioneer Edsger Dijkstra said, "But in the next decades something completely different happened: more powerful machines became available, not just an order of magnitude more powerful, even several orders of magnitude more powerful. But instead of finding ourselves in the state of eternal bliss of all programming problems solved, we found ourselves up to our necks in the software crisis! How come?"

The software crisis took root and grew. In 1994 Wayt Gibbs wrote in *Scientific American* that "despite 50 years of progress, the software industry remains years—perhaps decades—short of the mature engineering discipline needed to meet the demands of an information-age society."

Fast-forward to today. After a series of breathtaking innovations—including new technologies, new tools, and the software development workforce increasing tenfold—the software industry is still in crisis. But now the nature of the issues has shifted. The industry is crushed under the mass of existing software, which consumes more than half of the available software development workforce. Data analysis organizations estimate that the global

maintenance backlogs for information technology software amount to $1 trillion of technical debt. Government budgets struggle with legacy code built on top of poorly designed architectural foundations and outdated technology. Globally, software practitioners grasp the impact of technical debt and know how systems acquired their debt but fail to recognize managing technical debt as an essential aspect of running a successful software organization and developing successful software-enabled products. The problem is not new, but the industry is feeling it more acutely now than it has in the past.

Software development is an industry, and it can be sustained as an industrial activity only if it is economically viable. As more and more software is being developed, its long-term sustainment becomes less and less viable. Markets demand new applications and systems—and they demand them very rapidly. Some of these applications are ephemeral and have shelf lives of a few months or years, but some—the most successful ones and usually the largest ones—must be maintained for many years or for decades.

Today this is the biggest hurdle in software engineering: How should a development organization cope with this rapidly expanding software base while keeping it secure, running with up-to-date technology, and meeting its business and user goals in an economically viable way?

Your Own Story About Technical Debt?

Now that we have given you a taste of the various flavors of technical debt, maybe you can identify with some of the stories: "Oh, yes, we have some of this here, too!" or "Now this thing we suffer from has a name: technical debt!" You could add your own development (or horror) story here. Over the past few years, the authors of this book have heard similar stories from dozens of companies. These organizations became mired in technical debt from different paths, with different concerns and different consequences. We have heard enough of these stories to classify them into awareness levels about technical debt:

- **Level 1:** Some companies have told us they had never heard the term or the concept technical debt, but it was not difficult for them to see that part of their problem is some form of technical debt.
- **Level 2:** Some companies have heard of the concept, have seen blog posts on the topic, and can provide examples of their technical debt, but they do not

know how to move from understanding the concept of technical debt to operationally managing it in their organization.

- **Level 3:** In some organizations, development teams are aware that they have incurred technical debt, but they do not know how to get the management or business side of the company to acknowledge its existence or do anything about it.
- **Level 4:** Some organizations know how to make technical debt visible, and they have some limited team-level strategies to better manage it, but they lack analytical tools to help them decide what to do about technical debt and which part of it to address first.
- **Level 5:** We have not heard from many organizations that respond, "Thank you, all the technical debt is under control." If this describes your organization, we would love to hear from you about your successful software product.

This feels a bit like the levels of a "TDMM"—Technical Debt Maturity Model—doesn't it? Regardless of the level you feel you're at, this book has something for you.

Who Is This Book For?

There are many books and tools that can help you understand how to analyze your software. And there are yet other books that can help you adopt new technology for building microservices, migration to the cloud, front-end web development, and real-time system development. There are also many good books that walk through different aspects of software development, such as software code quality, software design patterns, software architecture, continuous integration, DevOps, and so on. The list is long. But there exists little practical guidance on demystifying how to recognize technical debt, how to communicate it, and how to proactively manage it in a software development organization. This book fills that gap.

We address the roles involved in managing technical debt in a software development organization, from developers and testers to technical leads, architects, user experience (UX) designers, and business analysts. We also address the relationship of technical debt to the management of organizations and the business leaders.

People close to the code should understand how technical debt manifests itself, what form it takes in the code, and the tools and techniques they can use to identify, inventory, and manage technical debt. This is the inside-out perspective.

People facing the customers—the business side of the organization, such as product definition, sales, support, and the C-level executives—should understand how

schedule pressure and changes of direction (product "pivot") drive the accumulation of technical debt. They should be especially conscious of how much the organization should "invest" in technical debt, without repayment, and for how long. This is the outside-in perspective.

Both sides of the software development organization—technical and code-facing or business and customer-facing—should understand the reasoning and decision processes that lead to incurring technical debt and how the consequences of debt result in reduced capacity. They should also understand the decision processes involved in paying back technical debt and getting development back on track. These decisions are not merely technical. For sure, technical debt is embedded in the code base and a few connected artifacts. But its roots and its consequences are at the business level. All involved should understand that managing technical debt requires the business and technical sides of the organization to work together.

