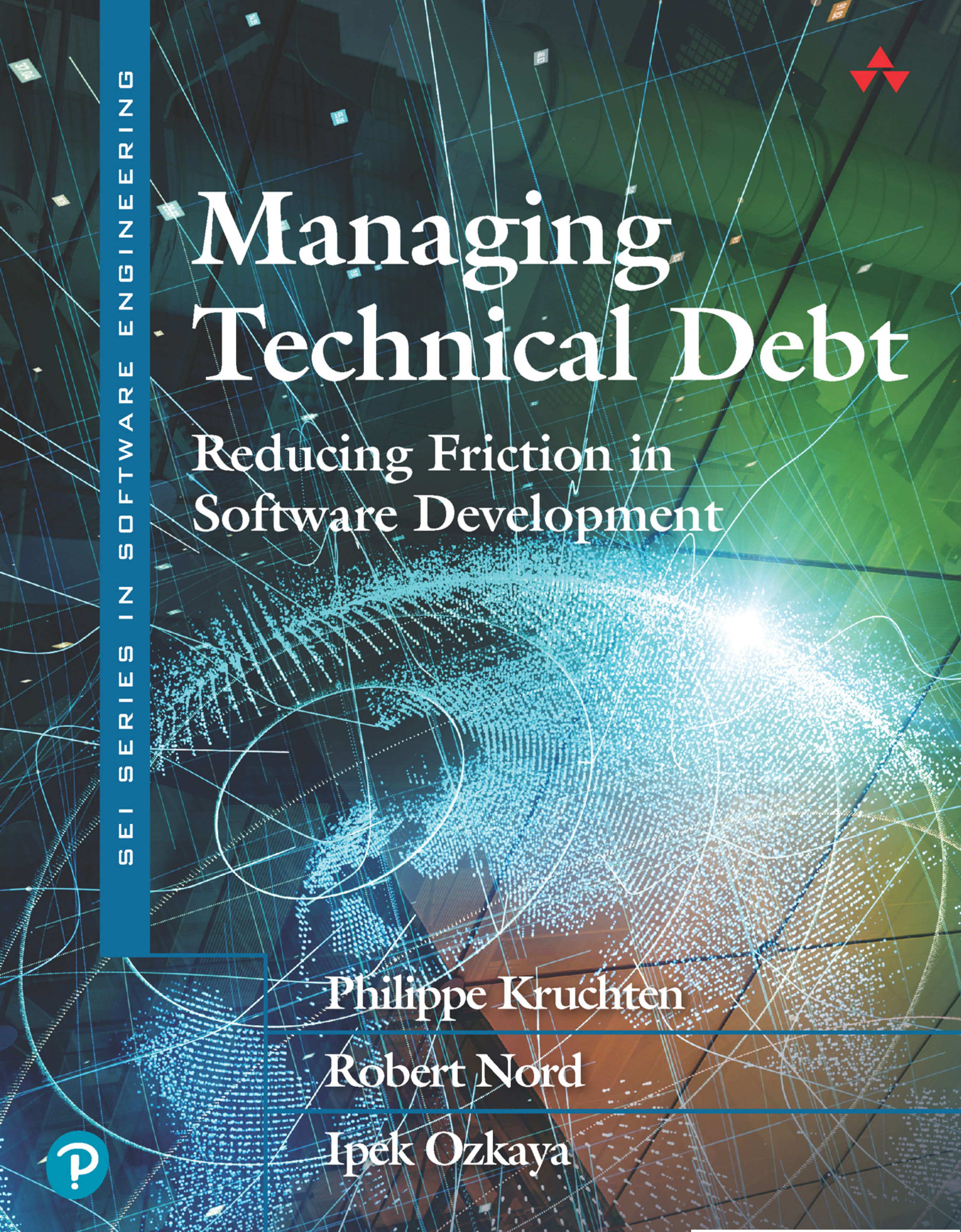


SEI SERIES IN SOFTWARE ENGINEERING

Managing Technical Debt

Reducing Friction in Software Development

Philippe Kruchten

Robert Nord

Ipek Ozkaya

Managing Technical Debt

The SEI Series in Software Engineering

Software Engineering Institute | Carnegie Mellon University

The CERT Guide to Insider Threats
How to Prevent, Detect, and Respond to Information Technology Crimes (Theft, Sabotage, Fraud)
Dawn Cappelli
Andrew Moore
Randall Trzeciak
CMMI for Development
Guidelines for Process Integration and Product Improvement
Third Edition
Mary Beth Chrissis
Mike Konrad
Sandy Shrum
Designing Software Architectures
A Practical Approach
Humberto Cervantes
Rick Kazman
Software Architecture in Practice
Third Edition
Len Bass
Paul Clements
Rick Kazman
The CERT C Coding Standard
98 Rules for Developing Safe, Reliable, and Secure Systems
Second Edition
Robert C. Seacord

Visit **informit.com/sei** for a complete list of available publications.

The SEI Series in Software Engineering is a collaborative undertaking of the Carnegie Mellon Software Engineering Institute (SEI) and Addison-Wesley to develop and publish books on software engineering and related topics. The common goal of the SEI and Addison-Wesley is to provide the most current information on these topics in a form that is easily usable by practitioners and students.

Titles in the series describe frameworks, tools, methods, and technologies designed to help organizations, teams, and individuals improve their technical or management capabilities. Some books describe processes and practices for developing higher-quality software, acquiring programs for complex systems, or delivering services more effectively. Other books focus on software and system architecture and product-line development. Still others, from the SEI's CERT Program, describe technologies and practices needed to manage software and network security risk. These and all titles in the series address critical problems in software engineering for which practical solutions are available.

Make sure to connect with us!
informit.com/socialconnect

Pearson
Addison-Wesley

informIT.com
the trusted technology learning source

O'REILLY
Safari

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

Register your copy of *Managing Technical Debt* at informit.com for convenient access to downloads, updates, and corrections as they become available. To start the registration process, go to informit.com/register and log in or create an account. Enter the product ISBN 9780135645932 and click Submit. Once the process is complete, you will find any available bonus content under “Registered Products.”

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

(SEI trademarks used in this book are registered trademarks of Carnegie Mellon University.)

Part I

Exploring the Technical Debt Landscape

Chapter 1: Friction in Software Development

Chapter 2: What Is Technical Debt?

Chapter 3: Moons of Saturn—The Crucial Role of Context

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.

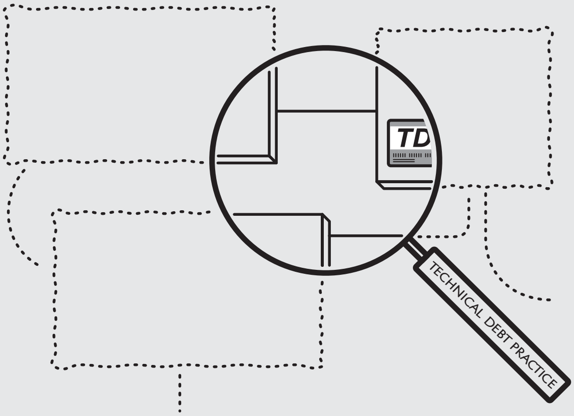
Principles of Technical Debt Management

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

- Technical debt landscape
- Technical debt timeline
- Technical debt item
- Software development artifacts
- Causes and consequences
- Principal and interest
- Opportunity and liability

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.

This page intentionally left blank

Chapter 2

What Is Technical Debt?

Drawing from a financial metaphor, the concept of technical debt shifts the conversation about decision making from a technical standpoint or an economic standpoint to a place where developers and managers can better understand the trade-offs and compromises in software development and decide on the way forward. In this chapter, we describe the technical debt landscape through the forms technical debt takes in different types of development artifacts across the software development lifecycle. We explore more thoroughly the concept of a technical debt item and its causes and economic consequences as principal and interest. We introduce the technical debt timeline to help you understand how technical debt unfolds over time.

Mapping the Territory

In Chapter 1, “Friction in Software Development,” we defined *technical debt* in software-intensive systems as the “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.” We added that “technical debt is a contingent liability whose impact is limited to internal system qualities—primarily, but not only, maintainability and evolvability.”

Technical debt is mostly invisible when looking at or using a software product. It manifests in two main ways: difficulty and additional cost in evolving the system (that is, adding new functionality) or maintaining the system (that is, keeping the system running when the technical environment changes). But concretely, opening the box and looking at technical debt at the software level reveals that it takes many different forms.

In this chapter, we survey the software development landscape with respect to technical debt and then dig a bit deeper into the technical and economic implications of this definition.

The Technical Debt Landscape

Figure 2.1 illustrates a typical technical debt landscape showing the software development issues that developers work on to improve the system. We distinguish the visible issues, such as new feature requests and defects that need to be fixed, from the mostly invisible issues, which are visible only to software developers. Issues related to evolution appear primarily on the left side of the figure; issues related to maintenance and quality appear primarily on the right.

Our focus is on the mostly invisible aspects of evolution and maintenance. Technical debt takes different forms in different types of development artifacts, such as the code, the architecture, and the production infrastructure. The different forms of technical debt affect the system in different ways.

The source code embodies many design and programming decisions. The code can be subjected to review, inspection, and analysis with static checkers to find issues of finer granularity: violations of coding standards, bad naming, code clones, unnecessary code complexity, and misleading or incorrect comments. Many of these symptoms of technical debt are referred to as *code smells*. When a system incurs technical debt at the source code level, the debt tends to hinder maintainability so that it will be hard to make corrections to the system when needed.

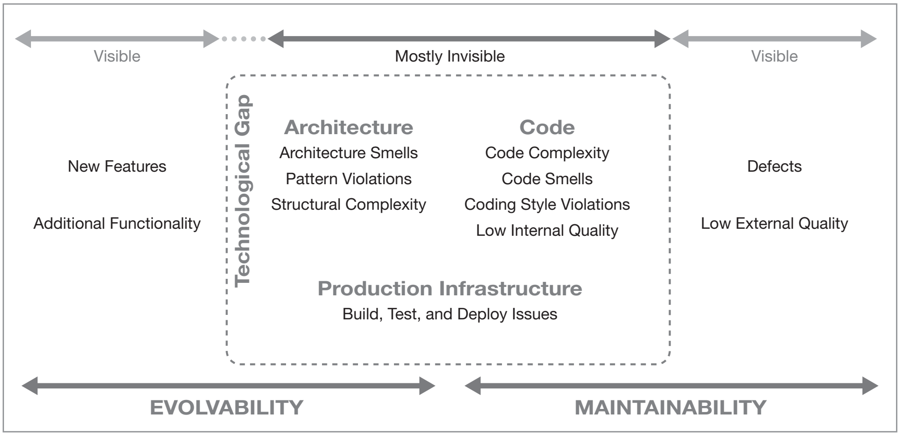


Figure 2.1 *The technical debt landscape*

Other technical debt items are more encompassing and pervasive. They involve choices about the structure or the architecture of the system: choice of platform, middleware, technologies for communication, user interface, or data persistency. Static code checkers don't find technical debt caused by these types of choices. For these elements, the principal and the interest are often higher than for technical debt in the code. When a system incurs technical debt at the architectural level, the debt tends to hinder evolvability: The system will be hard to extend to new features, with their functional and quality attribute requirements, such as scaling to a larger number of users, processing different kinds of data, and the like.

Not all technical debt is associated with bad internal quality. Technical debt incurred by the passage of time and the evolution of the surrounding environment is not the result of bad quality. Your system could have had the best possible design (or code) at the time you built it; five years later, it is deep in technical debt because of changes in your environment—not because the system has degraded. A technological gap has grown between the original state and the current environment. For example, perhaps you picked AngularJS as your front-end web application framework, but starting with the most recent release, the release documentation announces that AngularJS will not continue to support Internet Explorer. You ignore this version incompatibility for a number of releases, focusing on implementing other functionality, until you discover that the number of customers using Internet Explorer was not as small as you initially thought. You incurred debt just because time passed and you didn't revisit your initial choice, not because you took a shortcut in that initial choice.

Finally, some technical debt items are associated not with the code of the product but with the code of other closely related artifacts in software production: build scripts, test suites, or deployment infrastructure.

The constant characteristic across this landscape of technical debt is its invisibility. Technical debt is not visible outside the system's development organization; it is mostly invisible to customers, purchasers, and end users. These parties observe the systems. They are affected by a reduced ability of the development organization to evolve or maintain the software product and, in more dramatic cases, a degradation in the overall quality. In the financial world, you drive the BMW, and there is no observable evidence that you still owe 50% of it to your bank. In software development, end users use your software without knowing how much technical debt your organization owes on the product.

Some developers (and tool vendors and researchers) argue that defects—or any other form of visible low external quality—are technical debt. Some development teams even argue that unimplemented requirements are technical debt. We think that this makes technical debt too vast a category, rendering it a more or less useless concept. Defects and low external quality—such as poor performance, a cumbersome user interface, instability, and security holes—are not technical debt. They are just poor external quality; the system is not operating properly, and its problems must be addressed. However, the poor quality may be a consequence of technical debt. Software practitioners already know how to track and manage defects and unimplemented requirements. Technical debt refers to a class of issues not historically tracked or managed that represent the trade-offs among technical decisions and their changing consequences as the system grows.

As explained in later chapters, defects, new requirements, and technical debt must all be considered when planning upcoming work because all three compete for resources during software development. They all require the effort of the development organization and contribute in different ways to the value of the software product. But for now we limit our attention to what is inside the technical debt landscape.

Technical Debt Items: Artifacts, Causes, and Consequences

All software-intensive systems, regardless of their domain or size, suffer from some form of technical debt that negatively affects their evolution if it is not managed in a timely manner. This technical debt is not atomic or monolithic, but it can be decomposed into dozens or hundreds of items, which we call *technical debt items*, that accumulate over time.

A technical debt item is associated with the current state of a development artifact: a piece of code, build script, or test. It is a concrete development artifact that you can point to. What connects this item to technical debt is that the state of the artifact makes further changes to the software system more difficult: Evolving the software system is slower, more costly, more error-prone, riskier, or even impossible. Technical debt adds some kind of friction to further development, making it harder. In practical terms, how do you manage technical debt items? They can be mapped to entries in the same tool you use to manage your backlog, or they can be mapped to your issue tracker.

Technical debt is a state of your software system, and it has multiple causes and multiple consequences. Each technical debt item has one or more causes. The most likely cause we have observed is schedule pressure. A development team takes a path that is expedient now to save time and effort and, very often, because of some imminent delivery deadline. But there are other reasons to take on technical debt, as the stories in Chapter 1 illustrate. For example, you may want to investigate a possible product solution with minimal initial investment. Or the developer may not know a better approach at that point. While most technical debt can be traced to some decision made by the development organization, consciously or not, other causes are not linked to any decision made by developers or anyone from the business side. Some technical debt is caused by changes that occur outside the system, when some other element evolved in such a way that your system now suffers from technical debt; your software system has aged. We will look more closely at this "technological gap" in Chapters 6, "Technical Debt and Architecture," and 10, "What Causes Technical Debt?"

Be careful not to confuse technical debt with the cause of that technical debt. The necessity of meeting a hard deadline is not technical debt. But that necessity may lead you to make an expedient choice that changes the state of an artifact. Or you may miss a deadline because the current technical debt slows you down, and this is a consequence of technical debt.

The consequence of most technical debt is the additional costs that the development organization will potentially incur in the future. "Let us do <this> now, and we'll decide later if we can afford to do <it> better." Essentially, the debt is not borrowed money but borrowed time—or, more precisely, borrowed effort—which the organization can translate into monetary terms. These additional costs do not always appear clearly associated with specific technical debt items. Instead, they manifest in the form of reduced velocity (or productivity), longer release cycles, or even their effect on the development team's morale.

Technical debt, however, may have consequences other than costly future development; it may also manifest in more defects, by making the evolution of the system more error prone for other developers. For example, if the technical debt takes the form of missing documentation or code that is hard to read or overly complex, a developer might make changes to this code and introduce a bug inadvertently. In turn, this bug may have some impact on the value of the software and lead to future remediation costs.

Often, these consequences—unintended development, lower productivity, and system fragility—are first visible only to the development team. They are the externally visible symptoms of technical debt. By themselves, symptoms are not complete technical debt items, although some development teams and software managers mistakenly refer to them as technical debt. They require some deeper investigation to identify the actual state of the related development artifacts, as we will show in Chapters 4, "Recognizing Technical Debt," 5, "Technical Debt and the Source Code," 6, "Technical Debt and Architecture," and 7, "Technical Debt and Production." Using another analogy, if technical debt were a health issue, then stomach ache, coughing, or high body temperature would be the consequences, also called symptoms, while eating contaminated food or sitting next to someone who is ill on a packed six-hour flight would be possible causes. Relieving the symptoms, such as taking a fever reducer, often does not resolve the issue. To get the complete picture, it is necessary to determine the state of the lungs or the stomach to effectively diagnose the illness and treat the patient.

Principal and Interest

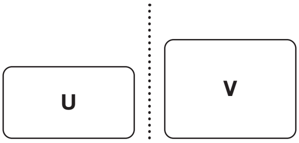
Using our metaphor of financial debt, such as the mortgage on a house, financial consequences can be articulated in terms of principal and interest. The principal associated with a technical debt item is proportional to the effort that a development team would expend to eliminate it. Similarly, the interest incurred by this technical debt item is the effort expended in additional development if the team leaves the technical debt in the system. Moreover, both the principal and the interest grow over time, as more development that depends on the related development artifact is done, which ultimately makes paying off the debt more and more expensive.

Let us illustrate these concepts with a simple example.

Step 1: Incurring Some Initial Debt

You need to implement a new feature, inventory management, in your system and the underlying software stack. You can choose one of two design strategies:

- **Design option U:** A home-brewed stack based on the MEAN stack (Mongo DB, Explore.js, Angular.js, Node.js), which is expedient but not quite extensible; it has a low cost, say, six person-days.
- **Design option V:** A commercial middleware product, which is a much better design and is extensible and elegant; it has a higher cost, at ten person-days.



Solution U is cheaper than V

Note

The size of the box indicates development cost: The larger the box, the more effort in its development.

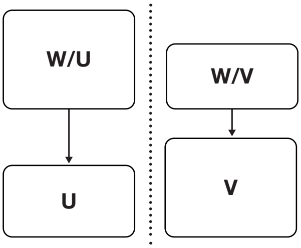
Because of schedule pressure, you choose the home-brewed stack, design option U, for the first release. Your home-brewed stack is not “buggy,” and your inventory management feature will work perfectly well in either option.

Step 2: Evolving the System and Facing the Debt

For the second release, you want to implement a new feature, an order entry function, which depends on inventory management and therefore on U. Let us call the implementation of this order entry feature W.

Implementing order entry, W, on top of the quick-and-dirty home-brewed stack, U, is costly: W/U. It is more costly than it would have been if you had chosen the middleware product option, V, in the first place: W/V.

So, the code associated with the home-brewed stack, U, has incurred some technical debt. The interest you pay on this technical debt is the additional effort it now takes you to implement the new feature on top of U, relative to implementing it on top of the middleware product, V.



W over V is cheaper than W over U

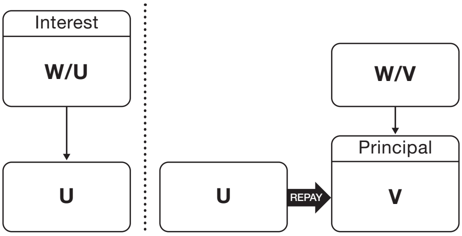
Step 3: Deciding How to Treat the Debt

You now have another choice:

- You can repay the debt by discarding U and implementing the commercial solution V. Since U cannot be refactored, this choice involves paying the full price for V: the original principal that was the cost savings plus the cost of replacing U. By choosing this option, you avoid incurring any interest on new features developed on it.
- Or you can decide to implement the order entry feature W on top of U, with interest.

Your choice will likely be driven by immediate considerations of cost and schedule pressure and how much of each is involved. The interest—the additional cost of implementing W on top of U—is small compared to the cost of replacing the homebrewed solution U with the commercial middleware solution V and is not offset by the lower cost of implementing new features, such as order entry W on top of V.

If replacing U with V costs ten person-days, but the difference between implementing W on top of U instead of V is only one person-day, you may be tempted to choose the cheaper option: Accept the interest and postpone the decision to change U into V to some later day.

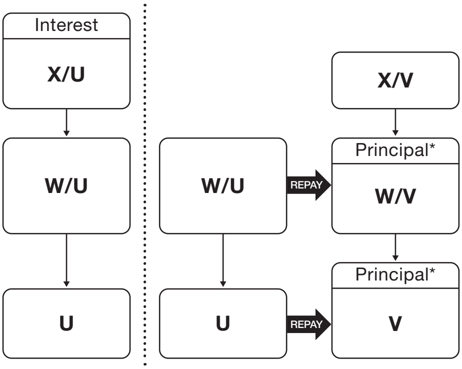


Pay interest, or repay the principal

Step 4: Just Paying Interest

If you do not repay the principal, your technical debt will continue to accrue as you add new features, implemented by X, as shown in the next figure. You have not added a new technical debt item, but you have made the current item more costly. At some later time, if you decide to replace the quick U with the nicer V, you will also have to retrofit W/U and X/U to make W/V and X/V; in other words, your implementation of each feature—W, X, and any others—on top of U would also have to be adjusted to the better V. The cost associated with moving from U to V therefore increases (principal* in the figure).

Interest is one of the key concepts of technical debt. In fact, understanding how and when interest accumulates in technical choices helps determine whether an issue should be managed as a technical debt item. Sometimes for good reason you choose to accept the interest. But often this choice is not conscious, and development costs increase. Managing such issues as technical debt items will increase their visibility so you can better assess the consequences and make informed decisions about the time and effort for treating the debt.



Pay more interest, or repay the higher principal

Cost and Value

Our description of technical debt so far has revolved around the costs, expressed as principal and interest. But like financial debt, technical debt has some value. You gain value in taking a mortgage to finance a home: You can enjoy the home now rather than wait until you are 60 years old to afford one. Similarly, software projects take on technical debt, consciously or not, because doing so creates some immediate value. And as in all other economic endeavors, there are trade-offs to be made. Cost must be constrained and value maximized.

Although we think of both value and cost in monetary terms—dollars, euros, or yen—they are different, and we will be careful not to confuse them.

Cost means any development costs: what it takes to get a system into the hands of its end users. For software-intensive systems, cost is primarily driven by compensating software developers. To estimate the cost of software developers, you must be able to estimate the amount of time they will spend developing. The cost comes in two forms:

- **Recurring interest:** The cost of the constant additional, possibly also growing, effort incurred because of the technical debt whenever the system must evolve.
- **Principal and accrued interest:** The cost of changing the design and the cost of retrofitting the dependent parts (the workarounds) in order to repay the debt.

A way to illustrate the difference between recurring interest and accruing interest is to look at a typical credit card statement:

Credit Card	Cost	Analogue in Software Development
Principal (at start of month)	$1,000.00	Current remediation cost
Interest for the month	$50.00	Accruing interest (coding a workaround)
Financial charge	$35.00	Recurring interest (slowdown of the team)
Balance due	$1,085.00	

The interest for the month corresponds to accruing interest; it is added to the principal as an obligation to pay in the future. The monthly financial charge of $35 corresponds to recurring interest; you pay it whenever your balance is not zero (a feature of credit cards in North America that you may not have experienced elsewhere!). Let's assume that you pay only this financial charge and do not repay the principal. The next statement will show the following balance:

Credit Card	Cost	Analogue in Software Development
Principal (at start of month)	$1,050.00	Remediation cost going up (accruing)
Interest for the month	$52.50	Retrofitting the workarounds
Financial charge	$35.00	Still slowing down general progress
Balance due	$1,137.50	

If you continue to defer payment, at the end of the year you will have paid $35 per month, for a total of $420 in financial charges, and the principal will have grown to $1,795.86 due to interest compounding monthly.