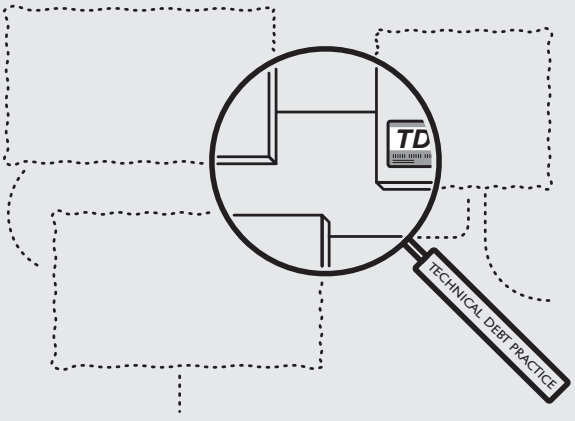
Principles of Technical Debt Management

As we progress through the book, we will identify a small number of key software engineering principles that express universal truths related to technical debt. They are rooted in our experience with technical debt in industry and government software projects, and they are accepted or at least acceptable by the software engineering community. The nine software engineering principles follow:

Principle 1: Technical debt reifies an abstract concept.
Principle 2: If you do not incur any form of interest, then you probably do not have actual technical debt.
Principle 3: All systems have technical debt.
Principle 4: Technical debt must trace to the system.
Principle 5: Technical debt is not synonymous with bad quality.
Principle 6: Architecture technical debt has the highest cost of ownership.
Principle 7: All code matters!
Principle 8: Technical debt has no absolute measure—neither for principal nor interest.
Principle 9: Technical debt depends on the future evolution of the system.

Here is our first principle.

Principle 1: Technical Debt Reifies an Abstract Concept



Technical debt is a useful rhetorical concept for fostering dialogue between business and technical people in a software development organization. On one hand, technical people do not always appreciate the value of shorter time to market, quick delivery, and rapid tactical changes of direction; on the other hand, business people do not always realize the dramatic impact some earlier design decisions can make in a software project and the costs they can lead to downstream. By identifying concrete items of technical debt, considering their impact over time, evaluating the lifecycle costs associated with them, and introducing mechanisms for expressing technical debt and estimating its impact, an organization can help everyone better understand the pains of software evolution and make the economic consequences more real and tangible. Then both technical and business people can plan how to reduce technical debt just as they plan new features, fix defects, and construct architectural elements.

We'll introduce more principles in the following chapters, and you will find them summarized in the final chapter of the book.

Navigating the Concepts of the Book

The goal for this book is to provide practical information that will jump-start your ability to manage technical debt. The chapters that follow inform the basic steps of technical debt management: become aware of the concept, assess the software development state for potential causes of technical debt, build a registry of technical debt, decide what to fix (and what not to fix), and take action during release planning. The

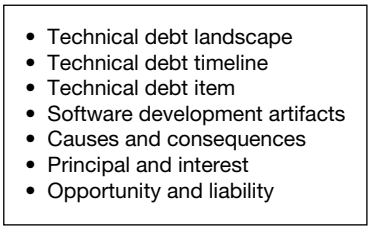


Figure 1.1 *Major concepts of technical debt*

steps draw on the seven interrelated concepts shown in Figure 1.1 that are the basis for managing technical debt.

This book organizes the chapters into four parts.

In Part I, "Exploring the Technical Debt Landscape"—Chapters 1, "Friction in Software Development," 2, "What Is Technical Debt?," and 3, "Moons of Saturn—The Crucial Role of Context"—we define technical debt and explain what is not technical debt. We introduce a conceptual model of technical debt and definitions and principles that we use throughout the book. We want to make technical debt an objective, tangible thing that can be described, inventoried, classified, and measured. To do this, we introduce the concept of the technical debt item—a single element of technical debt—something that can be clearly identified in the code or in some of the accompanying development artifacts, such as a design document, build script, test suite, user's guide, and so on. To keep with the financial metaphor, the cost impact of a technical debt item is composed of principal and interest. The principal is the cost savings gained by taking some initial expedient approach or shortcut in development—or what it would cost now to develop a different or better solution. The interest is the cost that adds up as time passes. There is recurring interest: additional cost incurred by the project in the presence of technical debt due to reduced productivity, induced defects, loss of quality, and problems with maintainability. And there is accruing interest: the additional cost of developing new software depending on not-quite-right code; evolvability is affected. These technical debt items are part of a technical debt timeline, during which they appear, get processed, and maybe disappear.