Where the analogy breaks a bit is that in software development, the interest is not a defined percentage of the principal. You may not accrue interest all the time; you may have only recurring interest. The principal may change for various reasons, and it may not be the principal incurred at the beginning. Other slight breaks in the analogy include the debt incurred by the passing of time that we described above, although you could think of it as not maintaining the cedar shingle roof of your house: It was perfect when you built it, but 15 years later, it has decayed and must be rebuilt.

Value is what the business draws as profit from selling the software, or the value perceived by the end users or the acquirers of the system. Value is even more difficult to estimate and forecast than cost. The accounting department can only tell you about the actual value of sales so far.

When you evaluate, assess, and decide what to do about technical debt, you must deal with forecasting. Forecasting is difficult because it requires comparing different scenarios about the future, with different values and costs associated with them. One method for getting past the question of whether you are including the right details in your estimation is to use some proxy for monetary value. Many software development processes and organizations use points for cost: function points, use-case points, story points, and similar terms. There is no well-established proxy for value.

Let us revisit our simple example, this time to look at both cost and value. Whether you choose the quick-and-dirty U or the more elaborate V in the first release, the value of delivering a feature implemented with W in the second release is the same. But using design option U is cheaper. Remember that technical debt is not an externally visible defect, so the end user has received the same value at this point, regardless of the cost.

In the next release, you add the new feature, X. The total value delivered is the same whether you use design option U or V. But again, the cost with V is less than the cost with U.

To optimize value for a given cost over time through multiple releases, you should have chosen the more complex V. However, there is more to value than what features are delivered. Value is also influenced by when they are delivered. On one hand, choosing V would have delayed the initial delivery of W, potentially reducing its market value. But the investment in V would have reduced the recurring interest in adding a new feature as the system evolves. On the other hand, the value of incurring technical debt by choosing U is the time you gain by delivering additional value early. This is a valuable use of debt—that is, an investment resulting from understanding the business trade-offs and actively managing the technology roadmap—if the software development endeavor is highly risky and if you are ready to walk away from the system, never implementing feature X.

The Installment Plan
by Ben Northrop

There are often many options for paying down financial debt, and this is the case for technical debt as well. Though we might like to pay off each loan (or technical debt item) in its entirety, given the constraints, risks, and context of the project, it's often more practical to pay it off in installments.

A few years back, I was working on a kiosk application for a food service client. The system supported a number of features, from browsing the menu to calculating nutrition facts, but the main function was to allow hungry and time-conscious customers to place their orders quickly. This efficiency component was crucial, so the overriding design philosophy was to keep the user interaction as quick and easy as possible—so the customer would need only tap and swipe, never type.

Well, philosophies are made to be broken. About a year into the project, it was determined that a little typing was needed after all. Power customers would like to be able to log in to the kiosk to see their favorite orders and receive personalized recommendations, and until fancy QR code readers could be installed for instant authentication (à la Starbucks), we would need to implement a simple email/password login screen. We were assured, however, that this feature was a one-time thing, and it would be the only typing interaction we'd need to support.

With marching orders in hand, off we went to implement a virtual, on-screen keyboard for our new login feature. And trusting in the assurance that this typing feature was an anomaly, we happily embedded the code for the virtual keyboard right into the login screen. It was quick and dirty but effective.

Of course, a few months later, things changed. Focus groups showed that customers who had not originally logged in might still want the ability to enter their email address after the order process, so, as we should have expected, we were now tasked with supporting a second instance of typing via the virtual keyboard. Had we initially implemented the keyboard generically (for example, abstracted it into some common widget), then reuse would have been a snap. Instead, our code was tightly bound into the login screen, and the solution would not be so easy. To further complicate the matter, we were in the final sprint of our release cycle, and capacity was stretched thin.

The decision before us was clear: Either we could go back and extract the virtual keyboard into a common component (as perhaps we should have done in the first place!), and incur the extra cost associated with the code refactoring and extra testing. Or we could take the easy road and simply copy and paste the original virtual keyboard code and embed it again into the feature code, this time the email entry screen. With the clock ticking, we took out a loan.

In the following months, our decision proved sound. We not only made our deadline but also found that the debt was manageable. Sure, there were a few minor defects or style enhancements in the virtual keyboard that required duplicate fixes, but overall the interest payments were low, and we were aware of the debt we would need to later pay (that is, it was on our backlog).

A few months later, however, a third requirement came for typing into our kiosk. At this point, we knew we did not want to increase the size of our debt by taking the copy/paste route again, but with a recently reduced team, we also knew we did not have the ability to pay down the entire principal in a single sprint. Was there a way we could avoid getting ourselves into more debt but also not pay off our loan in its entirety?

Our approach was to pay down our debt in installments. We recognized that the virtual keyboard was composed of three pieces: the style (CSS), the template (HTML), and the controller logic (JavaScript). Further, it was clear from the past few months of maintenance that the majority of changes were isolated to the look and feel and that the logic was generally stable.

Given all this, we decided to pay our debt in three chunks. In the first sprint, we extracted the CSS classes into our common stylesheet but left the duplicate template and controller logic in place. A sprint later, we tackled the template code, pulling it out into a reusable HTML snippet. In the third sprint, we abstracted a few of the nontrivial blocks of logic in the controllers into a common JavaScript library, and then we were done. In the end, this was not the purest design, and it was certainly not what we would have created if we had had perfect knowledge from the start, but it was pragmatic. In three smaller debt payments, we were able to consolidate about 80% of the solution, leaving only a very tiny debt of duplicated code that we were comfortable never paying off.

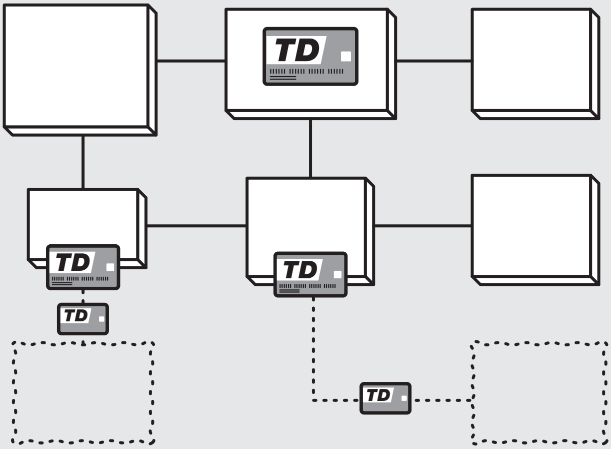
The point is that for any particular technical debt item, there is often a spectrum of options between the obvious poles of "do nothing" and "pay it all off." For us, though, either would have been a less-than-perfect end solution, and paying a technical debt in installments was a practical way to meet the demands of the business and also keep our technical debt under control.

Potential Debt versus Actual Debt

Not all technical debt items have the same impact. Whether an issue is actual technical debt or not depends on what future evolution a development team wants to make to the software system. If a technical debt item resides in a part of the code that is not affected by a future evolution, then this technical debt item is only potential debt. When it affects the evolution, it becomes actual debt.

The financial analogy again helps make this relationship to time and evolution concrete. If you borrow money from a bank with 0% interest initially, it is like free money; it is not incurring additional cost to you. Likely you will plan to repay it before the interest rate switches on. But until then, you may use the borrowed money for other purposes. You have the debt, but you have not started seeing its impact. This brings us to our second principle.

Principle 2: If You Do Not Incur Any Form of Interest, Then You Probably Do Not Have Actual Technical Debt



We use this principle as a litmus test when we classify an issue as actual technical debt or not. This gets tricky as the business and the technical context of your system changes; the likelihood of interest accumulating on design choices may change as well. For example, an area of the system with an abundance of problems may currently have zero interest if it is sufficiently decoupled from the rest of the system and does not need to be maintained. Understanding interest and how it changes is key to understanding and managing your technical debt. The amount of interest is not linear and fluctuates as systems evolve. Hence managing technical debt is not a one-time activity.

A system may have a very large potential debt, but at a given point in time, based on the next evolution increment, only a small part of this technical debt is actual debt. This distinction will drive our prioritization of technical debt remediation (or repayment). We focus mostly on actual debt first, and next on potential debt, depending on its likelihood of occurrence.

The Technical Debt Timeline

As you have realized by now from our description of technical debt, time plays a big role: Technical debt matters only as time flows and the software system evolves. If the system were to never evolve, you would never have to pay any interest, and therefore technical debt would not matter. Let us look at the technical debt timeline, which shows how technical debt unfolds over time (see Figure 2.2).

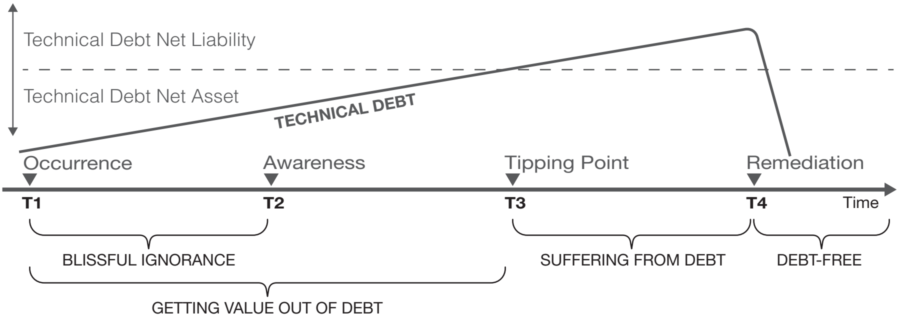


Figure 2.2 *Technical debt timeline*

T1: Occurrence

Occurrence is the point in time when a technical debt item is introduced into the system, for whatever good or bad reasons.

T2: Awareness

Awareness is when the development organization sees the symptoms of the technical debt item. For technical debt that was incurred intentionally, through a deliberate decision and for some clear immediate benefit, T2 = T1. A development team made an explicit decision and records it. But for many development projects, lots of technical debt items just happen, unintentionally, and they will be discovered only later, when symptoms of slow development or defects point to strange workarounds, “fix me” comments, or “todo” labels in the code. The period between T1 and T2 is blissful ignorance.

T3: Tipping Point

The tipping point is the time when the cost of having technical debt starts to overcome the original value of incurring the debt. In the interval from T1 to T3, the slower progress due to recurring interest and the accruing interest of not-quite-right code that will need to be retrofitted lead to a situation where you could be better off repaying the debt. Before T3, you might just as well live with it; you actually get some value from the technical debt. T3 is an inflection point: you now pay more than you gain.

T4: Remediation

Remediation is removing the technical debt item from the system. The cost of remediation includes the initial principal and all the accrued interest. In the period from T3 to T4, the debt continues to accumulate interest. But also (unlike in the financial world), the principal may evolve to be very different from the initial principal. So, the remediation will often be more effort than just undoing the not-quite-right code and implementing what would have been the right solution at T1. The remediation might lead to a very different design from the one you forfeited at T1 because the context has evolved significantly. After T4, you stop paying the recurring interest, too.

…Or No Remediation

The remediation step at T4 may not be chosen. If the tipping point T3 is far in the future, the remediation costs may be prohibitive, so you postpone the decision further into the future. Therefore, there may be a period from T2 onward during which you just live with the technical debt and accept to pay any recurring interest as you go. This is likely to be the case when the technical debt item is confined to a part of the source code that is unlikely to evolve in the future, so it will have very minimal recurring interest and no accruing interest.

Technical debt-savvy organizations may not wait to hit the tipping point at T3 and instead start remediation early.

In the following chapters, we will tackle key planning questions:

- Is incurring debt worthwhile?
- Once you have technical debt, when should you repay it?
- If you cannot afford to repay it all, which parts of it should you prioritize?

Software development organizations usually operate within budgets that constrain the costs. They try to optimize the delivered value at each step or release, and when they decide what to do at each step, they face competing demands on their budget, including adding new features, scaling up the system, increasing quality (in particular by reducing defects), and reducing technical debt. They need to estimate both cost and value for these competing demands.

What Can You Do Today?

The processes of developing, evolving, and sustaining software systems require making technical, organizational, social, and business trade-offs. The more these trade-offs are made explicitly and communicated broadly, the more likely resources will be allocated strategically. Start today by understanding the rich vocabulary of technical debt and socializing the concept with the key stakeholders of the system:

- Provide a clear, simple definition of technical debt in the context of your project.
- Educate the team about technical debt.
- Educate the people in the immediate project environment about technical debt: management, analysts, product managers.
- Create a "techdebt" category in your issue tracking system, distinct from defects or new features.
- Include known technical debt as part of your long-term technology roadmaps.
- Extend awareness activities to external contractors if they are part of the project.

As you learn more about different technical debt management principles and practices in the following chapters, you will fill a toolbox that will equip you to deal with conversations about trade-offs and managing your technical debt strategically rather than being overrun by it.

For Further Reading

Steve McConnell (2007) of Construx was the first to establish a classification (or taxonomy) of technical debt, differentiating small, scattered, and mostly unintentional debt from large, intentional, and strategic debt. Martin Fowler (2003) of ThoughtWorks presented a different twist on the taxonomy, highlighting the "prudent but inadvertent debt" caused over time by an evolving environment.

Steve Freeman and Chris Matt (2014) have argued that traditional financial debt is not the best metaphor; software technical debt is more like an unhedged call option in the derivatives financial markets. The buyer pays a premium to decide later if he or she wants to buy. The seller collects the premium and will have to sell if the buyer decides to buy. It is not predictable for the seller. When you incur the technical debt, you collect the premium: You immediately get benefit from it (shorted time). But as soon as you have to maintain or evolve this codebase, the option is called, and you have to pay an unpredictable amount of effort to achieve your new objectives.

The technical debt landscape emerged as the result of a workshop held at the International Conference on Software Engineering (ICSE) in Zürich in 2012 (Kruchten et al. 2012). The principles of technical debt came out of the workshop at ICSE 2013 in San Francisco (Kruchten et al. 2013).

Chapter 3

Moons of Saturn—The Crucial Role of Context

In this chapter, we introduce three case studies that we'll use throughout the book to illustrate the main concepts of technical debt and the strategies for managing it. All long-lived software-intensive systems have to deal with technical debt within their context. The interactions and specifics of the many factors of context help development organizations understand the systems and navigate the causes and consequences of the debt.

"It Depends…"

When asking questions about software development practices, how often have you heard the reply, "it depends"? This is not just a way to dismiss the question. There are no all-inclusive answers, universally applicable techniques, or standard recipes. The answer really does depend on a number of factors that describe the context of the system. Eight of these factors are shown in Figure 3.1.

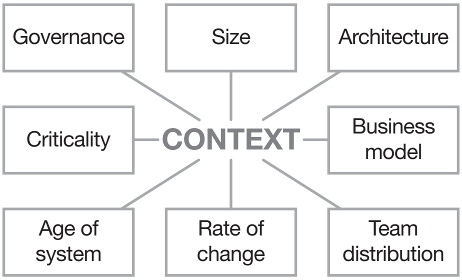


Figure 3.1 *"It depends": The many factors of context*

- **Size:** The size of the system is by far the greatest factor because it drives the size of the team, the number of teams, the need for communication and coordination between teams, the impact of change, and more. The number of person-months, the size of the code, and the development budget are all possible proxies for size. Size is often related to complexity. The larger the system, the more technical debt it can accumulate.
- **Architecture:** Is there a de facto architecture in place at the start of the project? Most projects are not novel enough to require a lot of architectural effort. They follow commonly accepted patterns in their domains. Many key architectural decisions are made in the first few days of development, such as choices related to middleware, operating systems, and programming languages. These choices may be based on what the developers are familiar with and their gut feelings rather than a careful analysis of long-term system consequences. We will show in Chapter 6, "Technical Debt and Architecture," that technical debt at the architectural level is difficult to identify and very costly to repay.
- **Business model:** What is the money flow? How is the project funded? Are you developing an internal system, a commercial product, a bespoke system on contract for a customer, or a component of a large system involving many different parties? Is it free/libre open-source software (FLOSS)? Financial considerations are a key factor in incurring technical debt or deciding to remediate technical debt.
- **Team distribution:** Team distribution is often linked to the size of a project. How many teams are involved and collocated? Distributed teams increase the need for explicit communication and coordination of decisions as well as stable interfaces between the software components that they are responsible for. Communication issues and organizational silos contribute to the accumulation of technical debt, especially at the architectural level.
- **Rate of change:** Though agile methods are all for embracing change, not all systems experience a rapid pace of change in their environment. Many projects have very stable requirements definitions. How stable is your business environment, and how many risks and unknowns are you facing? The volatility of the requirements will increase the propensity of the team to incur technical debt.
- **Age of system:** Technical debt has more opportunities to accrue on large and long-lived systems. These legacy systems carry hidden assumptions about their architecture, and evolving them can reveal technical debt. Constraints accrue in legacy systems, often causing another source of technical debt. Alternatively, creating a new system, with fewer constraints, can proceed without taking on a lot of debt.

- **Criticality:** How many people die or are hurt if the system fails? For safety-critical and mission-critical systems, documentation needs increase dramatically to satisfy external agencies that want to assure the safety of the public. More formal verification and validation techniques may be essential to ensure that a system behaves the way it should. Such systems often struggle with how to modernize hardware or software that can be a major source of debt—whether it is legacy hardware or some arcane software that implements a critical algorithm.
- **Governance:** How are critical decisions made? How are projects steered? How do projects begin and end? Who decides what to do when things go wrong? How is success or failure defined? Who manages the software project managers? Tension or lack of communication between a project and the management structure may cause technical debt accumulation, as discussed in Chapter 10, "What Causes Technical Debt?"

Other factors can change the context of the software development process, but they have more indirect effects on it. They mostly shape the eight factors just described. Some of these other factors are domain, process maturity, corporate culture, degree of innovation, and economic imperatives.

These factors combine in many different ways to create the context in which development organizations must plan their approach to technical debt. An old and large company might have mostly large projects, a significant level of governance, proprietary code, a stable architecture, large globally distributed teams, and a medium rate of change. A small startup might have a small codebase, an unstable or still fluid architecture, low criticality, a high rate of change, and a collocated team.

Three Case Studies: Moons of Saturn

We now introduce three example projects, laden with different types of technical debt and facing different kinds of tactical choices. We will use the context factors to describe these projects and the systems in development so you can quickly understand the environment, system characteristics, and whether they are similar to your own. We derived these examples from actual companies that we authors have interacted with, but we abstracted many characteristics and details for confidentiality reasons, and in some cases we combined characteristics from two similar organizations into a single example.

These examples feature three different companies, developing different kinds of software-intensive products in three different domains. We named the three projects after three moons of the planet Saturn. Their size variation represents the sizes of the three companies:

- Atlas (diameter: 30 km)
- Phoebe (diameter: 213 km)
- Tethys (diameter: 1,062 km)

An easy way to differentiate the projects is to remember that the sizes of the moons grow in alphabetical order: Atlas is smaller than Phoebe, which is smaller than Tethys.

Table 3.1 summarizes the key differences among the three software products and the respective companies in terms of the eight main factors and two others, describing domain and process.

Case Study 1: Atlas—The Small Startup

Atlas is a small startup company, barely three years old, whose original founders act as the senior management. Atlas has a single product in the e-commerce space.

The Atlas development team is collocated and has grown from 4 developers (the founders) to about 15 within two and a half years. They use an ad hoc agile process, neither formalized nor rigorously followed, but they do speak to each other daily, and all use a very well-defined tool set that allows them to quickly deploy new features to customers. They are very focused on their market and tactically "pivot," a term used to denote a change in product direction that drives a corresponding change in the software product specification. There is no clear role specialization in the team, and everyone contributes to all aspects of development, including requirements, design, coding, and testing.

The Atlas design has no deliberate or explicit architecture. It has no formal documentation: The developers say that "the code is the doc." Atlas uses an almost continuous delivery for its installed base, but for the wider audience using the open-source part of the system, it has a slower rhythm for releases of about three weeks. However, it has limited regression-testing capabilities. The codebase in Java and JavaScript, with some C, is now about 400,000 source lines of code (400 KSLOC).

The key business driver for Atlas is finding its niche and carving out its piece of the market. The development team added some features to the product in the open-source version to help Atlas attract new business for the full-blown product and develop a friendlier image. The company is in a domain with no external regulation or governance pressure.

Table 3.1 *Contrasting the three case studies*

Factor	Atlas: Small startup	Phoebe: Agile shop	Tethys: Global giant
Domain	E-commerce	Healthcare IT	Transportation
Size	400 KSLOC*	2 MSLOC	4 MSLOC
Architecture	Data analytics, usability, evolvability, cloud, MEAN stack (MongoDB, Explorer.js, Angular.js, Node.js), big data	Security, privacy, scalability, service-oriented architecture (SOA), cloud, large databases	Safety (reliability, high availability, fault tolerance), performance, multiple designs, hardware dependent, real-time embedded
Business model	Market-driven pivots in service to online user base	Open-source software of the partner organizations for business growth	Main contractor for an external customer
Team distribution	Single collocated team, fluid organization	Core team and a few dispersed teams in a single country	Multiple teams (>10), strictly defined roles, globally dispersed
Rate of change	Days to weeks	Months	Years
Age of system	Starting, active development	5 years, modifications for new markets	Over 15 years, in maintenance
Criticality	No	Moderate	High
Governance	Minimal: internal	Moderate: external regulatory compliance	High: multiple external standards, regulatory compliance, certifications
Process	Ad hoc agile with DevOps, rush to customers, multiple betas	Agile using Scrum, involved product owner	Hybrid, iterative, formal documentation and quality assurance

**KSLOC, thousand lines of code; MSLOC, million lines of code.*

As a result of constant pivoting, Atlas has accrued a moderate amount of technical debt, mostly under pressure to deliver the next prototype to the next key reference customer. The product suffers from scalability and evolvability issues, but the codebase has remained relatively clean. The development team has only limited regression-testing capability, and team members are wary of major refactorings.

The current level of technical debt in the codebase is becoming a source of tension between team members. Some developers are pushing to rebuild the product from scratch, which is a huge risk, as it would not allow any externally visible progress for six to eight weeks, and the senior management team is pushing back.

Case Study 2: Phoebe—Agile Shop with a Viable Product

The Phoebe team is developing an open-source software solution that supports health information exchange at the national level. The product has grown from meeting an initial small-scale need to attracting many organizations that would like to set up health information exchanges. The product has been in development and use for about six years, and it has been evolving with participation from both government and private-sector users as well as contributions from developers. Phoebe derives its revenue from selling services, not application or source code.

The core Phoebe development team is collocated, but a small number of developers in partner organizations also develop functionality and contribute to the backlog for their most pressing user stories. The core team size has fluctuated from 35 to 8, decreasing over the years. In addition, at times multiple subcontractor teams have developed different features of Phoebe. The core team has consistently used Scrum to manage iterations and followed agile software development practices.

The Phoebe design has evolved over the years to get ahead in a competitive domain dominated by critical quality concerns such as security and privacy. In addition, the development team must ensure that the product complies with a number of IT standards related to privacy and healthcare data. Phoebe is developed with a service-oriented software paradigm, and now the organization is investigating migrating some of its services to the cloud. To foster open contribution and enable new organizations to adopt the product, the development team has accumulated a substantial amount of online documentation on the architecture, design, open issues, and codebase as well as user documentation for deployment, installation, and use. These documents are open access and at times get out of sync due to different priorities of the core team.

The key business driver for the Phoebe product is to provide a reliable, safe, and efficient infrastructure for addressing the challenges of the growing health information exchange. There are many competitors from the private sector, but by embracing an open-source model, the product owner aims to increase contribution to development as well as product quality and use.