In Part II, "Analyzing Technical Debt"—Chapters 4, "Recognizing Technical Debt," 5, "Technical Debt and the Source Code," 6, "Technical Debt and Architecture," and 7, "Technical Debt and Production"—we cover how to associate with a technical debt item some useful information that will help you reason about it, assess it, and make decisions. You will learn how to trace an item to its causes and its consequences. The causes of a technical debt item are the processes, decisions, action, lack of action, or events that trigger the existence of a technical debt item. The consequences of technical debt items are many: They affect the value of the system and the cost (past, present, and future), directly or through schedule delays or future loss of quality. These causes and consequences are not likely to be in the code; they surface

in the processes and the environment of the project—for example, in the sales or the cost of support. Then we cover how to recognize technical debt and how technical debt manifests itself in source code, in the overall architecture of the system, and in the production infrastructure and delivery process. As you study technical debt more deeply, you'll notice that it takes different forms, and your map of the technical debt territory will expand to include this variety in the technical debt landscape.

In Part III, "Deciding What Technical Debt to Fix"—Chapters 8, "Costing the Technical Debt," and 9, "Servicing the Technical Debt"—we cover how to estimate the cost of technical debt items and decide what to fix. Decision making about the evolution of the system in most cases is driven by economic considerations, such as return on investment (for example, how much should you invest in the effort of software development in a given direction, and for what benefits?). For the technical debt items, we will consider principal and interest and associate elements of cost to reveal information about the resources to spend on remediation and the resulting cost savings of reducing recurring interest. We then revisit the technical debt items in the registry collectively and use information about the technical debt timeline to help determine which technical debt items should be paid off or serviced in some other way to ease the burden of technical debt: eliminate it, reduce it, mitigate it, or avoid it. We show how to make these decisions about technical debt reduction in the context of a business case that considers risk liability and opportunity cost.

In Part IV, "Managing Technical Debt Tactically and Strategically"—Chapters 10, "What Causes Technical Debt?," 11, "Technical Debt Credit Check," 12, "Avoiding Unintentional Debt," and 13, "Living with Your Technical Debt"—we provide guidance on how to manage technical debt. A key aspect of a successful technical debt management strategy is to recognize the causes in order to prevent future occurrences of technical debt items. Causes can be many, and they can be related to the business, the development process, how the team is organized, or the context of the project, to list a few. We present the Technical Debt Credit Check, which will help identify root causes of technical debt that show the need for software engineering practices that any team should incorporate into its software development activities to minimize the introduction of unintentional technical debt. The principles and practices you will have learned along the way make up a technical debt toolbox to assist you in managing technical debt.

At the end of each chapter, we recommend activities that you can do today and further reading related to the concepts, techniques, and ideas we discuss.

What Can You Do Today?

Apply the first principal by putting a name to your technical debt. Commit to applying a few basic techniques to your normal development practices as you read each chapter and continue to improve over time.

For Further Reading

The seminal paper that brought us the debt metaphor is the often-cited OOPSLA 1992 experience report, "The WyCash Portfolio Management System," by Ward Cunningham.

Steve McConnell (2007) provided one of the simplest and most accessible definitions of technical debt: "a design or construction approach that is expedient in the short term but that creates a technical context in which the same work will cost more to do later than it would cost to do now." Our current definition of technical debt was devised in a week-long workshop in Dagstuhl, Germany, in April 2016 (Avgeriou et al. 2016).

The software crisis was well described in 1994 by Wayt Gibbs, who interviewed many software pioneers and practitioners in industrial organizations, including Larry Druffel, Vic Basili, Brad Cox, and Bill Curtis.

A must-read is Fred Brooks' "No Silver Bullet" paper (Brooks 1986), which is also a chapter in the 10th anniversary edition of his famous book *The Mythical Man-Month* (Brooks 1995). Brooks reminds us that "There is no single development, in either technology or management technique, which by itself promises even one order-of-magnitude improvement within a decade in productivity, in reliability, in simplicity."

A durable software engineering principle should be a simple statement that expresses some universal truth; is "actionable" (that is, worded in a prescriptive manner); is independent of specific tools or tool vendors, techniques, or practices; can be tested in practice, where we can observe its consequences; and does not merely express a compromise between two alternatives. There are two classic books on software engineering principles: *201 Principles of Software Development* by Alan M. Davis (1995) and *Facts and Fallacies of Software Engineering* by Robert L. Glass (2003). In "Agile Principles as Software Engineering Principles," Norman Séguin (2012) did a thorough analysis of what constitutes a good software engineering principle—as opposed to a mere aphorism, wish, or platitude—and he debunked a few myths about principles.

Index

A

B

C

D

J – K

L

M

N

O

P

Q

R

S

T

U

V

W

X – Y – Z