In a domain that is not only competitive but also watched by many eyes in the nation, Team Phoebe struggles to manage multiple stakeholders with diverse requirements, get ahead of changing technology, and sustain a viable product. As a result, technical debt accrues, in most cases intentionally. While Team Phoebe has been trying to repay that debt by prioritizing technical debt reduction in major releases, technology lock-in has become a major hindrance to meeting this goal. The development team keeps track of technical debt items, which are managed with other items of the backlog, tagged as "techdebt." However, members of the core team do not have a consistent process for identifying and managing technical debt. For example, the team tried using some tools to look into code quality, but it did not sustain their use. Major refactoring releases have eliminated some of the existing technical debt or made it obsolete, but Phoebe has not communicated this broadly to its stakeholders, and it is not clear how the team determines which issues are most important.

Case Study 3: Tethys—The Global Giant

Tethys is a large, global, multi-business organization. The Tethys product is 15 years old. It is safety-critical embedded avionics software, developed as a product line. The product team needs to balance many concerns of an evolving legacy product-line system that has been in existence for over a decade: large customer-installed base, new markets to open, changes in underlying technology, and the like. There is constant pressure to stay on top of competitive innovation with increasing demand from customers to include features. As a result, Team Tethys must, on one hand, define a new rhythm of agility in a complicated context and, on the other hand, pay due diligence to tough quality attribute requirements such as safety criticality, reliability, and security.

The Tethys product is developed by multiple development teams, and at times there are more than 100 developers on task. Project management must coordinate across system engineers, quality assurance teams, and compliance teams, both internal and external to the organization. Team Tethys also works with contractors extensively, which introduces another level of complexity to development.

As is typical with such systems, Tethys evolves through major planned upgrade releases to meet business goals. The longevity of the product and the different families of products in the product line are sources of major revenue for the organization. As a result, the upgrades often prioritize new features over needed re-architecting. The complexity of the deployment makes it impossible to have more than one major release per year and some minor releases for emergency bug fixes.

Such a long history comes with a lot of technical debt, which includes both architectural issues and code quality concerns as a result of developer turnover and inconsistent subcontractor practices. While code quality issues are not ideal, they do not block day-to-day development. Tethys suffers the most technical debt due to its architecture. Needed re-architecting efforts have not occurred in a timely manner, technology has changed but the product has not, each contractor has introduced his or her own interpretation of the structure, and the list goes on. Everyone on the team, from the most junior developer to the most senior manager, is aware of this debt, although not everyone understands the gory details or the extent of it. Yet it is hard to motivate the team to allocate the time and funding to tackle the debt because no one knows how to gracefully reduce it while keeping the business rolling.

Case Study Comparison

Table 3.2 summarizes the technical debt issues the three projects are facing and how they are managed, if at all.

There is not one universal prescription for managing technical debt that would work for all three projects. The contextual factors color not only the specifics of each organization's technical debt but also the way it can be managed.

Technical Debt in Context

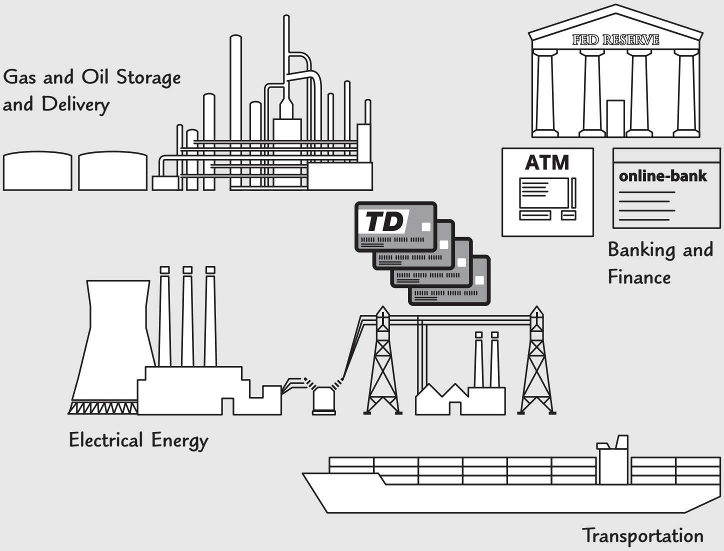
The specific context factors and their interactions will help you understand your system and navigate the causes and consequences of its debt. The bottom line

Table 3.2 *Technical debt issues addressed by the three case studies*

	Atlas: Small startup	**Phoebe: Agile shop**	**Tethys: Global giant**
Technical debt issues	Lack of scalability, lack of regression testing, using code as documentation	Locked-in architectural choices that have proved limiting	Mismatched assumptions between teams, high turnover, internal code quality, aging system lagging in technology
Technical debt awareness and management	Awareness of technical debt late in the timeline, conflicting priorities in addressing it	Identification of technical debt, regular focused debt reduction, incomplete consideration of all aspects	Technical debt as the elephant in the room

is that all organizations with long-lived software-intensive systems have to deal with technical debt within their context. We cannot emphasize enough the importance of understanding this as it is a critical first step in successfully managing technical debt.

Principle 3: All Systems Have Technical Debt



Only the most trivial systems would not have some form of technical debt because they do not evolve much over time. Other systems, such as safety-critical systems, may have technical debt that is more visible, especially relative to the aspects that could impact safety due to increased scrutiny.

As we progress through the book, we will look at how the three different organizations described here use various techniques to improve how they handle their technical debt.

All Opportunities Come with Risks
by Linda Northrop

We make life decisions—pick a college, choose a profession, take a job, vie for a promotion, select a partner, buy a house, have children. In each decision there are inherent opportunities that can provide fulfillment and growth. There are also inherent risks. Risks are problems that haven't happened yet. The risks in life decisions may materialize into problems that cause varying degrees of dissatisfaction, frustration, and worse.

So it is with software. Let me share some experiences.

Consider the decision made by countless designers and programmers in the 1970s to handle dates by storing a year value in a two-character string. Why would they have done that when a year is four digits? Memory at the time was at a premium, and every opportunity for memory conservation was important, especially for something as ubiquitous as the value of year. It worked. It was awesome…until 20-plus years later, when the glut of these systems taken together created a major problem with potentially disastrous consequences and global, vast technical debt. In the years leading up to 2000, what I just described was dubbed the "Y2K problem." This is personal. I designed and coded some of those systems. Even worse, I programmed some in PL/I, in which it was possible to overlay different kinds of storage—and I did, on the year field! Why did I do this? It was a great opportunity to save storage, and the probability of the risk I took becoming problematic was miniscule. I just *never* imagined anyone would be using these systems 10 years later, let alone 20. I was little concerned that my systems had technical debt that had to be repaid before January 1, 2000. Thanks goodness, it was.

Here is another bit more recent example. Beginning in 1994, a U.S. Army tactical command-and-control system, called Force XXI Battle Command Brigade and Below (FBCB2), was designed as a hardware/software prototype demonstration system for on-the-move operations (think tanks, Humvees, helicopters, forward operating bases) that would revolutionize situational awareness capabilities. For those of you without military background, situational awareness means knowing the answers to these basic questions: Where am I? Where are my buddies? Where is the enemy? What is the environment?

FBCB2 was to pioneer (among other innovations) the use of GPS receivers, a tactical Internet, and local computer displays with human interaction

(Bergey et al. 2005). In doing so, the designers and developers had an opportunity to provide unprecedented, sophisticated capability to the warfighters (who were still relying on physical maps): They made software decisions that prioritized functionality—proving this new capability. That strategy was successful. FBCB2 was used by U.S. forces in the Balkans, Afghanistan, and Iraq. It was the U.S. Army's most successful entrée into battlefield digitalization and, most importantly, it saved lives.

Not surprisingly, there were also risks in the architectural decisions that prioritized functionality. Modifiability, scalability, interoperability, and extensibility were poorly supported. As FBCB2 enjoyed widespread acceptance and accolades, the technical debt associated with the architectural decisions became problematic. Modifications, new configurations, and maintenance proved difficult and costly. The system needed to be re-architected, and it was. In my opinion, the opportunity to field a less robust system that saved lives and yet risked downstream evolution and sustainment problems was worth the risk and the technical debt. Again, thank goodness, the debt was repaid.

More recently, a colleague shared that his software development organization chose AngularJS—a great opportunity to take advantage of a powerful front-end web application framework that was widely used, supported, and interoperable. There was a proprietary framework layered on top of AngularJS and hundreds of internal applications using this stack. AngularJS not only provided functionality but standardization across the underlying applications. There was little risk as far as anyone could see…until Angular 2 (now called simply Angular) was released to replace AngularJS. Angular is considerably different from its predecessor in language (now TypeScript) and features. The result was considerable technical debt to migrate both the proprietary framework and associated applications to Angular. The changes to upgrade just the underlying proprietary framework were estimated to take one year, and until it was ready, the applications were to continue writing to the older AngularJS. Some of the applications chose a more expeditious route, going rogue and redeveloping to use Angular directly. The standardization across applications is now lost. Still, the opportunity afforded by AngularJS and Angular (at least in my opinion) is worth the risk. The coupling at the root of the technical debt might have been reduced from the outset, perhaps making it possible to preserve the standardization along with the functional advantage.

(continued)

There are many other examples I could share. Although I have no scientific evidence to substantiate my view, I have been at both life and software development for more decades than I would like to claim. What I do claim (and I don't think I am unique) is that it is wise to seize opportunities in life and in software development, mindful that there will always be risks and, in software, technical debt. This book is not about avoiding opportunities. Rather, it is about being cognizant of technical risks (as much as possible) and smartly managing the fallout should they become problems. All three of my examples could have benefited from these insights and approaches.

What Can You Do Today?

Identify the factors of context in your project that can create conditions for technical debt buildup. It is also important to use your knowledge of the context to gain insight into how specific practices for managing technical debt apply in your particular situation.

For Further Reading

The context of software development explained in this chapter is based on previously published work (Kruchten 2013). It is similar to the "agility at scale" model of Scott Ambler (2011).

The Atlas, Phoebe, and Tethys projects that we use as examples throughout this book are based on our experiences. There are other case study examples in the literature that may match your software context. Guo and colleagues (2016) describe a Brazilian software company that provides enterprise-level software development, consulting, and training services. They explain the impact of technical debt on a Java-based, database-driven web application for water vessel management. Ampatzoglou's team (2016) explores technical debt in seven embedded software systems. Klotins' team (2018) reports on how technical debt accumulates in a startup context using studies from 86 startups. And Sculley and colleagues (2015) reflect on their experiences developing industry-scale machine-learning systems and summarize the seven different categories of debt that they observe.

Part II

Analyzing Technical Debt

This page intentionally left blank

Chapter 4

Recognizing Technical Debt

In this chapter, we describe the causal chain of technical debt: causes and consequences. We expand on the concept of a technical debt item as a simple mechanism to identify and record the technical debt in a system. Then we explain how a software evolution strategy provides a starting point for analyzing the costs associated with technical debt.

Where Does It Hurt?

On any project that has run for a while, development teams might begin to observe signs that trouble is brewing, that something is not quite right or not working as well as it was before. The system becomes prone to certain types of defects, more bugs, or more crashes. Customers make more change requests, and it takes developers longer to satisfy them. Some customers even walk away from the system in frustration. Project managers are amazed by the estimated effort to implement what at first looks like a small improvement. These are *consequences* of technical debt, but they represent only the emerged part of the iceberg; they are *symptoms* of a more daunting condition in the system.

In Chapters 1, "Friction in Software Development," and 2, "What Is Technical Debt?" we characterized actual technical debt as mostly *invisible*, except to the developers. But technical debt has consequences—sometimes even a chain of consequences—some of which transpire outside the system. Some of these consequences may be visible as symptoms of the underlying technical debt.

This chain of causes and effects looks like this:

Causes → **Technical debt** → Consequences → *Symptoms*

Let's look at this chain in more detail with a simple example, based on a story introduced in Chapter 1, about a Canadian company that first developed a product for customers who speak English and then needed to make the product multilingual. This company is Atlas, the small startup, one of the three representative examples introduced in Chapter 3, "Moons of Saturn—The Crucial Role of Context."

Early in its existence, Atlas produced a demo version of its product almost overnight to show to a group of venture capital investors. A rudimentary scaffolding, called L10N (localization) and I18N (internationalization), was used in the code in lieu of proper localization and internationalization software. To appeal to another part of the Canadian population, the developers next wrote ugly code to support one other language, French, in addition to English. A few weeks later, the CEO of Atlas assured prospective Japanese customers that a Japanese version could be completed just as quickly as the French version. In fact, adding this third language proved extremely cumbersome. It required major changes in the way the code was developed. It also meant removing and redoing all the changes made to accommodate French. And it took quite some time to achieve and necessitated putting other developments on hold.

The chain of cause and effect in this case looks like this:

- **Causes:** Developers completed the first version in time for the demo under schedule pressure. They were also unfamiliar with I18N and L10N software.
- **Technical debt:** Code snippets to handle a second Latin-alphabet language were scattered all over the codebase because internationalization had not been considered as a key architectural driver and had to be retrofitted.
- **Consequences:** The code was error prone and could not support other languages, especially non-Latin-alphabet languages.
- **Symptom:** The visible consequence was a long delay to add support for a third (non-Latin-alphabet) language, when the team finally recognized the impact of the issue.

But the news was not all bad. Another consequence was that the investors were impressed by the bilingual version, and they moved forward with funding:

- **Consequence:** The company got a third round of investment from a venture capital firm (yeah!).

If members of the Atlas development team had been aware of the technical debt as it was incurred, they would have identified the risk early and could have made some contingency plans to deal with it. It is likely the schedule would not have given them the flexibility to accommodate French debt free, but they could have begun proactively managing the debt while supporting future languages. This would have helped set expectations for a longer release when negotiating resources for the Japanese version. More often than not, technical debt is unintentional and does not become visible until much later, when consequences surface.

The first step toward recognizing technical debt is to investigate the chain of cause and effect in reverse:

Symptoms → Consequences → **Technical debt** → Causes

Using the analogy of a health issue, a physician would start from the symptoms to diagnose the problem. Similarly, you should aim to detect more consequences of technical debt, possibly less visible ones, by looking inside the system, and they will eventually point you to the development artifacts containing the debt. We call these artifacts and their associated principal and interest a *technical debt item* (refer to Chapter 2). Identifying these items aids in resolving the problem at its source rather than treating the symptoms and then seeing the problems resurface.

Pursuing this path of analysis, you next ask, "Why do we have this debt item?" The answers to this question will help you locate the *causes* of technical debt—even its root cause. While understanding causes isn't strictly necessary for resolving the technical debt, it may provide insight into how the development environment is creating conditions for incurring technical debt. It may lead to changes in the organization to avoid generating more technical debt. We will explore causes of technical debt in more depth in Chapter 10, "What Causes Technical Debt?"

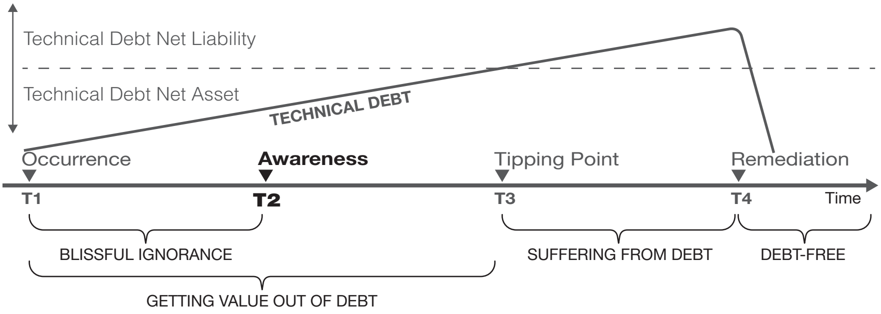


Figure 4.1 *Reaching the awareness point*

In the technical debt timeline we introduced in Chapter 2, the first goal is to reach the point of *awareness*, or knowing what technical debt you have in your system (see Figure 4.1). The technical debt item will enable you to track the debt you become aware of within your software development process so that you can estimate, discuss, and prioritize actions to take.

What Are the Visible Consequences of Technical Debt?

Some symptoms—such as the release delay faced by Team Atlas for its Japanese-language version—emerge after the entire system has been affected. They surface late in the development cycle and manifest as increased testing time, problems integrating subsystems into the rest of the system, and projects hitting major impediments that stop the release of new features. Other symptoms surface later still, during maintenance, and are reflected in increased maintainability and sustainment costs.

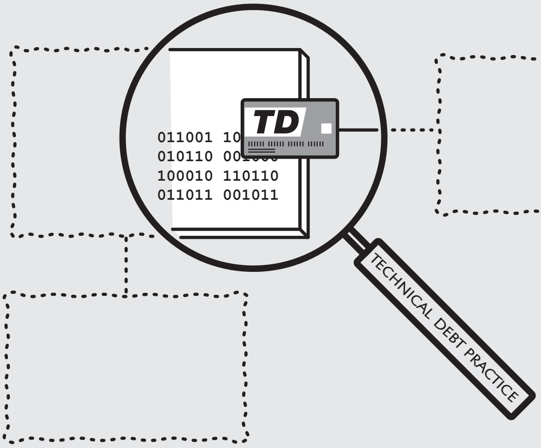
These symptoms are consequences of technical debt that manifest directly in the system. But consequences can reach further into the environment of which the system is a part. These consequences include an overall decline in quality that is visible to the end users and results in an increase in customer change requests or a decrease in market share as usage declines. When the consequences of debt are visible, they become easier for development teams to communicate to decision makers. Visible consequences also make it easier to get management buy-in for fixes, as Joe, a developer from Tethys, the global giant introduced in Chapter 3, summarizes:

> I think that it is fairly easy to convince management when performance is really bad, when they are experiencing latency, when systems stop working, and when they see exceptions on the user interface.

But it is also a risk because by the time consequences become visible, the debt may also have a higher cost of remediation.

Some symptoms of technical debt surface earlier during software development, before they affect the entire system. These include an increasing number of issues and bugs, a decreasing rate of development productivity (for example, velocity) or cumulative flow, and increasing code quality concerns (for example, cyclicity, McCabe complexity). Development teams often are aware of these symptoms, even the debt itself, but do not have the mechanisms or incentives to communicate the issue. This is where the technical debt item can help.

Principle 4: Technical Debt Must Trace to the System



To reason about technical debt, estimate its magnitude, and offer information on which to base decisions, you must be able to anchor technical debt to explicit technical debt items that identify parts of the system: code, design, test cases, or other artifacts. A development organization also needs to recognize other forms of friction related to processes, people, and the development infrastructure. But these sources of friction are causes of technical debt; they are not the debt themselves.

When you trace technical debt to the system, start with your business context, assess artifacts across the technical debt landscape, and record the results as a technical debt description.

Writing a Technical Debt Description

A technical debt description captures *where* in the system the debt is located (the concrete system artifact) and the associated state of consequences that it causes in the system.

Recall that Atlas's small startup team has just released the second version of its product, with the addition of French to a system that already supports English. The project team is now contemplating adding support for a third language.

A user story to describe the new feature request might take this form:

> As a <stakeholder>, I want to <action with system> so that <benefit>.

For Atlas, this looks as follows:

> As the Atlas Company, we want a Japanese-language version of our product so that we can increase market share and profit.

However, you need more than a user story to describe a technical debt item. You need to enhance the basic story by documenting some of the who, what, when, where, and why (also known as the *five Ws*, or *5W*) to describe the problem so that you can make it visible to the entire project team and deal with it as you would any other issue on the backlog. A technical debt description is a user story that includes the five Ws that explain the associated technical debt.

Here is the 5W version of the Atlas team's technical debt description:

> As a developer (**who**), I need to pay down the debt where internationalization (**what**) is scattered over the code (**where**). The accumulating cost to add support for additional languages will soon outweigh the initial benefit of implementing the ad hoc solution of if-then-else statements for the first two releases to obtain another round of funding (**when**). There will be a long delay to support the next language, and the code will soon no longer support additional languages, especially languages using non-Latin characters (**why**).

You will need to collect your technical debt descriptions in what we call the *technical debt registry*, or *registry* for short. But you can use the same repository and tool that you are already using to manage work—your backlog.

Table 4.1 lists the essential fields to capture a technical debt item. They can easily be incorporated into your issue tracking process and technical debt registry.

Typically, to track technical debt, software development teams use whatever tool they routinely use to manage the project, such as an issue tracker system or a defect database. Most issue trackers include capabilities to create custom types and fields. We strongly recommend creating a type for technical debt items and tagging technical debt descriptions with a label, such as "techdebt," if they are stored with user stories, defects, and other tasks.

Table 4.1 *Technical debt description*

Name	**What** is it? This field is a shorthand name for the technical debt item.
Summary	**Where** do you observe the technical debt in the affected development artifacts, and where do you expect it to accumulate?
Consequences	**Why** is it important to address this technical debt item? Consequences include immediate benefits and costs as well as those that accumulate later, such as additional rework and testing costs as the issue stays in the system and costs due to reduced productivity, induced defects, or loss of quality incurred by building software that depends on an element of technical debt.
Remediation approach	Describe the rework needed to eliminate the debt, if any. **When** should the remediation occur to reduce or eliminate the consequences?
Reporter/assignee	**Who** is responsible for servicing the debt? Assign a person or team. While in most cases the **who** aspect can be trivial, in some situations the debt resolution may need to be assigned to external parties. If remediation is significantly postponed, this field can communicate that decision.

If your team is disciplined, members can easily document the discussion of consequences and change requests as part of the detailed description field of an existing issue type. However, we often observe that software developers explain the *what* and the *where* as they incur or become aware of technical debt, but they fail miserably to highlight clearly the consequences of not fixing it, how the debt might grow over time, and a reasonable time to pay the debt if the fix must be deferred. Therefore, we recommend at a minimum creating a custom field and building the discipline to record the consequences of accumulating debt. That will help you assess how high the interest of the debt is growing. Such a simple practice has powerful operational benefits, such as retrieving all outstanding and possibly closed techdebt issues and assessing their importance and priority against the team's resources.

Table 4.2 shows the technical debt description for Atlas after the second release of its product, with the addition of French to a system that already supports English. The project team is now contemplating a third language.

Table 4.2 *Techdebt on internationalization*

Name	Atlas #5118, language internationalization handling scattered over the code
Summary	The code to handle a second Latin-alphabet language is scattered all over the codebase, and it cannot support other languages, especially non-Latin-alphabet languages. This choice was initially due to schedule pressure to meet a deadline for a demo, which took priority over modifiability concerns. It is also related to the team's unfamiliarity with language internationalization (I18N) and localization (L10N) software.
Consequences	Long delay to add support for a third language (non-Latin-alphabet). We ran an architecture dependency analysis and discovered changes ripple through the system. Change proneness leads to increases in the time to make changes due to the complexity of the system, to integrate due to dependency, and to reuse code and tests because dependent modules must be included. This will be a huge issue if we build on the existing structure. If we wait to make the change until the next release, consequences will be slowing velocity due to accumulation of debt that requires extra work to add support for additional languages.
Remediation approach	Remove the tight coupling that leads to more interdependency, coordination, and information flow issues between the user interface and the business logic. Select an existing I18N library. Xavier from Joe's team studied some options and suggests adopting one of the newer libraries as our best bet going forward.
Reporter/ assignee	Usability team discovered the issue. Joe's developer team will have to deal with it. They are analyzing the impact of the change to give an estimated time frame.

Understanding the Business Context for Assessing Technical Debt

Clearly understanding your business goals is essential for your team to establish criteria for selecting suitable techniques and tools to analyze your software, identify the technical debt, and document the technical debt items that matter to you. It will give you the proper starting point for managing technical debt.

The process to follow in uncovering technical debt is the same as for uncovering any other issue in your system. The challenge is to be disciplined enough to trace the concerns about the business goals to the relevant technical debt item and anchor it in the concrete system artifacts. We recommend starting with the business goals and concerns and anchoring the rest of the activities in those goals accordingly:

1. Understand your key business goals.
2. Identify key concerns/questions about the system related to your business goals.
3. Define observable qualitative and quantitative criteria related to your questions and goals.
4. Select and apply one or more techniques or tools to analyze your software for the criteria defined.
5. Document the issues you uncover as technical debt items.
6. Iterate through activities 2 to 5.

Understanding technical debt starts with enumerating the key business goals and the context of the business. The Atlas, Phoebe, and Tethys projects, like most if not all other software development projects, share a similar goal of reducing development costs, but their different contexts require executing this business goal in different ways.

The business goals have immediate bearing on key concerns related to the systems and, consequently, their source code, architecture, development, deployment, and delivery infrastructure. A clear enumeration of the business goals will help identify the criteria that you need to measure the concerns against. For example, if an organization has a business goal of reducing maintenance costs, some questions to ask about the source code can include "What is the degree of ease and speed required to enhance the software?" and "Does it make more sense to evolve the current system or develop a new one from scratch?" These questions should also take into account the team's position on the technical debt timeline as that will influence the analysis strategies. For example, did the system acquire the debt recently? Or has the debt been accruing for a while, with yet unknown impact beyond the tipping point?

The next step then becomes a matter of defining measurement criteria to assess the answers to those questions. If the criteria are not met within reason, technical debt starts accumulating. The larger the gap in meeting these criteria, the greater the consequences. Technical debt increases the costs of change and rework, so these criteria should be input for assessing the impact of rework and cost of change.

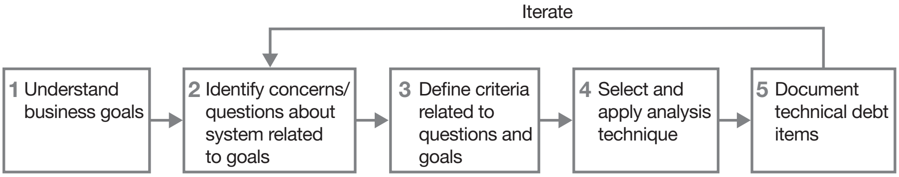

Iterate
1 Understand business goals
2 Identify concerns/ questions about system related to goals
3 Define criteria related to questions and goals
4 Select and apply analysis technique
5 Document technical debt items

Figure 4.2 *Identifying technical debt items*

Moreover, they allow the development team to select and apply analysis methods and tools to assess the artifacts accordingly. The final activity is to document the uncovered technical debt in the form of a technical debt item, while consolidating or linking to related issues, where possible.

After conducting these activities summarized in Figure 4.2, you will have a solid basis to reason about your technical debt. This brings you to the awareness point. You will also probably be able to determine whether you are beyond the tipping point on the technical debt timeline.

Ideally, the process of uncovering and managing technical debt is not a distinct, independent, one-time ceremonial activity but iterative and continuous. As development continues, you will incrementally revise and improve on identifying your technical debt items. Your business goals are not likely to change rapidly, but when they do change, check whether the traceability from goals to questions still holds or whether you need to add new questions and measurement criteria. And, most importantly, listen to team members and understand their concerns about where significant technical debt resides in the system to guide the assessment process and serve as a sanity check on the results. In addition, use the debt assessment process to avoid confusing the causes of technical debt with your current debt. Any sound approach to establishing solid technical debt management practices assumes that you are willing to assess the context and state of your software development project to identify the causes of your debt. In Chapter 11, "Technical Debt Credit Check," we provide a technique to guide your efforts.

Assessing Artifacts Across the Technical Debt Landscape

Using business goals to drive the approach to identify technical debt items will take you throughout the technical debt landscape, to the code, to the architecture, and to the production infrastructure.

Technical Debt and Code

Technical debt is closely associated with the code itself, usually resulting from schedule pressure, lack of a documented programming standard, lack of tools, and developers' errors. You will find plenty of resources on the Web about assessing your code for technical debt and its symptoms, including code quality standards, code smell examples, static code analyzers, security compliance checkers, and the like.

We will devote some time to code-related issues and technical debt in Chapter 5, "Technical Debt and the Source Code," where we describe how to look beyond external quality issues such as defects and recognize when there are internal code quality issues that may require you to deal with technical debt. We explain how using static code analysis techniques can help you discover accumulated issues in source code that could result in technical debt and how to filter and prioritize the results to more effectively avoid unintentional buildup of technical debt.

Technical Debt and Architecture

Technical debt associated with the architecture results from key early decisions made in the design of the software product, such as choices of technology, programming languages, platforms, frameworks, middleware, and how to partition the system. The key difference between technical debt at the code level and technical debt at the architecture level is that the code is much more concrete, tangible, and visible. It can be easily explored and manipulated by using software tools.

Many architectural issues surface during deployment or run-time, even if the structure of the system and its code appear to be satisfactory. Not only is it typically more difficult to detect and assess architectural technical debt with tools, but also the cost and value associated with repaying the debt are larger and deeply intertwined in a complex network of structural dependencies.

Changing major architectural decisions is hard because these decisions have wide-ranging consequences for a software system—its functionality, present and future; its key quality attributes, such as modifiability, performance, security, and availability; and its code, which will need to change to support changes in these decisions. Such changes are usually made in large and long-lived systems, where the payoff will be significant.

Paradoxically, architectural debt is what happens to successful systems and companies: The size and scope increase, the business targets shift, companies merge, and acquisition leads to the merging of incompatible systems or systems that were built using different premises.

We'll examine architectural technical debt more closely in Chapter 6, "Technical Debt and Architecture."

Technical Debt and Production

Not all technical debt is strictly associated with the code or the architecture. Technical debt can also occur in production infrastructure. Current DevOps trends are increasing automation capabilities and tool support, blurring the boundaries between development and operations, and exposing deficiencies in the production process used by the development organization. As a consequence, the delivery environment is becoming a key software development artifact (also referred to as *infrastructure as code*). The production infrastructure contains significant code and has an architecture as well. If the build, test, deployment, or delivery strategy and accompanying tools do not align, evolving the system is harder and riskier.

We discuss technical debt that stems from the delivery process and production infrastructure in Chapter 7, "Technical Debt and Production."

What Color Is Your Backlog?

Step back now and look at what we have added to the work of the software development team: a technical debt registry! At any point in time, the software development team is facing a set of "things" it has to do to make progress and tracking the backlog in one or more development tools. We sort the things to do into four categories, as shown in Figure 4.3.

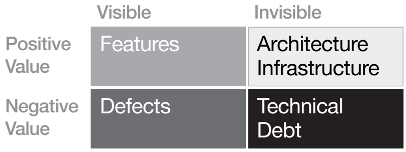
There are things to do that are directly visible to the outside world:

- Adding functionality or features
- Fixing defects

Features *add* value to the product, whereas defects *reduce* the value of the product. These activities drive the organization, the sales, the success, and the customer satisfaction, and they are visible on the release schedule. But there are also things to do that are not directly visible and that are not directly driven by the outside world:

- Defining a software or system architecture and establishing and refining a production infrastructure
- Repaying technical debt

These activities have a cost to implement, though they do not directly add value. Technical debt is in this category: invisible to the outside world but indirectly adding negative value to the software product.



	Visible	Invisible
Positive Value	Features	Architecture Infrastructure
Negative Value	Defects	Technical Debt

Figure 4.3 *The four things to do*

Figure 4.3 summarizes the four kinds of things to do that are managed in the software development product backlog. You can make them easier to distinguish by color coding the items.

- **Features:** Visible and positive value (green)
- **Defects:** Visible and negative value (red)
- **Architecture and infrastructure:** Invisible and positive value (yellow)
- **Technical debt:** Invisible and negative value (black)

Whether your backlog is managed in a single tool or not is another choice.

What Can You Do Today?

Simply providing a means to document known technical debt as technical debt items can be an eye-opening experience for a development team. It also creates a technical debt awareness mindset for the team, which helps reduce the rate of unintentional technical debt going forward. You can use this as a starting point for your registry and step back from there, articulating the overall business goals and deciding what further analysis may be needed. Use the following activities to start documenting your technical debt:

- Refine the “techdebt” category in your issue tracker into a technical debt description. Point at the specific software artifacts involved—code, architecture, or production infrastructure.
- Going further, possibly reorganize your backlog to explicitly “tag” the four categories of work shown in Figure 4.3.

- Create coding, architecture, and production infrastructure standards against which to measure technical debt.
- Standardize on a single form of “Fix me” or “Fix me later” comment in the source code to mark places that should be revised later. They will be easier to spot by using a tool.

Software developers can easily incorporate technical debt management into their daily tasks. For example, for issues that have related technical debt items, developers should incorporate remediation strategies as part of their “done” criteria. We have observed that creating such a technical debt management practice changes developer behavior. Developers disclose their technical debt in the form of discussions and comments by either explicitly referring to issues as technical debt or adding comments such as “fix me,” “workaround,” or “this is a hack.” Creating explicit technical debt items is an opportunity to enable a more proactive management strategy.

For Further Reading

The process we introduce in this chapter to identify technical debt items is strongly influenced by the Goal Question Metric (GQM) approach defined by Vic Basili, Gianluigi Caldiera, and Dieter Rombach (1994).

The motivation and benefits of writing a good technical debt item are similar to those related to writing a good bug report. A lot of research, especially by Microsoft, has underlined the importance of clearly written bug reports, which are likely to get more attention than poorly written ones. Work by Tom Zimmermann and colleagues (2010) provides empirical evidence in this regard. Much of this evidence also supports the importance of writing a good technical debt item. Li and colleagues (2015) have proposed a scenario-based approach to identifying actual architectural debt items.

Developer discussions and code comments include technical debt item discussions. Bellomo and colleagues (2016), Bavota and Russo (2016), and Potdar and Shihab (2014) give some examples of such technical debt items. The technical debt description we discuss in this chapter systematizes this practice. Many authors have identified the concept of a technical debt registry, but in particular, see Narayan Ramasubbu and Chris Kemmerer (2017) at the University of Pittsburgh.

Shane Hastie interviewed Philippe Kruchten for InfoQ in 2010 on the four colors tactic for your backlog illustrated earlier in Figure 4.3.

Chapter 5

Technical Debt and the Source Code

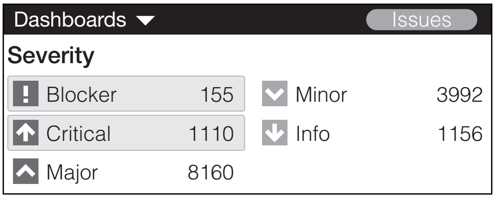
Comprehensive analysis of technical debt requires understanding of short-term and long-term issues with business goals, source code, architecture, testing, and build and deployment infrastructure, as well as how they are all related to each other. While you might conduct separate analyses for each of these artifacts, it is important to recognize that they are intertwined. When you make decisions about remediating technical debt, their interrelationships are especially important, as discussed in later chapters. In this chapter, we explain how to use source code as input to recognizing technical debt.

Looking for the Magic Wand

A web search for analyzing technical debt results in many vendor web pages describing tools, mostly those that conduct automated static program analysis. They promise that such analysis will help measure, and consequently reduce, your technical debt.

When you are faced with technical debt for the first time in a software development project, you might feel tempted to rush out to acquire one of these tools, hoping that you can identify and measure all your technical debt in one magic stroke. But do these tools provide the right approach to understanding your technical debt? And are they sufficiently comprehensive?

Let us look at an example from the Phoebe project. During a quarterly project review, the project manager became concerned about the increasing number of defects. She noted, "Our maintenance costs are increasing." The developers felt that this was the result of *spaghetti code*, or unnecessarily convoluted and unstructured source code. They looked into using a static program analysis tool to help them understand the complexity of their system, and they selected SonarQube, an



Dashboards ▼ Issues

Severity

Blocker	155	Minor	3992
Critical	1110	Info	1156
Major	8160		

Figure 5.1 *Results of the code analysis for Phoebe*

open-source but well-tested tool that supports their Java-based project. Running such code quality analyzers typically yield results similar to that shown in Figure 5.1.

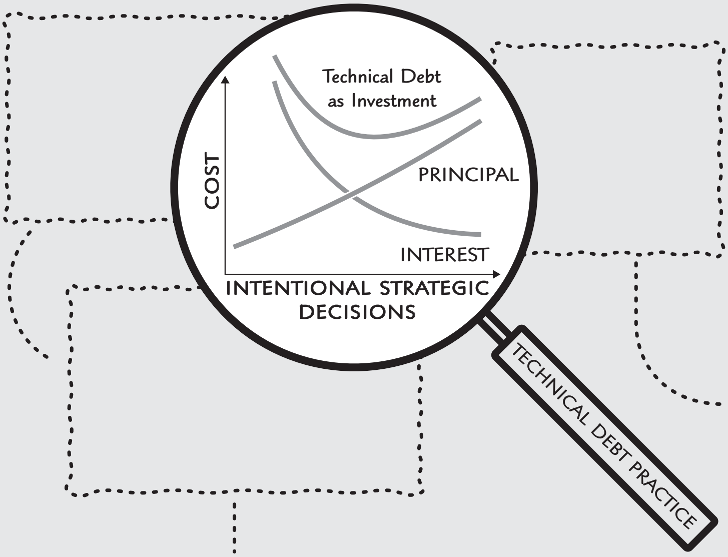
This snapshot demonstrates some potentially confusing results about the quality of the Phoebe project's source code. Static analyzers are likely to provide a long list of issues related to your code, and those issues may or may not be technical debt, it may or may not be essential to resolve them, and they may or may not be related to your current business objectives. Understanding how to use static analyzers to locate your technical debt without getting lost in overall defects or bad code quality is one of the most daunting aspects of technical debt management.

For the Phoebe project, the tool found a total of 13,417 issues in the code, most of which it listed with the severity code blocker, critical, or major. The tool does further sorting by bug, vulnerability, and code smell. Code smells and some of the bugs and vulnerabilities could be symptoms of deeper underlying issues related to technical debt. What should the development team do about them? Should team members record each one as a technical debt item in the issue tracker? Overwhelmed by the result, the team created one new issue and added it to the backlog: "Resolve technical debt based on the results of the static analysis." And there it lingered. Needless to say, this is not a well-defined technical debt description.

We propose a more focused and deliberate approach to technical debt analysis, which includes using static analysis tools only after deciding what you will do with the information they provide. Depending on where you are on the technical debt timeline, you may consider using source code as input for technical debt analysis for three reasons:

- The team is struggling to meet a deadline, and there are increasing numbers of defects. These symptoms should trigger code analysis.
- The team conducted a Technical Debt Credit Check (described in Chapter 11, "Technical Debt Credit Check") and identified causes such as staff turnover, lack of skill development, and time pressure. Such business issues should trigger analysis of the code, as it is likely that mistakes and complexities may have been introduced.
- There are no immediate concerns, but the team would like to be proactive with code quality by performing regular lightweight checks of the code. This is a best-case scenario.

Principle 5: Technical Debt Is Not Synonymous with Bad Quality



The original definitions of technical debt and the wide use of the term in the blogosphere could lead us to think of it as simply bad code quality. Using terms with negative connotations—such as *quick-and-dirty*, *shortcuts*, *bad design choices*, *death by a thousand cuts*, and so on—amplifies this impression.

Low internal code quality is effectively a kind of technical debt—maybe the prevalent kind in the technical debt landscape. Tools including static code analyzers assist in identifying problems with low internal quality and related issues with documentation and testing. However, as Steve McConnell, Martin Fowler, and others have pointed out, there are also deliberate, intentional, strategic decisions at the level of the architecture of the system or the choice of technologies that are made for an immediate gain, usually to reduce time to market. These choices also create technical debt, and they are not related to bad code quality at all.

You may decide not to develop a user interface in multiple languages right away but instead choose to defer this choice to a later time, when the original market's needs have been satisfied. This does not mean that your code is of bad quality. You do, however, need to figure out a way to deal strategically with the thousands of issues that quality analysis of the codebase may reveal.

The information you get from a source code analysis can help you recognize and describe technical debt items and determine where you are on the technical debt timeline, especially whether you've passed the tipping point. In other words, are you approaching the suffering period—when the cost of technical debt surpasses the original value of incurring it—or are you well within it? The following sections proceed through the activities of technical debt analysis described in Chapter 4, "Recognizing Technical Debt."

Understand Key Business Goals

If you don't know where you are going, any route will do. Simply answering the question "How much technical debt does this code have?" is not useful. Technical debt should be identified as an enumeration of meaningful technical debt descriptions, not as an enumeration of code quality violations. Identifying the amount of technical debt occurs in the context of addressing a business goal about system quality and functionality. Investigating system quality and assessing whether it meets the business goals may reveal a portion of the code that is producing the symptoms of debt. The consequences of this piece of code ultimately give rise to two kinds of technical debt interest: recurring (constant additional effort incurred due to keeping this piece of code in the system—that is, living with this debt) and accruing (the cost of changing the system and retrofitting parts).

Each software development organization has its own distinct business goals and objectives. These are highly dependent on the context and product of the organization. In Chapter 3, "Moons of Saturn—The Crucial Role of Context," we cover the many factors that can create conditions for the occurrence of technical debt. Business goals and associated risks serve as a good starting point to articulate how to go about technical debt analysis. Knowing your position on the technical debt timeline and your plans for remediating the debt helps you align the technical debt analysis with your business goals.

Table 5.1 provides some common examples of business goals related to productivity, quality, cost, and time to market. The associated pain points are symptoms that can inform code analysis for identifying technical debt. Some organizations do a good job of clearly communicating short-term and long-term business goals, while some development teams have to infer the goals through the pain felt across their organization as a consequence of the debt they are carrying.

The first row in Table 5.1 shows the business goal "Create an easy-to-evolve product," one of the goals driving the Phoebe project. Symptoms of technical debt have become visible outside the development team, and management has noticed that features are being delayed and maintenance costs are increasing. When management

Table 5.1 *Examples of mapping business goals to the technical debt timeline*

Business Goal	Pain Point	Causes	Point in TD Timeline
Create an easy-to-evolve product	Our maintenance and evolution costs are increasing. Developers are new to the project and say we have spaghetti code, resulting in an increased number of defects. We need to understand the extent of the problem before taking any action.	Inexperienced team members create conditions for the occurrence of technical debt.	Awareness
Increase market share	Customers have started switching services. We have had at least two security breaches in the past six months. We keep patching things up, but we need to step back and understand what is going on in the codebase. More security breaches could result in further loss of business.	The teams stopped following standard procedures and did not understand key architectural requirements—security in particular.	Tipping point: The project is experiencing symptoms, and the team needs to do something now.
Reduce development costs	If we reuse this piece of software, we anticipate reducing our development time, which is currently quite lengthy, but we are not sure if we will incur technical debt in the future if we go forward with the reuse strategy.	Building on a product that already has debt could create more debt; the team does not completely understand the future contexts where reuse may be needed.	Occurrence
Reduce time to market	Our velocity keeps dropping. It takes forever to implement even a simple change and test it, and we don't know what is causing the delays.	Teams do not create sufficient documentation or follow many of the standard processes.	Past the tipping point

asked the developers why, they pointed to buggy code of increasing complexity. The team conducted a Technical Debt Credit Check (described in Chapter 11), which revealed that the cause is the frequent addition of new team members to the project without appropriate onboarding. The organization has just become aware of technical debt on the technical debt timeline.

Identify Questions About the Source Code

Your business goals and position on the technical debt timeline will inform the specific questions and concerns about your source code. Let's keep building on our example. The pain experienced by the Phoebe project in the context of its business goal led the team to ask key questions about the system and consequently its source code: "Where is the maintenance cost being spent? How do we trace symptoms such as defects to the codebase?" Table 5.2 provides the driving questions for the source code analysis of Phoebe's business goals from Table 5.1.

There are two categories of data that Team Phoebe needs to collect information about to answer the driving analysis questions. One is code measurement criteria that can be evaluated with code analysis. The other is symptom measures, such as the number of defects or lingering issues and the time spent resolving such issues and adding new functionality, which can be obtained from solid issue-tracking procedures, along with configuration management and code check-in/check-out procedures. The team can now correlate the results of the code analysis with the symptom measures by answering questions such as these:

- How much time have we spent patching vulnerabilities?
- Where in the code are maintenance costs increasing during development?
- How are defects related to the areas of the code that are causing increased maintenance?
- How often do developers change these areas of the system?
- In how many places in the code do the developers need to implement changes?
- How many change requests are developers able to complete per sprint/iteration? How long does each one take, including testing?
- Where in the codebase do the developers spend most of their time?

These examples suggest that the kinds of questions that code analysis can help the team answer are often related to modifiability, maintainability, and secure coding. There may be other related concerns; for example, to enable reusability, the team may consider moving to decoupled microservices. Static analysis results alone would

Table 5.2 *Common questions for source code analysis*

Business Goal	Pain Point	Driving Analysis Questions
Create an easy-to-evolve product	Our maintenance and evolution costs are increasing. Developers are new to the project and say we have spaghetti code, resulting in an increased number of defects. We need to understand the extent of the problem before taking any action.	• Does the code suffer from established industry maintainability or modifiability issues, such as complexity, cyclicity, or extensive unwanted dependencies? • What percentage of the system is impacted? In which areas?
Increase market share	Customers have started switching services. We have had at least two security breaches in the past six months. We keep patching things up, but we need to step back and understand what is going on in the codebase. More security breaches could result in further loss of business.	• Are there areas of our codebase with known vulnerabilities or secure coding issues? • Are there areas of the code with known security issues that are related to each other? • Are there other similar areas of the code, and do they have similar issues?
Reduce development costs	If we reuse this piece of software, we anticipate reducing our development time, which is currently quite lengthy, but we are not sure if we will incur technical debt in the future if we go forward with the reuse strategy.	• How easy is it to extend the existing software, as measured by criteria such as reachability and dependency propagation? • Are there existing defects and evolvability issues in the software that we need to be aware of?
Reduce time to market	Our velocity keeps dropping. It takes forever to implement even a simple change and test it, and we don't know what is causing the delays.	• How complex is our code? • How understandable is our code?

not return enough information to assess that approach, but it could provide input to the decision-making process.

User-observable operational issues, such as frequent crashes, and unintended functional results may also prompt code analysis for evaluating design fitness. Some examples of checking for design fitness for operational concerns include analyzing memory management, data flow, exception handling, performance, and security. While limited, static analyzers do have analysis rules that check for design fitness. Examples include the following:

- Exception classes should be immutable (performance and security).
- NullPointerException should not be explicitly thrown (performance and security).
- The user interface layer shouldn't directly use database types (enforce Model–View–Controller pattern).
- Avoid the Singleton pattern (improve testability).

Define the Observable Measurement Criteria

By now we hope we have made the point clear: Static analyzers provide useful information, but there is no magical metric or tool for identifying technical debt with code analysis. There are some common maintainability/modifiability threads among business goals and concerns about source code, as our examples demonstrate, but one-size-fits-all measurement criteria for driving business goals do not exist. Choice of technology and development language as a consequence of the business goals also influence the measurement criteria. Therefore, a development team should determine the measures that will help members analyze a system in light of the analysis questions the team generated.

If your source code is messy, then you are probably paying a lot of recurring interest. Recurring interest occurs in the form of added time to implement new features or test the system, added complexity that results in increased maintenance costs, and system structure and behavior that are hard to understand and explain. In such cases, there is no one area that you can scope as the location where the debt resides, but overall the code has become too brittle. Creating concrete technical debt items helps you focus on the problem and document supporting evidence as you apply different kinds of analysis, be it tool-supported code analysis, architecture and design reviews, or infrastructure monitoring metrics. We will discuss the latter in subsequent chapters.

Table 5.3 associates symptom measures and code measurement criteria with the Phoebe project's business goals to better elaborate the pain points and driving analysis questions. The quality measures provide a means to measure the pain and to check whether the symptoms are decreasing as the Phoebe team makes changes to repay the debt and improve the system. Conducting the source code analysis against measurement criteria associated with business goals will generate an initial list of candidate technical debt items.

These are examples of starting points to help you recognize how multiple sources of information are related to each other. One theme that emerges from these

Table 5.3 *Examples of symptom measures and code measurement criteria*

Business Goals	Symptom Measures	Code Measurement Criteria
Create an easy-to-evolve product	Defect trends (new defects per iteration, defects lingering over multiple iterations)	• Maintainability and evolvability violations against established industry measurement standards (e.g., the ISO/IEC 25010 standard for system and software quality) • Code complexity measures (e.g., combination of source lines of code, coupling and cohesion, fan-in/fan-out, dependency propagation) associated with the current maintenance costs and defect rates
Increase market share	Security bug trends Amount of time spent patching	• SEI CERT secure coding standards
Reduce development costs	Propagation of change	• Maintainability and evolvability measures • Code complexity
Reduce time to market	Changing velocity	• Maintainability and evolvability measures • Code complexity
Improve governance	Potential effort spent per violation	• ISO/IEC 25010:2011 system and software quality model • Specific coding standards for quality models

examples is that unmaintainable code can result in declining development efficiency. The development team needs to ensure that members minimize accidental complexity in order to manage the system with minimal unintentional technical debt and keep the codebase understandable.

The devil is in the details. Writing clean, understandable, and well-thought-out code is every team member's responsibility. Integrated development environments, automated code review, and unit testing software as well as static code analyzers have increasing capabilities to assist developers in writing high-quality code. Improving the capabilities of these tools is an ongoing challenge for the software industry, especially in minimizing false-positive rates and warning messages and making it easy for teams to incorporate them into their day-to-day development activities.

What Tools Should You Use?

Probably the most commonly asked question about technical debt is "What tool should we use to measure it?" Tools that link analysis results to your business goals can help you identify and manage technical debt. Tools can also be extremely useful if they can be integrated into continuous integration tool chains, giving timely feedback to the developers, who can then decide how to minimize unintentional technical debt.

As we already established, determining the right measurement criteria, tools, and techniques for analyzing source code depends on your business goals. And static analysis results alone will not provide a list of technical debt items. The technical debt items are the areas of the system where the quality measures are symptomatic of the violations of code internal quality rules revealed by the code measurement criteria.

Techniques such as code inspections and peer reviews can provide some of the analysis results for the established measurement criteria. But a number of static analyzers have increasing capability for assessing source code quality. Some examples include Understand for C/C++; SonarQube for Java and C/C++; Klocwork for Java, C/C++, or C#; and AppScan for analyzing security in mobile apps and web-based systems. By the time you read this book, there will probably be other relevant tools that we could have listed. Some of these tools have real-time support, and they call attention to security weaknesses and coding errors as developers write code.

The Object Management Group (OMG) released the Automated Technical Debt Measure specification developed by the Consortium for IT Software Quality. The specification includes 86 measures for maintainability, performance and efficiency, reliability, and security, as well as the estimated time to fix each violation of a measure based on a survey of developers. The estimates for individual violations culminate in one rolled-up technical debt figure based on the aggregated measures and adjusted for the software context that can cause a variation in the time to fix. These factors include complexity, concentration, evolution status, exposure, and technological diversity. Whether a tool implements these measures and whether the measures are relevant to your system quality goals determine how much benefit you may get from managing your technical debt using this specification.

The ideal usage scenario in adopting a static code analyzer is to get ahead of unintentional technical debt by using a tool that is accepted by developers and integrates well in their day-to-day workflow. Google developed an in-house tool, Tricorder, to address this challenge. The motivation that led Google to develop an in-house solution was to ensure that the tool could scale to its needs and empower developers to write and deploy their own static analysis to fit their needs. One driving success factor that resulted in Google developers incorporating Tricorder into their development flow was the ownership they were given that enabled them to disable rules that did not serve their context and write rules that did. This is a proactive approach to catching code problems before they turn into technical debt.

As these examples demonstrate, just as there are no one-size-fits-all measurement criteria for technical debt, there is no one-size-fits-all tool that will help you understand the issues related to technical debt in your code or overall system.

Select and Apply an Analysis Tool

System quality goals for the Phoebe project include minimizing new defects at each iteration and the amount of time existing defects linger on the backlog. Accordingly, the team established code quality criteria that included industry standards for writing maintainable code and avoiding code complexity. However, despite these actions, the team found itself struggling with unresolved defects, in particular hard-to-trace defects in the code. According to the developers, one cause of messy code is the tendency to copy and paste blocks of code. To avoid this practice, a new packaging scheme had begun to be implemented, but the developers suspected that it hadn't been fully implemented yet.

To help evaluate the code quality, the Phoebe team selected SonarQube as its static analysis tool because it is open source, has a community to address developer questions, has a reasonably well-established rule set for Java, and incorporates maintainability as well as security measures. SonarQube can detect duplicate code blocks and the presence of packaging schemes. Developers' knowledge of the areas of messy code helped them configure the tool to run on these areas of interest. The code measurement criteria also helped them configure the rules and set their priority.

As the team interpreted the analysis results, a closer look revealed that about one-fourth of the 13,417 issues were related to duplicate code blocks around adapters. This observation overlapped with the developers' observation about spaghetti code that was hard to understand. Using the navigation features of the tool, the developers located the areas in the code that were impacted most severely. The results also showed a large number of empty Java packages; while such issues are usually minor, they significantly increase the recurring interest on the debt because they increase the software footprint and reduce the clarity of the system. The developers also mapped these areas to the number and kinds of defects they had observed.

Document the Technical Debt Items

Once the development team has generated the initial data, the next step is to ensure that members record the relevant results as technical debt items so that they can start managing them. This is the role of the technical debt registry, which can be any tool already used by the project: an issue tracker, a defect tracker, or a backlog management tool. The team should take two actions:

1. Document existing technical debt and create a strategy for paying it back.
2. Address how to ensure that the team does not inject new debt into the source code so no one has to deal with this many thousands of issues again.

These actions require establishing and enforcing some development practices. Here we focus on documenting the existing technical debt items. Later, we summarize development practices that minimize unintentional technical debt in the code.

The Phoebe team wondered if it should look at all 13,417 issues tagged as tech-debt. Or should the team focus only on the blockers, which total 155 items? Or should it also include the major and critical issues in the registry? Team Phoebe started this analysis of the source code to see if any of the findings would overlap with the increasing defects and maintenance costs. The team recognized that the duplicate code and empty Java packages contributed significant amounts of recurring interest in the form of decreased understandability and small but annoying defects that ripple through the duplicated code snippets.

The team decided to introduce two major technical debt items into the registry: remove empty Java packages (see Table 5.4) and remove duplicate code (see Table 5.5). The experienced developers on the team also recognized that while they uncovered these technical debt items through a static code analysis of their codebase, resolving them would probably require some architectural thinking and analysis. For example, rather than duplicating the code, they would need to think about a common service that could be invoked.

They each recognized that the accrued cost of removing the empty packages was currently low, but it could increase over time if developers started adding code, creating a drift between the implementation and the initial architecture of the system. Removing the empty Java packages took care of about 250 of the issues returned. Once the packages were removed, several hundred minor issues also disappeared. Alternatively, the team could have elected to exclude these packages from the source code analysis. In this case, however, including them helped the Phoebe team recognize a recurring interest that it had been paying every sprint, in addition to complexities arising from the unnecessarily increased deployment footprint of the system.

Table 5.4 *Techdebt on empty packages from the registry of the Phoebe project*

Name	Phoebe #345: Remove empty Java packages
Summary	The re-architecting of the source code to support multiple adapter specifications has introduced a new Java packaging scheme. Numerous empty Java package folders are present across multiple projects.
Consequences	No impact to functionality; however, may lead to confusion for users implementing enhancements or modifications to the source code.
Remediation approach	Using SonarQube, the team identified the empty packages. New and existing classes have been moved into new package folders; however, the previous package folders have been left in place with no class files. Cleaning up these packages should be trivial and ensure that there are no unintended calls left.
Reporter/assignee	A composite technical debt item as a result of our SonarQube analysis retrospective. Will be assigned to the Adapter team.

Table 5.5 *Techdebt on duplicate code*

Name	Phoebe #346: Remove duplicate code
Summary	AdapterCore and CoreLibrary grew organically with a lot of copy/paste code, resulting in over 40 blocks of code duplicated within each of the modules in these subsystems.
Consequences	No immediate impact to functionality; however, every time a change needs to be made, several small defects are injected due to the inability to propagate the changes to the blocks of code duplicated.
Remediation approach	See the results of the SonarQube analysis to identify the classes. The remediation will need to include a re-architecting effort and possibly introduce a factory class to handle the common functionality across the duplicated blocks. This should have been undertaken with the adapter architecture changes.
Reporter/ assignee	A composite technical debt item as a result of our SonarQube analysis retrospective. We will have to postpone this to the next sprint as the effort involved is higher than we anticipated.

Dealing with duplicate code is not as simple as removing empty packages because the remediation strategy requires architecting a new solution to pay the current principal. To address the accruing interest, Phoebe's development team recognized that the team needs to retrofit a significant number of classes with duplicate blocks, which introduces a risk and adds time commitment, especially in testing. Therefore, the remediation approach field emphasizes that the team needs to conduct further re-architecting.

A large percentage of the major violations that the tool reported are related to how exceptions are handled, how errors are logged, and how comments and commented-out code are handled. These violations signaled to the Phoebe project manager that the development team needed a reminder about using development practices that avoid introducing unintentional debt, in particular by focusing on good software craftsmanship and understanding of software design.

Then Iterate

Following the process for identifying technical debt items that the Phoebe team demonstrated, team members decided to understand the complexity of the system first. The business goal and questions about the source code that they struggled with were related to increasing defects as a consequence of spaghetti code. So, they decided to analyze the structure of their code as well as its quality with the help of static code analyzers and prioritize what areas would need to be remediated. They did this prioritization based on the areas that were evolving the most and where they observed the most defects.

When you complete a first analysis iteration, the goals and frequency of subsequent iterations should become clear. You might learn that analysis for security and analysis for maintainability require different analysis questions, criteria, and tools, so you might conduct these analyses in two different iterations of the technical debt analysis. In another scenario, once you identify the technical debt items, you can decide on a frequency of analysis to ensure that similar code quality issues do not accumulate, in which case you may only apply the tool and document the issues.

When the software undergoes major changes or when business goals change, it makes sense to iterate all the activities to align the analysis process with the new software or new business goals. If you are intentionally taking on technical debt, then you are motivated to optimize your effort for a business goal anyway. Consequently, the measurement criteria derive from the intentional design decisions that lead to technical debt. Having observable measures within the code that map to the design decisions allows proactive management of technical debt within the code.

What Happens Next?

After selecting analysis criteria, running tools, and inspecting the code, you probably have a handful of technical debt items in your technical debt registry. The process for identifying technical debt items also assumes that you are performing the analysis in retrospect. The registry does not answer questions about the overall assessment of the system's technical debt, but individually the items do address problematic areas. At this point, you can take two courses of action:

1. Address each technical debt item in isolation through local refactorings within an iteration/sprint boundary.
2. Consider the dependencies between the technical debt items.

Larger projects may require some planning across iteration boundaries.

Refactoring is the process of restructuring existing code without changing its external behavior. Depending on the nature of each technical debt item, refactoring the code locally without inducing architectural change may be the best strategy for removing the debt. To determine the most cost-efficient approach to paying back your debt, the technical debt description provides a starting point for estimating the cost of each technical debt item in isolation. The consequences identified help you understand the recurring interest. The analysis of change provides input about accruing interest. Using the following simple formula, assess each item over several iterations:

> Recurring interest (consequence) + Accruing interest (propagating cost of change) × Probability of high-cost change scenarios

In this formula, the greater the probability of change, the higher the total cost of debt. When deciding whether to pay down debt, compare the cost of the impact of different change scenarios. If the technical debt items do not have the potential to cause ripple effects or if they have no dependencies on other items, the approach of focusing on one item at a time might work. We elaborate this first course of action in Chapter 8, "Costing the Technical Debt."

However, software development is rarely that simple. More often than not, you will have to treat the technical debt items in reference to each other. Therefore, the second approach requires a more elaborate design and traceability analysis to assess the dependencies within the system boundaries as well as the technical debt items. We tackle this second course of action in Chapter 9, "Servicing the Technical Debt."

What Can You Do Today?

Now that you know how to use source code to recognize technical debt, you can begin looking beyond external quality issues such as defects and recognize when you have internal code quality issues that may require you to deal with technical debt. Start by conducting these activities:

- Understand the business context to guide the use of source code as input for technical debt analysis.
- Acquire and deploy in your development environment a static code analyzer to detect code-level issues.
- Analyze the code for the presence of unintentional technical debt and respond by including debt items in the technical debt registry.

As an example, the Phoebe project realized the importance of maintainability for a system. Lack of maintainability and technical debt are not the same thing, but unmaintainable code will have a lot of unintentional technical debt. It is never too late to make maintainability a non-negotiable software design principle for your projects.

For Further Reading

There are several guidelines and standards for code quality. The *ISO/IEC 25010:2011 System and Software Quality Models* standard summarizes quality characteristics, internal metrics (metrics that do not rely on the execution of the software), and external metrics (those applicable to running software) (ISO/IEC 2011).

The Consortium for IT Software Quality (CISQ) publishes standards for automated measures of quality characteristics in security, reliability, performance efficiency, and maintainability. These standards are a concrete place to start defining the code measurement criteria related to your quality attributes.

The CERT Division of the Carnegie Mellon University Software Engineering Institute has published secure coding criteria for C/C++ and Java that have become an industry standard (SEI 2018).

A book by Visser and his colleagues (2016), *Building Maintainable Software*, has successfully simplified the otherwise complex and ubiquitous problem of maintainability to ten guidelines, with examples in C# or Java. The book reflects the decades-long experience of the Software Improvement Group in the Netherlands in assessing software projects.

In addition, the industry has developed methods to improve automation of software quality. For example, Software Quality Assessment based on Lifecycle Expectations (SQALE) is a method developed by Inspearit, Inc., that identifies code violations based on a categorization of testability, reliability, changeability, efficiency, security, maintainability, portability, and extensibility. The method creates an assessment on technical debt reduction based on fixing these issues (Letouzey 2016; Letouzey & Ilkiewicz 2012).

There is wide application of using static code analysis to assess technical debt. Two examples are work by Arcelli-Fontana and colleagues (2015) and Zazworka and colleagues (2014). They show how to analyze for code smells that can lead to technical debt and summarize similar challenges of using the existing tools, as discussed in this chapter.

Chapter 6

Technical Debt and Architecture

In this chapter, we explain how to recognize technical debt at the architectural level. We introduce lightweight structural analysis techniques that you can apply to the code or the design to help identify and understand design decisions that lead to technical debt.

Beyond the Code

In Chapter 5, "Technical Debt and the Source Code," we showed how the accumulation of small deficiencies in the code can lead to a substantive amount of technical debt, which can in turn make forward progress harder, more costly, and more error prone. But there is increasing evidence that the most expensive technical debt is related to the *architecture* of the software system—and it is harder to pay back. The effective management of technical debt must therefore extend beyond coding issues and consider the architecture of the system.

One common example of this type of technical debt is created when a development team, pressed for time, designs an initial system with little modularity for its first release. This lack of modularity affects development time for subsequent releases. Additional functionality can be added later only by doing extensive refactoring, and this refactoring impacts future timelines and introduces additional defects. In this category, which we will call "architectural debt," we find not only the structure of the system—organization, decomposition, and interfaces—but also the choice of key technologies, from operating systems to programming languages and from selection of frameworks to open-source components.

Compared to code-level debt, architectural debt is more likely to be *intentional*. It follows from decisions made in the early phases of a project, often because the development team did not understand how the system would evolve in the future or because the business context significantly changed. Architectural debt can also be an unintentional consequence of what we called the *technological gap* in Chapter 2, "What Is Technical Debt?": The original design was fine at the time it was made, but technology evolved over the years, turning the original choice into technical debt. For example, perhaps you designed a system with a local database, but 10 years later, having all your data in the cloud would be a better choice, and your local database now represents technical debt.

In Chapter 5, we explained how tools can assist in spotting most of your code-level technical debt. For architectural debt, these tools are less helpful. Some tools can expose the structural issues of a system, such as circular dependencies, high coupling between modules, and classes that have too much responsibility. These and other practices result in unmaintainable and hard-to-modify systems that require significant rework later in development; hence they accumulate technical debt. But there are aspects of the architectural debt that cannot simply be detected by tools. This type of debt must be dug out of the heads of the people most familiar with it: its designers. No tool will tell you that you should have used a NoSQL database instead of a relational database. Architectural constructs and decisions are in many cases only *conventions* used in further design and implementation.

There is a direct relationship between a well-thought-out architecture that also guided the implementation of the system and a manageable accumulation of technical debt. For example, if the goal is for the system to be sustained for decades and to respond to changing technology, the architecture of the system must enable separation of concerns, use decoupled technology layers for ease of upgrading, and ensure that change is localized for ease of adding new functionality. These are important architecture concerns that should drive the design reviews as well as manifest themselves in the codebase, not only at the beginning of the system's development but throughout its lifecycle. The system should be designed and monitored for *quality attributes*, or architecturally significant requirements, such as requirements about how reliable, secure, or maintainable the system is. Quality attributes help focus attention on cross-cutting aspects of the system, such as how it performs under different conditions, how data flows and is managed, and how it depends on other software such as databases, user interface and backend frameworks, middleware, and so on.

We can supplement the limited functionality of tools in uncovering architectural debt by assessing specific quality attributes. Again, these assessments will likely mostly reveal *symptoms* of technical debt; designers will have to identify the actual architectural elements that are subject to debt as technical debt items. For example, when scaling from a few hundred users to 10,000 simultaneous users, the drop in performance is a *symptom* of technical debt: A key quality attribute is affected. The symptom is caused by the large number of remote procedure calls between these two subsystems—the debt item itself—which was a not a problem when the system had only a few hundred users.

Here is an example of architectural debt voiced by a developer of the Phoebe project:

> There were some problems in the infrastructure code where there was originally an architecture in place, but it wasn't followed consistently. So, thought had been given to the architecture, but in the implementation, shortcuts were taken, and dependencies were not clean. This shows up as increased complexity and coupling in the codebase.

This phenomenon is called *architectural drift*: The intended architecture is poorly or inconsistently implemented throughout the system. This example emphasizes that this kind of technical debt accumulates slowly over the life of the project, which gradually drifts into debt. It is not a sudden, visible event that could trigger corrective action. Now Phoebe developers know the areas of the codebase where the increased complexity has become overwhelming, and their best course of action going forward is to concretely specify the highly complex areas. With some strategic thinking, code analysis can help you uncover such accumulating architectural issues.

Paradoxically, too much early focus on architecture and evolvability may lead to technical debt, too. The developers of Phoebe complain:

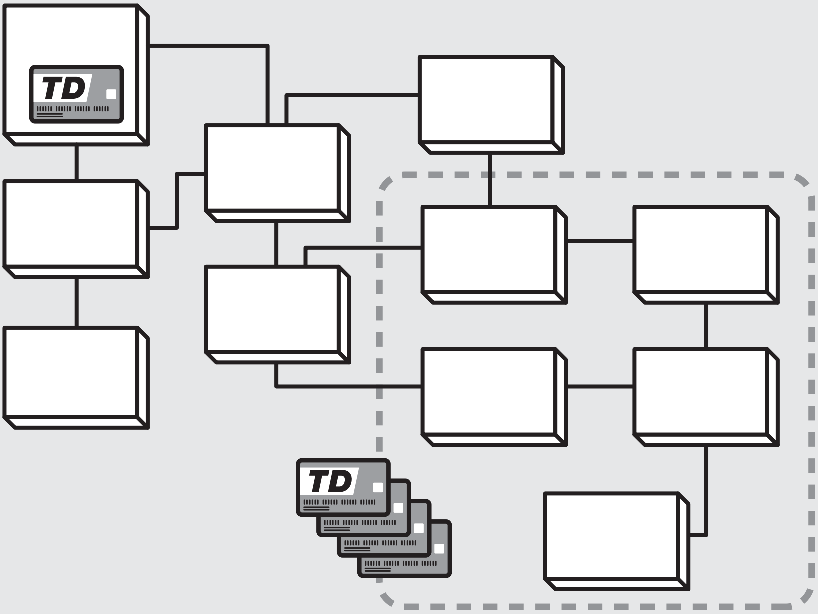
> The original design had lots of options and flexibility, which in the end we were never to exploit. But as a result, many of the interfaces to key components are very heavy, complex, hard to use (especially by newcomers in the project), and error prone. This is now slowing us down, with no real benefit yet to the project.

There are several strategies you can use to uncover technical debt in the architecture of a system as you iterate through the activities of the technical debt analysis (as described in Chapter 4, "Recognizing Technical Debt"). You can ask the designers about the general health of the system or start with a problem. You can examine the architecture itself or the code and other software artifacts to get insight into the architecture. Typically, the best approach is a combination of these activities:

- Ask the designers about the health of the system or a problem.
- Examine the architecture.
- Examine the code to get insight into the architecture.

We'll review these options in this chapter. The starting point, the line of investigation, and the analysis differ among these three approaches, but the objective is the same: to identify architectural technical debt items in the context of key business goals.

Principle 6: Architecture Technical Debt Has the Highest Cost of Ownership



Architectural technical debt items have impact across the system as they are deeply intertwined in a complex network of dependencies. If the architecture is not well thought out, costs accumulate as the system becomes hard to evolve. Changing major architectural decisions can be much harder than changing source code, especially as the system grows, since such changes have wide-ranging consequences. Remediation is a major undertaking that may span several iterations or consume most of the available resources over multiple releases.

Ask the Designers

Ask the people who know the system best, the *designers* themselves, about the current state and history of the system. Ask the designers about the general health of the system or start with an important problem.

Here is a sketch of a strategy to inquire about the general health of a system and start locating technical debt items:

- Identify the people who have been involved in the project as software architects, technical leads, or experienced developers.
- Secure some time to meet with them individually or in small groups of two or three. A one-hour interview should give you enough information.
- Explain clearly the objective of the meeting and define the term *technical debt*. Stress that it may not be major defects of the system that are already known and visible in the project issue tracker. To better focus the interview, you may also explain some of the ultimate goals: flexibility, shorter release cycle, higher dependability, and so on.
- Ask questions such as these:
    - In retrospect, what design decisions did you or others make about the system that you regret now?
    - Why do you regret that decision now? (What are the negative consequences?)
    - Was there an alternative at the time?
    - Is this alternative still feasible today?
    - Can you envision another alternative that would remediate the situation?
- Focus only on the software, not on the people who made the not-quite-right decision, or who pushed the team to do so, to avoid blaming anyone.
- Rephrase the concern to express the *technical debt items*—the software artifacts affected, causes, and consequences.
- Break down generic, high-level concerns into several smaller technical debt items.
- You may rapidly find references to already identified technical debt items as you do a sequence of individual interviews; move quickly to each new one.
- Quickly move on when you encounter what appears to be a matter-of-taste issue: “For this kind of system, I much prefer Java over Ruby. Our original choice of Ruby was a mistake!”

Doing individual interviews has some advantages and some drawbacks: On one hand, it is more costly and time consuming. On the other hand, it allows Designer 1 to express concerns about a decision made by Designer 2, who may be his or her supervisor or a much more senior person. Honesty might be harder to express in a group setting, depending on the culture of the organization.

Some of the findings from these interviews may have to be validated by inspecting the design and code. On very large systems that have evolved over time, or if the interviewee has not worked on the project recently, some technical debt items may have already been repaid. You may be told, for example, that "we removed MySQL and replaced it with Neo4J for Release 7 about three months ago."

This interview strategy will bring out the elephant in the room, the technical debt that everyone is aware of but does not want to express for a variety of reasons:

- Protecting the person who made the decision that resulted in technical debt, who may be a key player in the organization
- A fatalistic feeling that nothing can change the system now, or it would be too costly, so why bother
- Cultural and social dynamics issues, such as losing face
- Familiarity with the current situation and fear of the unknown (uncertainty avoidance)

The Five Whys is an iterative interrogative technique used to explore the cause-and-effect relationships underlying a particular problem. The primary goal of the technique is to determine the root cause of a defect or problem by repeating the question "Why?" Each answer forms the basis of the next question. When multiple causes are suspected, they can be represented as a fishbone, or Ishikawa, diagram. Here is an example of inquiring about an observed symptom that involves asking "Why?":

> "This type of update takes too long to make."
>
> "Why?"
>
> "Because the code to update is in six different places."
>
> "Why is the code in six different places?"
>
> "Because of the strict decomposition of classes to realize the domain-neutral component pattern we picked."
>
> "Why are we using this pattern?"

The outcome of this activity is the addition of technical debt items to your technical debt registry. These new technical debt items must be investigated by inspecting the design or the code.

Examine the Architecture

A number of analysis techniques have proven useful for examining the architecture as it is being designed and used throughout the software development lifecycle:

- **Thought experiments and reflective questions:** Conducting thought experiments and asking reflective questions can augment analysis. People think differently when they are solving problems than when they are reflecting. Asking reflective questions can challenge the decisions people have made, and that challenges them to examine their biases. Ask questions such as these: What are the risks that certain events will happen? How do the risks influence the solution? Is the risk acceptable?
- **Checklists:** Use a checklist to guide your analysis. A checklist is a detailed set of questions developed based on much experience evaluating systems. Checklists can come from taxonomies of quality attributes and associated architectural tactics that cover the space of design possibilities for managing the quality attribute. For example, architectural means for controlling the properties of modifiability are concerned with coupling and cohesion. Ask questions such as these: What is the cost of modifying a single feature? Does the system consistently support increasing semantic coherence? Does the system consistently encapsulate functionality? Does the system restrict dependencies between modules in a systematic way? Does the system design regularly defer binding of important functionality so that it can be replaced later in the lifecycle, perhaps even by users? Checklists can also be based on experience with particular technology choices or specific domains.
- **Scenario-based analysis:** A scenario is a short description of an interaction with the system from the point of view of one of its stakeholders. A stakeholder may pose a change scenario to see how costly it would be to modify the system, given its architecture. Analysts can use quality attribute scenarios to examine whether and how a scenario can be satisfied.
- **Analytic models:** Well-established models can be used to predict properties of a system such as performance or availability.
- **Prototypes and simulations:** The creation of prototypes or simulations complements the more conceptual techniques for analyzing the architecture. Prototypes provide a deeper understanding of the system but with added cost and effort.

A risk is an indicator of poor architectural health. These analysis techniques can bring to light architectural risks, potentially problematic design decisions whose consequences put the achievement of system requirements at risk and business goals in jeopardy. Over time, if overlooked, they can create large amounts of technical debt. Design issues in conjunction with evidence of accumulating rework could result in adding a new technical debt item to the registry or conducting additional analysis to confirm whether there is a risk or not.

Looking for Debt in Your Databases

by Eoin Woods

Much has been written about technical debt and how to find it, manage it, and avoid it. But nearly all this work relates to the algorithmic and structural aspects of an application. This ignores another important type of technical debt that can be a significant liability for a system—that found in its data models, the code that accesses databases, the database schemas, and the data stored within them. This debt can be incurred at the conceptual and logical levels as well as the physical levels of the data model (Al-Barak & Bahsoon 2016).

Manifestations of Database Debt

Intentional database debt is often found in situations where a specific set of trade-offs is made during the database design phase to achieve specific quality property requirements, typically performance or modifiability. This can result in a system with a very highly normalized or denormalized relational database schema. These database designs can then have undesirable side effects for other qualities, resulting in problems such as query complexity or high degrees of data duplication that make change difficult.

For accidental database debt, the trade-offs are often not clearly identified. An example is where parts of the database schema are overloaded, and entities or tables intended for one purpose are used for another (such as a transaction table being used to hold “magic” rows with special identifiers that hold summaries or totals). This can occur at conceptual, logical, or physical levels of abstraction.

Another problem that manifests at logical or physical levels is schema structure duplication. For example, there should be one entity or table for, say, all the sales records, but due to variations between regions, the expedient choice was to have separate, slightly different sales record tables for each region, making consolidated reporting and changes to sales record keeping much more difficult.

Most relational databases incorporate features to allow metadata such as column nullability, foreign and primary key constraints, and data constraints to be stored in the schema. These features can help reduce technical debt within the physical data model and ensure that the logical model is correctly implemented and maintained. However, to save time in initial implementation, this step is often skipped, and the result may be a database that is hard to work in and keep consistent, as well as other problems such as "foreign key" debt, as identified by Weber et al. (2014).

Some types of database technical debt occur specifically at the physical implementation level.

One common problem seen in old systems is the abuse of strings that are used to hold types of data that are interpreted in application code to be much more specific types (such as numbers). This can even happen with date types. I have seen situations where some records in a table had very odd dates in a column, all a long time in the past. When I investigated further, I discovered that someone had decided to store an integer value in the column in some cases, converted it to a date in the code, and simply reversed the process when they needed to read it again!

Nearly all databases rely on indexing for query performance, and a related physical-level problem that can be prevalent in new and old systems is a lack of consideration of indexing during development or maintenance changes. This results in a database or specific queries performing fairly well for small data volumes but mysteriously becoming disastrously slow as soon as significant amounts of data are added.

When working with physical database design and implementation, achieving good database performance can be a complex balancing act of conflicting forces (such as update versus retrieval performance), and working out the right balance takes time, skill, and experience. Sometimes when pressure is tight, we rely on intricate query or optimizer hints, tricks, or obscure configuration settings, which are like a sticking plaster in that they solve the problem right now, but they become parts of the system that no one dares to change as they are hard to alter without hurting performance.

There are also some types of database technical debt that are introduced during the design and development phases of system delivery.

Some databases, particularly relational databases, allow large amounts of sophisticated code to be stored in the database as procedures and triggers. This code often becomes badly understood spaghetti code as it is written in specialist languages and can be difficult to test in isolation, making it difficult to apply techniques such as test-driven development.

(continued)

Most database systems work best when processing sets of data. This is particularly true for relational databases that are inherently set processors. However, many inexperienced developers don't know this and have an iterative "row-by-row" mindset, which leads them to write code that accesses the database a row at a time. This is highly inefficient and may work well for small tests, but the code inevitably needs to be rewritten for production use.

It is also important to apply the right database model to the problem at hand. Over the past few years, we have had an explosion of the so-called NoSQL databases, which include document databases, tuple stores, distributed cache-based data stores, and graph databases. Each type suits certain kinds of workload very well and is very poor at others. Knowing which to apply takes experience, and using the different models adds complexity. A common type of database design debt is using one database model for all types of problems due to familiarity with or ease of access to it. This can result in an unsuitable use of the database—such as a relational database struggling to process graph-style queries—and inevitable maintenance problems later.

Avoiding Database Debt

Given that database debt is not only possible but probably inevitable on most systems that include a significant database, what can you do to avoid or mitigate it?

The key point is to treat database debt as a potential problem whenever building or maintaining systems with complex databases. The techniques you use to avoid debt in the rest of the application—such as pair programming, design reviews, code reviews on check-in, automated tests, standards, automated code checking, and refactoring—are just as important for database code.

That said, some of these techniques are quite difficult to do, such as unit testing database access code or SQL. Similarly, automated code quality tools for database code are significantly less common and advanced than for languages like Java and C#. This means that as well as awareness, you need a proactive approach to prioritize the monitoring and management of database technical debt in your projects and may need to integrate unfamiliar tools into your environment (see Arulraj 2018; and Redgate 2018).

A database is a critical component within many computing systems, but practitioners have often ignored the potential problems of technical debt building up within this part of the system. This has caused significant operational and maintenance problems in many systems as their databases have grown in size or needed to undergo significant changes.

It is important to maintain awareness of the potential problems that lurk in the database layer. If you want to sustain useful and flexible systems, you need to monitor for database technical debt just as actively as you would monitor the rest of the application's design and implementation, and you need to be prepared to invest in remediating these problems over the long term.

Examine the Code to Get Insight into the Architecture

Even if you do not have a description of the architecture to work with, you can still get insight into the architecture by examining the code with the help of a tool that understands dependencies and structures in the code.

Tools that support code analysis are becoming increasingly sophisticated and now often also support dependency analysis. Quantitative techniques involve applying some technique or tool to a software artifact to answer specific questions about specific system properties. Many of the quantitative measures used on code can be applied to the implementation structure or module view to assess the state of the architecture. Some tools provide the ability to extract this module view directly from the code. Other tools provide the ability to represent the module view as designed and compare it with the code structure to check that the code conforms to the architecture.

Code measures have been adapted to code and design elements of increasing scale. For example, cyclomatic complexity has been adapted to code and design elements such as methods, classes, packages, modules, and subsystems of large systems; complexity can serve as a starting point for understanding how a system is structured. Some tools also include rules to check for well-established architecture-relevant patterns—for example, decoupling business logic from SQL statements (Model–View–Controller) or checking for conformance to framework usage. Run-time measures bring to the surface other architectural concerns that have close relationships to how the code is structured—for example, how services are decomposed and interact with each other, how responsive the system is, and how data is handled.

To understand the impact of a change, developers need to identify the modules of a system that are the focus of a change and follow the dependencies to the dependent modules that will be affected by the change. Relevant techniques for analyzing individual elements and their dependencies include the following:

- **Complexity of individual software elements:** Lines of code, module size uniformity, cyclomatic complexity

- **Interfaces of software elements:** Dependency profiles identifying hidden, inbound, outbound, and transit modules; state access violation; API function usage
- **Interrelationships among the software elements:** Coupling, inheritance, cycles
- **System-wide properties:** Change impact, cumulative dependencies, propagation, stability
- **Interrelationships between software elements and stakeholder concerns:** Concern scope, concern overlap, concern diffusion over software elements

In using these techniques, it is important to focus not only on the results but also on the assumptions under which a measurement was taken. Not all measures are applicable, but there are a number of useful measures to draw from. Those you select will depend on a number of criteria. What part of the system are you measuring? Account for external dependencies, libraries, and frameworks. What is being measured? Tools often produce different results for seemingly simple measures such as lines of code. How is the system represented? For example, propagation measures make assumptions about data and control flow using an abstract model of the code that makes trade-offs in the fidelity of the results (for example, accuracy and precision). How are results combined? Some tools roll up technical measures into a single economic measure of health. The underlying measures can still be useful. For these reasons, it is helpful to look at the dependencies among the measures and understand whether the assumptions apply to your situation. But looking at the code is not ideal: Having different repositories or technologies makes spotting the many interactions and dependencies very difficult.

These measures, whether qualitative or quantitative, can be compared with industry trends or the project's own data to establish thresholds. Exceeding a threshold is an indicator of poor architectural health that could result in adding a new technical debt item to the registry or conducting additional analysis to confirm whether there is a risk.

The Case of Technical Debt in the Architecture of Phoebe

In Chapter 5, we looked at examples of strategies Team Phoebe employed to uncover debt. Phoebe started with an observed symptom of increasing defects and worked to get to the root cause. The first step was for the project manager to ask the developers, who pointed to the spaghetti code. Then a quality objective was elicited that set the context for examining the code. The team identified two technical debt items in the code: "Remove empty Java packages" and "Remove duplicate code."

Team Phoebe continues to monitor the system for symptoms, iterating through the steps of the technical debt analysis to see what additional information the architecture analysis will uncover. The team focuses on the following activities:

1. Understand the key business goals.
2. Identify key concerns/questions about the Phoebe system related to these business goals.
3. Define observable qualitative and quantitative criteria related to their questions and goals.
4. Select and apply one or more techniques or tools to analyze the software for the criteria defined.
5. Document the issues uncovered as technical debt items and add them to the registry.
6. Iterate through activities 2 to 5 as needed.

Team Phoebe plans to switch focus between code and design as issues are uncovered. Related issues in the code could lead to an overarching design issue. Issues in the architecture could point to hotspots worth analyzing in depth in the detailed design and code. When team members perform activity 4, they now have the three new techniques in their toolbox that we just described: ask the designers about the health of the system or a problem, examine the architecture, and examine the code to get insight into the architecture.

Understand Key Business Goals and Concerns/Questions

The key business goals were defined in the first iteration. One business goal driving the Phoebe project is "Create an easy-to-evolve product." The development team has already looked at this goal from a code perspective. Another related business goal is "Increase market share." There is growing concern over security breaches that are causing users to have lower confidence in the system. These breaches are another pain point and have been traced to security-related bugs such as a crash due to an out-of-bounds number. The developers discuss possible solutions. One offers, "We could just fend off out-of-bounds numbers near the crash site, or we can dig deeper to find out how this is happening."

Another developer notes, "Time permitting, I'm inclined to want to know the root cause. My sense is that if we patch it here, it will pop up somewhere else later."

Given the urgency of the issue, the team makes a quick fix and closes the issue, only to have to open it again. A team member records the rationale as a comment in the ticket associated with this issue: "Hmm…reopening. The test case crashes a debug build. I have confirmed that the original source code does crash the production build, so there must be multiple things going on here."

The team members turn their attention to the two business goals to understand technical debt in the architecture. The architecture design is now the artifact of interest to complement the concerns and questions about the source code. The team tries to answer more questions: How do we understand whether or not the design is messy? How is the architecture related to the areas of the code that are messy?

The team also tries to answer questions about the new attribute of concern: How much time have we spent patching the code in response to the breaches? Do these patches get to the root cause, or is there an underlying design issue? Are the breaches related to each other? Are they related to the messy design?

Define the Architecture Measurement Criteria

From the questions and concerns, team members define the criteria that provide a measure of the architecture to see if they are on track to achieve key business goals. Maintainability, as defined in the ISO/IEC 25010 standard, comes from a collection of subattributes: modularity, reusability, analyzability, modifiability, and testability.

Modifiability may be related to adding new capability, a change in technology (which we call the *technological gap* in the technical debt landscape), or the evolution of other operational quality attribute scenarios to handle more stringent demands as the system grows over time. Modifiability can be cast as a quality attribute scenario:

> The developer wishes to change the user interface by modifying the code at design time. The modifications are made and unit tested, with no side effects within three hours.

The response measure of the modifiability scenario (no side effects within three hours) can be analyzed in terms of system quality measures (properties of the software development process) such as cost-effectiveness in avoiding or eliminating defects. Or it might be analyzed in terms of design measurement criteria (properties of the architecture) such as module design complexity, module independence, complexity in interrelations, and concern scope, overlap, and diffusion. The latter overlaps with the code measurement criteria that the team employed earlier. Some code grouping constructs such as classes and packages can give insight into design elements.

Next Team Phoebe defines the criteria for security. Security as defined in the quality standard ISO/IEC 25010 is a collection of subattributes including confidentiality, integrity, non-repudiation, authenticity, and accountability. Security can be cast as a quality attribute scenario:

> An attacker from a remote location attempts to access private data during normal operations of the system. The system maintains an audit trail, the data is kept private, and the source of the tampering is identified.

The response measure of the security scenario (how much data is vulnerable to a particular attack; how much time passes before an attack is detected) can be analyzed in terms of system quality measures (properties of the software development process) such as cost-effectiveness in avoiding or eliminating vulnerabilities. Or it might be analyzed in terms of design measurement criteria (properties of the architecture) such as adherence to secure design standards. If the response measure cannot be met, then the ease of supporting this requirement can be considered a growth scenario that has implications for modifiability.

Select and Apply Architecture Analysis Techniques to Get to the Artifact

Realizing that there is only so much that can be learned from the code, the Phoebe project brings in an external team to conduct an architecture evaluation. During the evaluation, all the business goals and quality attributes are considered to discover risks and trade-offs throughout the system. Qualitative reviews of the design uncover risks to meeting Team Phoebe's quality attribute goals. The analysis from the architecture review shows what business drivers are at risk.

The Phoebe team identified risks related to the adapter/gateway separation of their architecture. Their architecture concept had a common *gateway* component that presents a transaction service interface to the integrated enterprise systems and applications while hiding the external resource interface. It also had a customized *adapter* component to bridge the incompatible interfaces of the enterprise systems and applications. The concerns they identified included the following:

- The reference implementation for the adapter is not production quality.
- The gateway has evolved to include operations not needed by all users and defers some common operations, such as audit and logging, to the adapter. These dependencies make it difficult, if not impossible, to separate the two components.
- For use cases that require interaction with multiple endpoints, an application can orchestrate multiple transactions itself or allow the gateway to handle the request fan-out. The responsibilities of the gateway and adapter are not well defined, leading to implementations with different performance, robustness, security, and other quality-of-service characteristics.

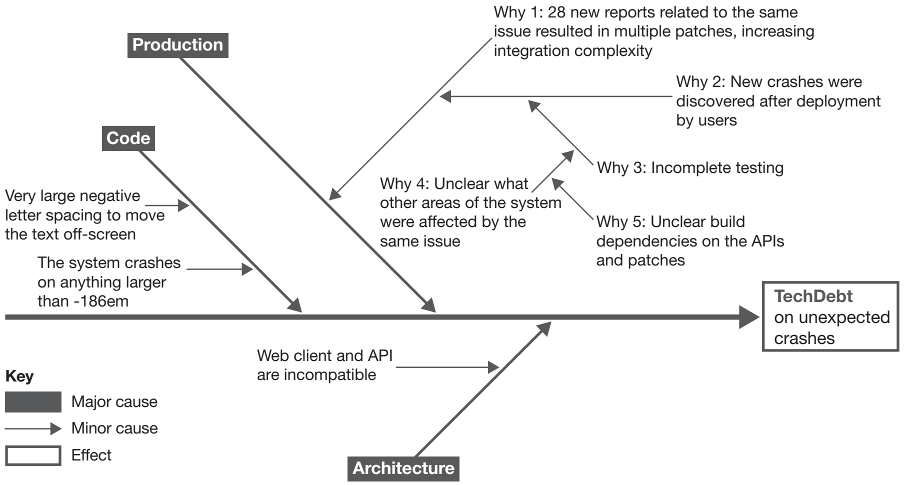

Production
Code
Architecture
Why 1: 28 new reports related to the same issue resulted in multiple patches, increasing integration complexity
Why 2: New crashes were discovered after deployment by users
Why 3: Incomplete testing
Why 4: Unclear what other areas of the system were affected by the same issue
Why 5: Unclear build dependencies on the APIs and patches
Very large negative letter spacing to move the text off-screen
The system crashes on anything larger than -186em
Web client and API are incompatible
TechDebt on unexpected crashes
Key
Major cause
Minor cause
Effect

Figure 6.1 *Exploring the cause-and-effect relationships underlying the problem of unexpected crashes*

The design review also provided details about the problem of crashes. They weren't caused by a local problem, as the developers suspected. Tracing interconnections in the Phoebe design revealed a dependency on an external library maintained by another group. Figure 6.1 shows these causes and their effect as a fishbone diagram (also called an Ishikawa diagram).

To complement the architecture review, the team used automated software analysis measures to uncover the fact that the system is becoming difficult to maintain. Risks from the review provided context for scoping the code analysis to gain insight into the design by measuring the complexity and change propagation of the architecture. A number of methods, classes, and packages demonstrated high complexity, measured with a combination of metrics such as method and class size, cyclomatic complexity, and fan-in and fan-out. The analysis also showed a rise in system cyclicity.

Document the Technical Debt Items

As team members apply the methods and tools, they document the analysis outcome as the starting point of comparison with the project's key concerns. The sample technical debt item in Table 6.1 shows analysis of both the design and the code to get insight into the maintainability of the architecture.

Table 6.1 *Techdebt on architectural choices*

Name	Phoebe #420: Locked-in architectural choices in adapter/gateway separation
Summary	Phoebe is based on service-oriented architecture design principles and web service interfaces. The architecture is broken down into two sections: a gateway and an adapter. The gateway handles communication between different organizations' health information systems. The adapter adapts the gateway to an organization's backend system. Phoebe has evolved to reflect a more complete architecture but was stymied by increasing complexity and locking in architectural choices that later proved limiting.
Consequences	Immediate benefit is implementing a solution within schedule constraints. Review of the feature matrix by each release shows that the project is struggling to add new functionality. Most releases are preoccupied with dealing with integration, security, and other quality-related issues. Long-term cost is predicted to be slowing velocity due to accumulation of debt that requires extra work to add more capabilities. Analysis of the artifact indicates the risks and areas of rework: • A major risk theme surfaced by the architecture review is adapter/gateway separation. • Static analysis of code provides insight into areas of the architecture of major complexity and change propagation based on dependency information.
Remediation approach	Better define responsibilities of the adapter and gateway; refactor to better separate the two components.
Reporter/assignee	Design team.

As shown in Table 6.2, the team also documented a technical debt item to record the design issue at the root of the unexpected crashes.

Table 6.2 *Techdebt on unexpected crashes*

Name	Phoebe #421: Screen spacing creates unexpected crashes due to API incompatibility.
Summary	The source code uses a very large negative letter-spacing in an attempt to move the text offscreen. The system handles up to –186 em fine but crashes on anything larger. A similar issue #432 was fixed with a patch, but there was another similar report. Time permitting, I'm inclined to want to know what the root cause of this is. My sense is that if we patch it here, it will pop up somewhere else later.
Consequences	We already had 28 reports from seven clients. And it definitely leaves the software vulnerable. Finding the root cause of this crash can be timely.
Remediation approach	The quick and easy solution is to write a patch, but we already seem to have done this twice. The responsible thing to do is to first find the root cause and create a patch at the source. I have a feeling the external web client and our software have an API incompatibility. The course of action I would take is to: • Verify where the root of this is. • See if we can fix it on our side, but I am tempted to believe the external web client team needs to fix it, so we would need to negotiate.
Reporter/assignee	I need to discuss this with Brant as the fix may be more involved than we think.

Service the Debt

After selecting analysis criteria, conducting the analysis, and inspecting the design, Team Phoebe has a handful of technical debt items. Some of these items pertain to code conformance issues. The code does not conform to the architecture. Understanding the architecture as designed provides the context for refactoring the as-is architecture embodied in the code. Other items pertain to design verification issues. The architecture does not support the business goals and needs to be re-architected, which in turn triggers corresponding changes in the code. We will say more on this topic in Chapter 9, "Servicing the Technical Debt."

What Can You Do Today?

It is important to communicate the goals and the design approaches chosen for the project with your team. These activities may be useful:

- Get clarity on the yardstick by which you measure design and architecture, at a minimum by clearly identifying architecturally significant requirements, including their measurable, testable completion criteria.
- Review the architecture. If it is not documented, glean insights from team knowledge, source code, and the issues being tracked.
- Make reviewing architectural concerns a regular part of iteration/sprint reviews and retrospectives.
- Use your knowledge of architectural risk to guide automated analysis of the source code.
- When fixing a defect or adding a new feature request, look beyond the immediate implementation to see if there are longer-term design issues leading to technical debt.

Look for the presence of technical debt during these activities and respond by including them in the technical debt registry.

For Further Reading

If you are not familiar with the concept of software architecture, start with the Wikipedia definition (2018). Ian Gorton's book *Essential Software Architecture* (2006) is a fast and easy read, and if you are coming from an agile perspective, Simon Brown's *Software Architecture for Developers* (2018) is for you. For a more thorough treatment of the topic of software architecture, our colleagues at the Software Engineering Institute have evolved over 10 years the reference opus *Software Architecture in Practice* (Bass et al. 2012). This book also provides more information about quality attribute scenarios and architectural tactics. *Just Enough Software Architecture: A Risk-Driven Approach* focuses on the risks that prevent development progress (Fairbanks 2010). A continuous architecting approach to system development and sustainment is essential for avoiding unintentional technical debt.

The Architecture Tradeoff Analysis Method (ATAM) is a method for evaluating software architectures relative to quality attribute goals to expose architectural risks that could potentially inhibit an organization's achievement of its business goals (Clements et al. 2001). Knodel and Naab (2016) introduce architecture evaluations in the context of continuous architecting. *Designing Software Architectures*, by Humberto Cervantes and Rick Kazman (2016), provides more information about lightweight analysis techniques during design, and the appendix contains tactics questionnaires.

An architecture description language (ADL) could be used to describe a software architecture. The appendix of *Documenting Software Architectures: Views and Beyond* by Clements and colleagues (2011) provides an overview of AADL, SysML, and UML. These three ADLs are representative of the range of formal or semiformal descriptive languages, textual and/or graphics languages, and associated tools. The benefit of using an ADL is the support it provides in design and analysis activities.

Design Rules introduces design structure matrices to understand dependencies between product elements and how to decouple them for effective evolution (Baldwin & Clark 2000). Researchers and tool vendors have applied the ideas from this book to software to provide tool support. For example, Tornhill (2018) and Kazman and colleagues (2015) put such an analysis in the context of technical debt.

Ford, Parsons, and Kua (2017) introduce the idea of an executable "fitness function" in their book *Building Evolutionary Architectures*. This is one way of trying to spot architectural debt when it occurs, though only some kinds of architectural constraint are amenable to being checked like this.

Chapter 7

Technical Debt and Production

In this chapter, we explore technical debt that arises in the process of putting software in the production environment and into the hands of its end users. This process includes the build and integration, testing, deployment, and release aspects of software development. These release activities involve essential software artifacts that can cause technical debt or that can be subject to technical debt themselves.

We explain how to recognize technical debt in the infrastructure of the release activities. We again illustrate our lightweight analysis technique to assess technical debt in such artifacts and to ensure traceability so that misalignments between these artifacts do not introduce technical debt. We focus on automated testing, continuous integration, and deployment aspects.

Beyond the Architecture, the Design, and the Code

In Chapters 5, “Technical Debt and the Source Code,” and 6, “Technical Debt and Architecture,” we looked at how technical debt appears in the traditional activities we usually associate with software development: code, design, and architecture. But technical debt can also appear in the steps that deliver the software to its end users, wherever there are code and structural considerations.

How does software get into the hands of users? Industry practices vary widely. Software can be embedded in another physical product, such as your TV monitor; it can be delivered to individual computers or devices, such as your laptop or cell phone; or it can run in large operations centers using the SaaS paradigm (Software as a Service).

SaaS has been undergoing a big transformation lately, evolving from software *development* teams throwing candidate software releases over the wall to *operations* teams, to more integrated approaches, nicknamed *DevOps*, for *development* and *operations*.

Just as processes used by the software industry vary, so does the terminology they use to describe this tail-end process. We will begin by defining a few terms.

We use the term *release* for the part of the process that brings completed code to a running, operational system in the hands of its end users. So, release is the process that brings the software into *production*, as shown in Figure 7.1.

The release part of the process encompasses the following four activities:

1. **Build:** Creating the executable software
2. **System test:** Validating that the software is ready for use
3. **Deployment:** Bringing the software (and data) to the place of use
4. **Turn it on:** Making the software operational

Release occurs at various time increments—from years, to months, to weeks, to more or less continuously. Continuous integration and deployment enable developers to push a code change through the release activities immediately into production.

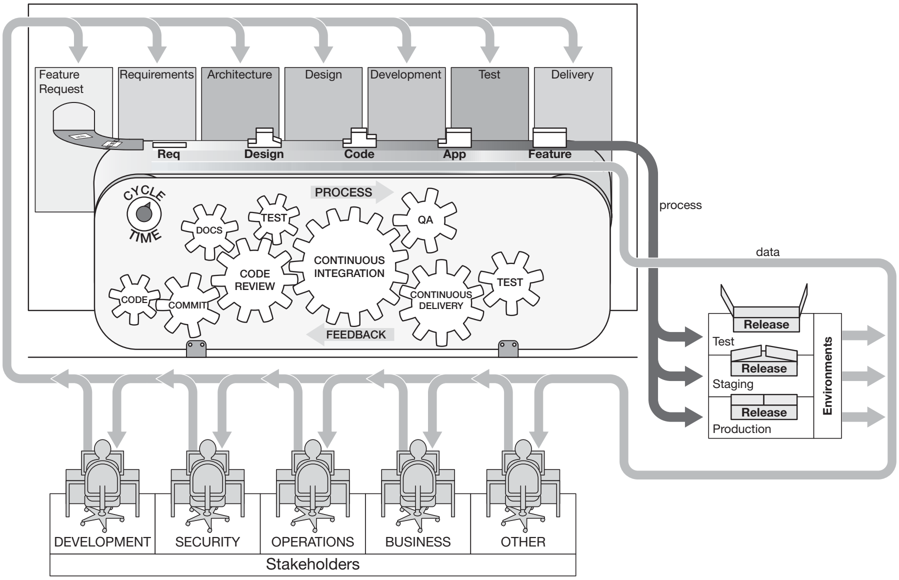


Figure 7.1 *Release pipeline*

Continuous integration involves rebuilding the software when any significant change occurs and is practiced throughout the industry. Continuous integration involves integrating artifacts on every change, notifying the team immediately of success or failure, and requiring issues to be fixed before moving forward. *Continuous deployment* involves deploying changes into production as soon as possible, to make the software operational.

These activities are supported by tools, and there are many good ones to choose from today. These tools are usually driven by programs called *scripts* that are written in various languages, including operating system shell scripts.

Because of all this script-driven automation, technical debt in production is not very different conceptually from technical debt in code or software architecture. You can think about your infrastructure as a complicated codebase. *Infrastructure as code* refers to the process of managing the IT infrastructure through automated processes. All assets are versioned, scripted, and shared, where possible.

All the three project examples we've been examining (the three moons of Saturn) have a significant production element: They have an operations team. Atlas uses a DevOps approach, Phoebe is an agile shop, and Tethys uses a more traditional method. Here is an example from the Phoebe project about its build automation tool, called Make, which automatically builds executable programs from source code:

> Make's dependency calculation is taking 20% of the time for an incremental build, and we need to speed things up. We had been able to make some small performance improvements in the past but are no longer able to continue with such workarounds.

So, the Phoebe project has both the software that is the product, which is "shipped," and the software that helps build the software, which is the product. Previous chapters discuss Phoebe's software product; here we consider the software that builds the product. For shrink-wrapped software (what is in the box or the installer you download) or embedded software, the distinction between the software that is the product and the software that helps build the product is pretty obvious. For SaaS, it is a little trickier. But this software still impacts what the end user experiences.

There are several important differences between software products and software used in production:

- **Different tools:** The production phase often employs a chain of several tools, using plugins to refine and specialize them; this is an extension of the traditional build tool chain of compilation/linking and not a fundamentally different animal.

- **Different languages:** The languages used in production software are often not known for their legibility and maintainability.
- **Different people doing operations or different maturities of the personnel involved:** These differences can lead to cultural issues; some organizations do not treat the infrastructure code as first-class software.
- **Different degrees of automation:** Often some manual steps need to be performed.

And above all, greater degree of difficulty to test before putting software into production. This was easy in the shrink-wrapped context, but much harder in an SaaS environment.

In developing the codebase, the language often provides some conceptual integrity, especially when using well-known frameworks. For example, you may have all your application code written in JavaScript, using the MEAN stack (MongoDB, Explorer.js, Angular.js, Node.js), and manage it in Git repositories. In contrast, the tools in the release process may be more scattered and may have evolved organically (as opposed to being well designed), sometimes in the hands of people with a lesser degree of software engineering sophistication. Version control may have a 1990s feel, or it may not be done at all.

The field of infrastructure and its code is not as mature as the software development field, despite the availability of many tools to assist in the process, so it is more difficult to have a top-down design. There is little in the way of standard practices, guidelines, or education available. In large systems, the tool chain will also contain elements to monitor the behavior or health of the running system, collect metrics to allow reflection on the system, react automatically to specific misbehavior, and guide future evolution.

Build and Integration Debt

Technical debt in build and integration appears in two ways:

- **Imperfect or suboptimal design and coding of the build scripts themselves:** Build scripts are, in effect, code, sometimes supported by special code embedded in the application under development.
- **Misalignment between the build dependencies and the actual code:** As the software rapidly evolves, new components may not be backward compatible.

Principle 7: All Code Matters!

All code matters: the code that goes into unit tests, the code you decide not to include in this release but will include in the next release, the build scripts that deploy the software, the generated code that allows you to take advantage of frameworks, and the script that automates running the test, integrates the functionality, and deploys it to the production environment for release. Dependencies between these artifacts become barriers rather than enablers when the system is being refactored or evolved.

Developers write code within their development environment that might be deployed on specially provisioned virtual machines or simply on their own computers. At the various stages of deployment, the test, staging, and production environments are provisioned to match the expected infrastructure configuration. These environments are independent, prone to change, and easily manipulated. Without careful management, they will diverge.

Technical debt appears within each of these environments and as a consequence of misalignment among them. One example is a bug found in production that cannot be reproduced in development. Even rolling back development code to the production version doesn't allow it to manifest. It may be an issue with updated packages or the operating system in the development environment. We will consider how technical debt accumulates in each of the build and integration, testing, and deployment aspects of production.

Build automation keeps builds consistent. Build scripts build the product but are often used for other tasks, such as running unit tests, packaging binaries, and generating project documentation, test coverage reports, and internal release notes. The absence of build infrastructure is a source of technical debt because it increases the setup time when new developers join the team or a new machine is installed.

Automation and continuous integration require an investment in infrastructure and the ramp-up time to design, develop, and use the continuous integration server. Building such infrastructure involves architecting and implementation and hence can introduce technical debt—much as described in Chapters 5 and 6.

Doesn't Embracing DevOps Help *Eliminate* Technical Debt?

Well, yes and no! We have explained that production infrastructure is not immune to technical debt. And in the context of continuous integration and continuous deployment, DevOps is positioned as an enabler to reduce, if not eliminate, technical debt. There is definitely some truth to this claim. Manual analysis, testing, and integration are not only error prone and incomplete but also have issues of scalability, reusability, and correctness as the software evolves. Automation helps standardize the artifact submission process and provides consistent results, improving integration consistency and speed. Continuous integration goes a step further, using a build server to integrate artifacts on every change and enforce quality standards.

The process of adopting automation for these practices and moving to DevOps helps teams uncover technical debt and inconsistent processes and assess what they can eliminate with automation enabled by DevOps.

There are many positive outcomes in automating the production pipeline with a DevOps model including the following:

- Higher productivity due to automating routine, error-prone, and time-consuming tasks
- Incorporating analysis tools
- Quicker delivery and faster resolution of problems
- A continuous software delivery environment
- Stable operating environments
- Improved communication
- A more stable product (eventually)

As the capabilities of automating testing, integration, and conformance tools improve, DevOps will deliver on its promise of achieving faster and more reliable software delivery. However, it is not a magic solution to resolving your technical debt. In this chapter, we have discussed different ways that technical debt can exist in the production environment. But in addition to those, there are kinds of technical debt that an automated pipeline will not be able to detect and have a solution for. For example, architecture decisions can be tough to automate and monitor. A DevOps pipeline, no matter how smooth the automation process, will not tell you whether you have selected the UI framework that best fits the user interaction you need to implement. While you can often push patches and upgrades to run-time, these can actually accumulate technical debt rather than fix the problem at its source. An automated tool chain will not help you detect major re-architecting that may need to be done, as the software will continue to work.

DevOps is one of the practices in improving software development quality and timeliness and can be an effective approach for intentional management of technical debt. However, DevOps does not replace a holistic technical debt management practice. There is no free lunch!

Testing Debt

Technical debt in testing appears in three ways:

- **Imperfection or suboptimal design and coding of tests:** Test suites are, in effect, code, and they are sometimes supported by special code embedded in the application under development. Large sets of automated tests may not have a clear purpose; when they fail, something is probably wrong, but it is unclear what artifacts contributed to the failure and why.
- **Misalignment between the tests and the actual code:** As software evolves rapidly, new tests may be missing or may test an older interpretation of the requirements. Very fine-grained tests introduced early in development, especially with mockup software, become a nightmare to maintain as they create complex webs of code around the production code; one small change might, for example, cause 60 tests to fail.
- **Challenges of SaaS contexts:** Development, testing, and production environments can become misaligned. If your developers use version X, your continuous integration system version Y, and your production servers version Z, then your tests aren't testing the right thing, and your developers might not know about it. Or code that worked perfectly during development might fail when deployed to the test infrastructure.

Here is an example of a technical debt item from the Tethys project, whose developers have grown frustrated because multiple tests have a similar purpose, and other tests override each other:

> Page_test_runner and benchmark_runner_test are duplicates. The duplication is a consequence of trying to expedite a request by the controls team. When the actual test code got written, they did not realize that the test got dubbed. These test codes should be merged and refactored, as the code also includes a page setup test that can be overwritten.

This example demonstrates that an organization needs a deliberate strategy for managing technical debt not only for development but also for testing and production. Tests need to be designed and aligned to their purpose, implemented following sound coding practices, and executed in alignment with the functionality and attributes they are meant to test.

Infrastructure Debt

Technical debt in deployment appears in two ways:

- **In the structure of the operational system:** This may include the lack of "observability" of the system, which may be referred to as *monitoring debt*.
- **In scripts:** This may include scripts that enact the deployment of the code, the data, and the updates on the operational system.

This is *infrastructure debt* hiding in infrastructure code. A task that must be performed manually, again and again, by the staff on the operational system is an example of such infrastructure debt. The operations team must continuously pay the recurring interest, while dealing with significant risks.

The lack of verification of deployment scripts is a source of technical debt. It is essential to check that the scripts are compatible with the architecture to avoid inconsistencies between development, testing, and production environments and to minimize risk.

The Case of Technical Debt in the Production of Phoebe

Previous chapters describe how Team Phoebe identified technical debt items in the code and the architecture. Let's continue with the Phoebe example to see what additional information the team can uncover by analyzing the infrastructure. Treating infrastructure as code, team members again follow the steps of technical debt analysis (described in Chapter 4, "Recognizing Technical Debt"). In the first iteration, they define key business goals. The development team has already looked at pain points related to two of the business goals—"Create an easy-to-evolve product" and "Increase market share"—from code and architecture perspectives. Another related business goal is "Reduce time to market." There is growing concern that velocity keeps dropping. It takes forever to implement even a simple change and test it. The developers turn their attention to improving the build time and test infrastructure.

Improve the Build Time

As Team Phoebe evaluates possible solutions for improving performance for Make's dependency calculation, team members consider the consequences of technical debt. Should the team continue to incur more debt, pay it off at the expense of some performance, or make a partial payment on the debt while still meeting their performance goal?

The sample technical debt item in Table 7.1 shows the team's analysis of the build infrastructure to get insight into the maintainability of the build and integration scripts.

Improve the Test Infrastructure

Team Phoebe would also like to reuse new test helper modules for a legacy test framework. While the development team has been migrating its integration tests to

Table 7.1 *Techdebt on build infrastructure*

Name	Phoebe #500: Improve build time
Summary	Make's dependency calculation is taking 20% of the time for an incremental build. The team is considering three alternative solutions and the trade-offs involved in incurring technical debt to optimize performance.
Consequences	Slowing build time and turnaround time for feedback.
Remediation approach	I tried three approaches: 1. extra_cflags on the cc compiler command, separate precompile header command 2. override cflags per rule to add -include for source files and -x for precompiled header files 3. base_cflags with normal flags, set cflags to $base_cflags -include, override it with $base_cflags -x for precompiled header files 1 is messy but fast, 2 is cleaner but a lot slower (due to cflags per object file), and 3 is cleanish and fast.
Reporter/assignee	Build team

the new test framework, there have been two parallel test helpers to maintain. This code duplication is a source of technical debt and requires team members to make changes in two places. They often forget, which leads to unintended drift between the two frameworks.

The remediation approach the team is taking allows the legacy test framework to reuse the new test framework's helper modules, which are essentially a cleaned-up port (better documentation, linted, obvious errors fixed). The sample technical debt item in Table 7.2 shows analysis of the test infrastructure to get insight into the maintainability of the test framework.

Table 7.2 *Techdebt on test infrastructure*

Name	Phoebe #501: Improve test infrastructure
Summary	While DevTeam has been migrating its integration tests to the new test framework, there have been two parallel test helpers to maintain.
Consequences	This code duplication is a source of technical debt and requires team members to make changes in two places. They often forget, which leads to unintended drift between the two frameworks.
Remediation approach	Reuse the new test framework's helper modules. The goal isn't 100% code reuse between the old and new test frameworks but 80–90%. The test methods that remain are here for three reasons: • When ported to the new test framework, they were refactored into different modules, and legacy tests need to be updated to load new modules. • Navigating the page in the old test framework is hacky and has been cleaned up in the new test framework, so the tests won't ever share implementations. • Subtle refactoring changes make the new implementation fail certain tests. This test failure should be followed up by using the old implementation and then refactoring once all tests have been migrated.
Reporter/assignee	DevTeam developers

Service the Production Debt

After inspecting the infrastructure, team members have added a few more technical debt items to the registry. These items pertain to the build and test infrastructure. They will need to consider trade-offs with other system properties and understand the consequences of partial payment of the debt. They also need to examine the legacy test framework and assess how the debt will change over time as the developers migrate tests to the new framework. We will say more on these topics in Chapter 9, "Servicing the Technical Debt."

What Can You Do Today?

At this point, it is important to identify the software that helps you build the software that is the product and start treating it as first-class code. These activities may be useful at this stage:

- Put it under configuration management.
- Document it (see Chapter 12, "Avoiding Unintentional Debt").
- Integrate its operation into your overall development process.
- Architect for ease of deployment, observability, and automated processes.
- Analyze the code and the design of the infrastructure for the presence of technical debt as you would for the product.

You need to identify steps that require manual intervention, that are error prone, and that could be automated. You also need to integrate elements and tools to observe software in development and operation (static analysis, monitoring, logging) to obtain information about its architecture health and run-time behavior that can inform priorities and guide future decisions.

For Further Reading

Andrew Clay Shafer (2010) came up with the concept of infrastructure debt hiding in infrastructure code, and *Infrastructure as Code* is actually the title of a book by Kief Morris (2016).

In their novel *The Phoenix Project*, Gene Kim and coauthors (2013) give a great illustration of the impact of technical debt on infrastructure and the notion of DevOps. In *Site Reliability Engineering*, Beyer and colleagues (2016) emphasize that thoughtless automation in the production and testing infrastructure will create more problems than it solves.

To learn more about DevOps, you can find many resources that provide practical guidance. The *DevOps Adoption Playbook*, by Sanjeev Sharma (2017), provides guidance on implementing DevOps in large organizations. The *DevOps Handbook*, by Gene Kim and Patrick Debois (2016), is another such industrial reference on what is good DevOps. For a software architect's perspective on the DevOps movement, see the book *DevOps* by Len Bass and colleagues (2016).

On documentation, especially documenting the allocation views of the architecture, see Simon Brown (2018) and Clements and colleagues (2011). The deployment and install views describe the mapping of architecture elements to the computing platform and production environment.

PART III

Deciding What Technical Debt to Fix

This page intentionally left blank

Chapter 8

Costing the Technical Debt

Despite the adjective *technical*, technical debt is ultimately an economic issue. Your strategy for managing it revolves around how many resources to spend and when to pay back the debt. In this chapter, we shine an economic spotlight on technical debt items to reveal the information you need to make decisions about how to service your debt. We explain how to estimate the remediation cost and the resulting cost savings when you reduce the recurring interest.

Shining an Economic Spotlight on Technical Debt

In general, the key driver for making decisions about a software project is maximizing value while minimizing costs. This is also the case with technical debt and the decisions you make about whether to do something about it, as well as how much and when. At some point in the life of a software product, you must be able to calculate *costs* of doing whatever you need to do with technical debt items. This involves computing or estimating the cost to carry and to eliminate the debt.

Here is how Team Atlas weighed the value of reducing recurring interest against the cost of paying the debt:

> Running a static checker, the Atlas team found 34 clones of a certain piece of code. They noticed the issue because an inconsistent modification to only 32 of the clones had triggered a bug that was hard to find. The proposed refactoring to service the debt consists of encapsulating the logic of these 12 lines of code in a single method and then replacing all the 34 clones by an invocation of this method. The cost? About one hour. Oh, wait, they probably need to do some regression testing to validate that they have not affected the

logic of the whole system. Oh, wait, they do not have unit and regression tests for several of the affected locations. Adding the tests, running the tests in the "before" version, and then running the regression tests will add another two hours.

The bottom line is that eliminating this technical debt item requires one day of work. The team determines that the benefit of reducing the debt by tracking down these bugs is worth the cost of the fix.

If you take a technical debt item from your registry, you can estimate the *total effort* involved in eliminating the associated technical debt. The associated debt is what we have called the *current principal*, and it includes the cost of changing the code or design option and all the accruing interest—that is, undoing the modifications and workarounds that piled up on the not-quite-right code, design, or production infrastructure.

Let us assume that you have to break apart a class into two distinct classes. If you've waited very long to repay this technical debt item, a lot of other code has been written that depends on the class. You will need to revisit and modify all these places in the code. And these modifications may have further consequences on other dependent code. An original naive estimate of requiring one day to reorganize the class rapidly grows to three days of work to deal with all the ramifications of accrued interest.

A simple return on investment (ROI) calculation for debt reduction compares the benefit of reducing the recurring interest with the cost of paying the current principal and accruing interest (remediation cost).

In the technical debt timeline we introduced in Chapter 2, "What Is Technical Debt?" you need to know the cost of the technical debt you have in your system and understand when you will reach the *tipping* point (see Figure 8.1). Refining technical debt items will enable you to estimate the cost and prioritize actions to take.

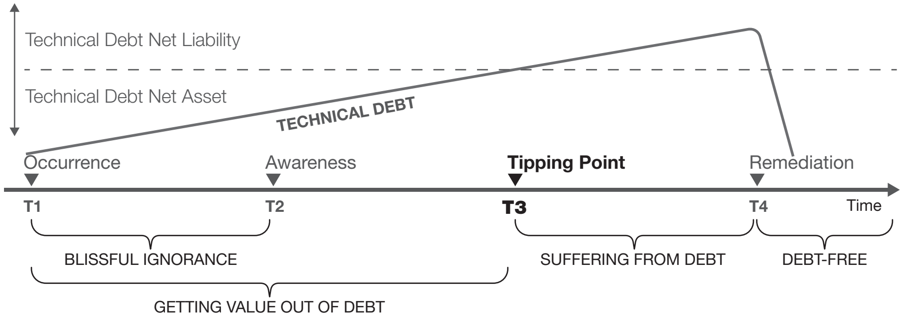


Figure 8.1 *Reaching the tipping point*

Shining an economic spotlight on the technical debt items involves doing the following:

- Refining the technical debt description to identify the impacted and related software artifacts (code, tests, build scripts, and so on)
- Using the artifacts to calculate the cost of remediation
- Using the artifacts and consequences to calculate the recurring interest

Let's look more closely at the technical factors of principal and interest.

Refine the Technical Debt Description

When you or your manager, client, or CTO asks, "How much technical debt do we have?" the real questions are "How much would it cost to fix the issues now?" "What benefit would it have?" and "How much impact would it have if we didn't fix it now?" These questions about the future do not consider only the code, the architecture, or the production infrastructure; they assume that when the issue is fixed, all the associated tasks will be fixed. Any calculation of technical debt should assess it from such a holistic perspective.

Holistically automating the entire decision-making and resource allocation process is not possible, and automated static analysis tools cannot make these calculations for you. You can identify issues and make design trade-offs for fixing them, but assessing issues as technical debt and managing them as such requires building an end-to-end economic argument. Sometimes the fix is a trivial code change, even if you find the issue during an architecture analysis; other times remediation requires a re-architecting effort, even though the technical debt item was discovered through static code analysis.

Looking back at the Phoebe agile shop that we studied in Chapter 6, "Technical Debt and Architecture," the large negative-letter spacing issue was initially addressed with a patch, completed with two hours of a developer's time. That is when the debt started accumulating because the team initially failed to assess the architecture, in addition to the code, until one of the developers sensed that the system required a more involved analysis and fix.

So, one of the developers entered a technical debt description, an excerpt of which is shown here (see Chapter 6 for the full description):

Name	Phoebe #421: Screen spacing creates unexpected crashes due to API incompatibility.
Summary	The source code uses a very large negative letter spacing in an attempt to move the text offscreen. The system handles up to −186 em fine but crashes on anything larger.

Table 8.1 *What and where is the debt?*

Name	Phoebe #421: Screen spacing creates unexpected crashes due to API incompatibility
Affected components	UIsetuplayer, transparency layer, UILogic
Affected code	Isolated to the frame renderers the text is fed into
Dependent components	LayoutTests, external web component
Other analysis data	40 reports from 7 clients in 10 days

This is a critical issue that impacts multiple fronts: The software crashes leave the users frustrated, and the negative spacing causes integer overflow, which creates a security vulnerability and leaves the software brittle. The developers have patched the code, but they have not yet identified the root cause, leading them to believe the fix may be more complicated.

Table 8.1 shows the refinement of the technical debt description and identifies the concrete software artifacts related to it. Although Team Phoebe recorded the technical debt item during an architecture analysis, team members now know that the code, architecture, and production infrastructure are related to each other, and it is not always easy to tease them apart. One or the other may be the starting point of the analysis and may trigger reflection on other related aspects. When team members plan remediation, they need to consider how changes to one artifact could impact the others.

The driving analysis questions guide the developers in tracing symptoms such as crashes to the codebase. (Recall the questions for the “Increase market share” business goal in Chapter 5, “Technical Debt and the Source Code.”) For example, in the context of this particular issue, the team sees that the negative out-of-bounds problem creates a crash in three components. The team identifies the cause in the frame renderers and an internal dependent component. Team members recognize through architectural thinking that this error is being injected externally to several different areas in the code; hence, they need to understand the influence of the external component on the code to develop an appropriate remediation approach.

This refinement exercise guides developers in assembling the analysis of code, architecture, and production that we discussed in Chapters 5, 6, and 7, “Technical Debt and Production.” As teams become more sophisticated, they can link their development environments and autofill some of these fields with the relevant information. The goal is not to trigger analysis paralysis but to be aware of the added costs related to accruing interest and to make the changes so the system is production ready. We strongly underscore the benefits of a robust integrated configuration management and version control environment. You can use these tools to refine your technical debt items and manage them throughout the software development lifecycle.

Calculate the Cost of Remediation

Table 8.2 lists the activities for remediating the debt from source code through unit tests for Phoebe's unexpected crashes. The cost of fixing the quality problems comprises the current principal and the accrued interest. The team adjusts these costs by an uncertainty factor and the cost to test the fix. Accounting for uncertainty provides team members a mechanism to express their confidence in their ability to localize changes, so they can determine how much they need to account for unexpected ripple effects.

Team Phoebe analyzed the issue and decided to write a wrapper to remediate the problem. The developers refined the technical debt description to reflect this decision:

Remediation approach	We could just fend off negative numbers near the crash site, or we can dig deeper and find out how this –10000 is happening. Code changes are trivial but distributed in the classes. That was the mistake made with the patches. With Brant, we decided to write a wrapper around the external web component.

Table 8.2 *Cost of remediation*

	Remove Technical Debt	Retrofit Other Areas of Software
Architecture (design and analysis)	The real cost was finding the dependency to the external web component and the existing patches.	In a later release, we can just remove the patches. Trivial.
Code	Write a wrapper around the external web component. We estimate one-half day.	A bunch of debug code needs to be cleaned, though, like GetLastError() following the UIFrame calls. These should now return null, too. Maybe spend another half day to ensure cleanup.
Infrastructure (test)	Write new test for the wrapper. One-half day.	Run the previous tests to ensure that the fix and removed patches resolve the problem. One-half day.
Uncertainty multiplier for propagating issues	Hopefully none as we were able to localize the fix.	

The artifacts that constitute the debt (the architecture, code, and infrastructure) identified in Table 8.1 provide input into the cost of remediation.

With this information, the cost of the remediation becomes clearer, but Team Phoebe needs a little more information to weigh this decision against the benefit of removing the recurring interest. Remember that team members already patched the software several times in the local sites and then figured out that this was not a routine bug but rather technical debt. So now they have to consider the trade-off between the quick solution of patches (recurring interest) and fixing the software properly (paying off the principal).

Calculate the Recurring Interest

This next step is to calculate the resulting benefit of reducing the recurring interest. This requires understanding the nature of future changes and putting some quasi values around them. Table 8.3 shows the factors involved. You need to know the consequences of continuing to carry the existing debt so you can weigh them against the consequences of your strategy to remediate the debt (which may or may not pay off the entire principal). The symptom measures and the artifacts identified in Tables 8.1 and 8.2 provide the information to assess the consequences of continuing to create patches compared to the proposed remediation.

To make a simple calculation of the benefit, you look at only the cost saved from no longer carrying the debt. This assumes that you completely pay off the principal and eliminate the debt, so there will be no recurring interest. You know the cost of living with the debt up to this point. You might base predictions about future costs on an extrapolation of the past debt, the rework cost of anticipated changes to the

Table 8.3 *Trade-offs of change*

	Carrying Debt	**Remediating Debt**
Cost of future change	Medium: Each patch costs one-half day.	Low
Frequency (adjust for accumulating interest)	High: Many sites use this renderer, so they will also experience the issue requiring the patch.	High: Many sites use this renderer; they expect a smooth and secure experience.
Uncertainty (adjust for potential propagating issues)	High: Without rework, each new function is messier and messier.	Low

system, or the growing gap between the state of the software and good software engineering practices.

To make a more nuanced calculation of the benefit, subtract the recurring interest of your remediation strategy from the cost of carrying the debt. This difference becomes more important when you are contemplating a partial fix—reducing but not eliminating the recurring interest.

Compare Cost and Benefit

Determining the ROI of the proposed remediation involves comparing the cost of remediation with the benefit of the reduced interest. Team Phoebe refined the description of the techdebt in their backlog to include the ROI of the remediation approach:

Remediation approach	ROI of remediation: High. The remediation cost is paid back in reduced developer effort to patch and rework the software almost immediately. There is less time spent considering the already implemented multiple local patches at crash sites. Even if we get only three or four more of these issues and continue with the patch-locally approach, which we will, the architectural fix pays off.

Comparing strategies for managing technical debt depends on understanding both the probability and impact of future change.

In this example, we have explained how to refine the technical debt description to include economic information by using consecutive analysis steps. In reality, this is an iterative process throughout development. Filling in the details of where the debt is found (refer to Table 8.1) can and should happen as developers discover or take on the debt. They can supplement their efforts with tool-supported analysis as well as architecture reviews. This supplemental analysis (for which we discuss several techniques in Chapters 5, 6, and 7) should happen for issues that require substantial changes. This analysis can be another task on the backlog with the goal of providing further details.

Remediation requires a team to generate possible solutions and evaluate the alternatives and cost. Some items are simple fixes with known costs and can easily happen through local refactorings. Other items involve substantial changes and require a design exercise and understanding of trade-offs—and maybe even several dedicated iterations. These changes will likely resolve multiple technical debt items and other issues that make them worth the time and effort. Finally, capturing the information

Principle 8: Technical Debt Has No Absolute Measure—Neither for Principal Nor Interest

A mortgage, which is an example of financial debt, has defined principal and interest from the beginning. Technical debt does not; it is tied to the current state of the system, and principal and interest are tied to your intentions to change the system in the future. Most attempts to give an absolute meaning to the value or cost of technical debt will fail, but they do give some general indication of where to look for the debt.

Your system may have a potential for technical debt, but it will have actual technical debt only if you have to evolve it. You may also decide to walk away from your technical debt, and you can't do this with your mortgage! So, technical debt has a value and cost relative to a point in time, based on potential evolution scenarios. Its value and cost change as the system evolves and expectations for future evolution change.

about the cost savings of the change requires knowledge of the business context as well as team skill sets.

In the case of Phoebe, the backlog prioritization approach resulted in the team getting tunnel vision, even after fixing the same issue a number of times. The situation—with the customer reports and the potential impact of the vulnerability—became so disruptive that the team had no choice but to take an approach based on design analysis rather than continuing the one-off patches. The information we present about the artifacts in an organized way here happened as organic and opportunistic discussions and team members' comments on the open issue in their project issue tracker. An explicit focus on a technical debt item will signal that at some point, the team may need to go through a trade-off analysis to remediate the debt. Not all debt has equal impact. Some debt can be serviced locally during routine refactoring exercises. A team will have to do more analysis when paying back debt requires architecture-level changes.

With an analysis approach that costs all the impacted software development artifacts and considers associated uncertainty with and without remediation, you should be able to identify the technical debt items that have high cost consequences today or that have low risk but high return in fixes and then allocate them to your releases. However, software development is rarely so simple.

Technical Debt: More Than Simply Dirty Code
by Michael Keeling

Six months after releasing our software to the world, the WIRE team was in trouble. Customer support requests were increasing. Four AM pages were firing far too frequently. Our velocity slowed to a crawl. As if this weren't bad enough, it was also quickly becoming apparent that pieces of our architecture were not going to be able to handle the next major batch of features. On the road to our first release, we purposefully, and occasionally accidentally, accepted technical debt so we could ship our software sooner. Now we were feeling the consequences of that debt. The question the team now faced was "What are you going to do about it?"

Our first actions were purely tactical. We needed to create breathing room to relieve pain and buy time to hatch a more strategically focused repayment plan. We started by focusing on the greatest pain points in our system. We fixed our monitoring dashboards, logging, and debugging tools so we could diagnose problems faster. We reevaluated our alerting strategies to remove superfluous pages. We fixed the most disruptive bugs. After a few months of hard work, the pain lessened, people started getting a full night's sleep again, and morale began a slow ascent from its all-time low.

Things were looking better, but we had still not addressed the root cause of our woes. The team's velocity was still slow, and pieces of our architecture still were not prepared to take us where our roadmap showed we needed to go next. As time moved on, the business landscape also started to shift under our feet. Components we thought were clean and well designed began unraveling as our users found new and interesting ways to flex the system.

(continued)

We needed a strategic plan for not only repaying our technical debt but also managing it better in the future if we were to continue delivering software. To create this plan, we hosted a simple workshop. The software engineers kicked off the workshop by showing where potential technical debt might live in our architecture. One afternoon we measured potential debt in our system by examining various code quality metrics, such as churn, conceptual design integrity, and defect data. Most of the metrics came from readily available sources such as git logs. Next, our product manager shared the roadmap for the next three to six months. Starting with the highest-priority roadmap items, we worked together to determine which parts of the architecture would need to be touched and how much effort might be required so we could deliver each roadmap item.

By the end of the workshop, we had a technical debt repayment plan. Surprisingly, some of the worst-quality code would not be scheduled for cleanup for another six months or more. As it turned out, though the potential technical debt in these components was high, they required few changes over the next three to six months. Through our analysis we also learned that it would be impossible to deliver some potentially important features beyond the six-month time horizon if we didn't start repaying some technical debt right away.

Perhaps the greatest outcome of the workshop was that engineering and product management had a shared strategic vision for paying down technical debt. The conversation about debt shifted. Instead of complaining about bad code or making excuses for slow velocity, the team now talked about positioning the architecture so it could successfully carry us into the future. In addition, discussions about technical debt had elevated from pain to prevention. Our analysis made the metaphor of "debt" concrete in a way everyone could understand. We added new stories to our backlog to prevent us from taking on more technical debt accidentally and adjusted the process to have more meaningful discussions about design decisions that introduced potential technical debt.

Reflecting on this experience, I think the WIRE team was successful for a few important reasons. First, we relied on data instead of gut feelings to find pockets of potential debt, and we found simple, reliable ways to measure code quality. Second, we collaborated with product management to understand how our software system might need to change instead of simply fixing the worst code. Finally, the team's mindset shifted away from thinking of technical debt as something always to avoid toward using technical debt responsibly to help us move faster.

Manage Technical Debt Items Collectively

In the larger system of Tethys, team members waited two years to thoroughly analyze their technical debt. Even though they followed the technical debt identification process to filter nonessential issues, they still came up with a list of about 200 tech-debt items. This became rapidly overwhelming. The amount of debt they have estimated far exceeds the available resources for several iterations. It may even exceed the amount of effort expended so far to develop the system!

Bringing in an army of contractors or student summer interns to knock down your technical debt is not going to resolve it. Making a large number of scattered changes can introduce new defects and new items of technical debt. And the debt at the architectural level is hard to parcel out into small bursts of activities. Refactoring at this structural level may halt development for several weeks.

Development teams clearly need additional criteria to decide what to do about a long list of technical debt items. A naive strategy of repaying them all one by one does not scale up. More often than not, the team will have to treat the technical debt items in reference to each other as they think about possible ways to restructure the system to service the debt and the implications over time.

The problem is even more complicated. You cannot treat technical debt in isolation from satisfying new requirements, adding new features, and other evolutions of the system, and you cannot separate treating technical debt from correcting defects and flaws in the system because they compete for the same resources: developers. Remember the four categories of items you have on your backlog: features, defects, architecture and infrastructure, and technical debt items (see the sidebar "What Color Is Your Backlog?" in Chapter 4, "Recognizing Technical Debt").

Figure 8.2 shows a backlog of product issues consisting of desired features, architectural elements, defect fixes, and technical debt items. As team members groom the backlog, they identify and refine the top-priority issues, which become candidates for tasks in the next release.

The decisions in prioritizing the backlog are challenging because of all the *hidden dependencies*. Some features depend on elements of technical debt. Similarly, features may depend on some architectural element. And the same is true for defects: Their resolution may depend on some missing structural element, or they may be linked to some technical debt items.

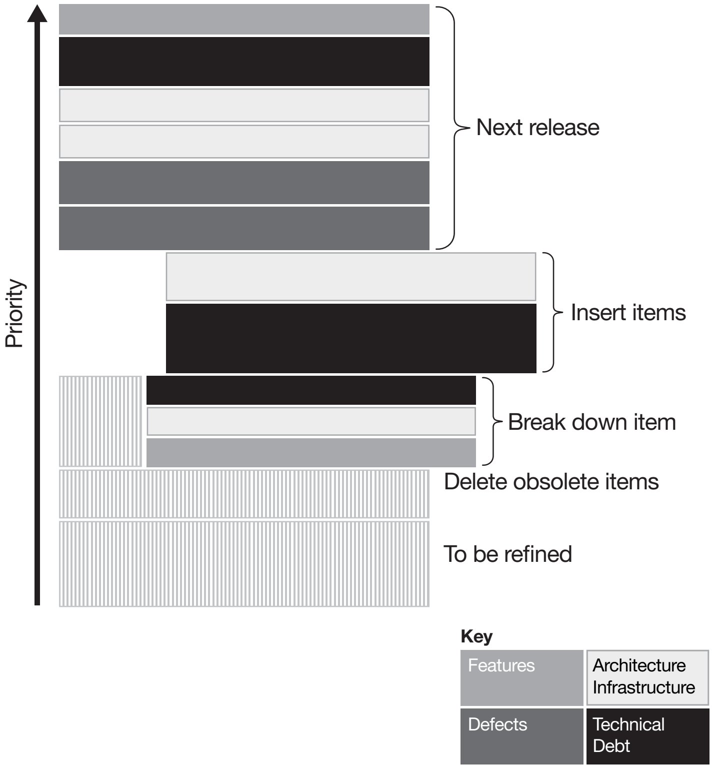


Figure 8.2 *Grooming the product backlog*

To determine whether to include a technical debt item or postpone it for subsequent iterations while grooming backlog items, consider the answers to these questions:

- In what ways are technical debt items that are related to development of features visible to the customer?
- What architectural decisions have an impact on technical debt?
- What defects can be traced back to the consequences of a technical debt item?
- Are any technical debt items blocking progress?
- Do any technical debt items need further refinement?

If the answers reveal that a technical debt item has dependencies with other issues on the backlog, then it becomes a higher priority to consider remediating it when working in this code for other reasons. How backlog issues concentrate in areas of the code can be another factor in setting priorities. For example, code with high defect rates or code that has been modified a lot in the past (assuming that the same will be true in the future) could be symptomatic of technical debt and thus worth prioritizing. If a technical debt item has no dependencies with other issues on the backlog, it has potential to incur cost, though not for the moment or the foreseeable future.

There is a clear distinction between approaches that help you identify technical debt and those that help you manage technical debt. We have already discussed tools that help you assess your code. These approaches, such as SQALE or OMG's Automated Technical Debt Measures, create assessments of technical debt reduction based on fixing all these issues and assigning an effort estimate to each line of code to fix. These techniques can help you detect technical debt. However, they cannot help you manage your technical debt throughout the software development lifecycle. They are only part of the toolbox.

We will take up the challenge of servicing the debt in Chapter 9, "Servicing the Technical Debt," where we explain how to use information about costing debt to resolve your technical debt during release planning and the delivery cycle.

What Can You Do Today?

At this point, it is important to calculate the technical factors of principal and interest in the artifacts that they trace to. These activities may be useful at this stage:

- Refine technical debt descriptions to identify the software artifacts at the root of the debt and any other components affected by the debt. This will help you calculate costs.
- For identified technical debt items, estimate not only the cost to pay them (in effort: person-days or person-weeks) but also the cost to not pay them (how much will it slow current progress?). In making your estimates, include the overall uncertainty associated with the cost of future change.
- If you are not able to provide an actual cost, use a "T-shirt sizing" strategy: XS, S, M, L, XL.

At the very least, you need to describe qualitatively the impact of any technical debt item on productivity or quality.

For Further Reading

Cost can be measured very accurately *post facto*: Just ask your accounting division to tally all the development costs, direct and indirect. For cost estimation, software developers have moved away from using a direct monetary value. They use various proxies—that is, point-based systems. Over the years we have seen *function points* in the 1970s (Albrecht & Gaffney 1983; ISO 20926:2009), *object points* in the 1980s (Boehm et al. 2000), *use-case points* in the 1990s (Alan et al. 2012), *story points* in the 2000s (Cohn 2006), and associated methods and tools to assist in making estimates (Grenning 2002). These approaches come with specific ways to calibrate what a "point" actually represents, so you can be consistent inside a development project or—better—across multiple development projects in a given organization. When the actual costs are known, it is also possible to use a cost-per-point or dollar-per-point factor to help with planning.

Automated tools that have rules for finding code quality issues often have a default value and a remediation strategy with an associated cost that you can tailor. Value is often qualitative, such as high, medium, and low or the top-ten rules in a given category. Costs for these more localized fixes are on the order of minutes or hours, computed as a constant function per fix, an increasing function based on complexity, or a base function for common infrastructure plus a cost per fix.

The Agile Alliance Technical Debt Initiative has developed guidelines for executives, managers, and developers. In particular, it proposes the *Agile Alliance Debt Analysis Model* (A2DAM), which gives directions on how to estimate remediation costs for known code quality violations (Fayolle et al. 2018).

Chapter 9

Servicing the Technical Debt

Organizations are often perplexed by questions like “Do we have too much debt?” “Which technical debt items should we remove?” and “Which project should we close out because of technical debt?” In this chapter, we examine the paths you can take to service your technical debt: eliminate it, reduce it, or mitigate it. Using the technical debt descriptions in the registry and the technical debt timeline, we offer an approach to help you decide which technical debt items you should service first and which you can put off for later.

Weighing the Costs and Benefits

At this stage, you have a registry of technical debt items. You know what consequences they could have on the future of your software project in terms of recurring interest and remediation cost as you consider whether to carry or pay the debt.

What should you do about your debt? You might be tempted to answer, “Repay the technical debt items, all of them, one by one and as fast as possible to avoid interest.” This is what you might do with ever-increasing credit card debt. However, there are other options to consider in managing your overall financial health. While it is prudent to eliminate the most severe credit card debt, you would manage a car loan or home mortgage differently. You might be more concerned with cash flow and want to continue making an affordable fixed monthly car payment. Or you might be optimizing your overall financial portfolio. Early in the life of a mortgage, the majority of the payment goes to interest, so you might make additional payments that apply to principal. Later in the life of the mortgage, you might redirect those additional payments to other investments since the majority of payment goes to principal and the incentive to reduce interest is gone. Your goals and the context of your situation will influence your decision.

Table 9.1 *Costs and benefits of servicing technical debt*

Cost	Benefit
Current principal and accruing interest	Reduced recurring interest
Opportunity cost of delaying features	Reduced risk liability

In software development, you have these and even more options to manage and grow your technical wealth. Unlike with financial debt, you may not have to repay any of your technical debt, or you might have to repay some but not all of it. You can choose.

In deciding what to do, you need to consider the business case for debt reduction, including the costs and corresponding benefits (see Table 9.1). You should evaluate the benefit of reducing risk liability and recurring interest. You should also estimate the opportunity cost of delaying the delivery of new features as you remediate the debt and the cost of paying the current principal and accruing interest. Conversely, the business case to incur or carry debt swaps these factors. The benefit becomes the cost savings of carrying the debt along with earlier feature delivery. The cost becomes the recurring interest and increased liability.

Understanding the costs and benefits of carrying versus remediating the debt will give you a sense of where you are on the technical debt timeline introduced in Chapter 2, "What Is Technical Debt?" Have you passed the *tipping point* so that the cost of interest has become greater than the benefit of incurring the debt in the first place? With the answer to that question, you can examine the technical debt items in your registry and determine which technical debt items you should *remediate* and which ones you can continue to live with. Weighing costs and benefits of the technical debt items in the registry will enable you to discuss and prioritize actions to take to decide how to remediate the debt (see Figure 9.1).

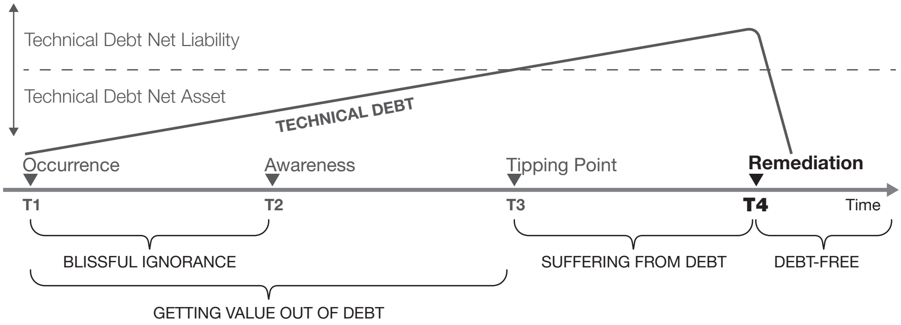


Figure 9.1 *Reaching the remediation point*

Risk Exposure and Opportunity Cost
by Eltjo R. Poort

A realistic business case for technical debt reduction is an important tool to put the risk and cost related to technical debt on the radar of the business stakeholders who can do something about it. On top of recurring maintenance and remediation costs, it should also include less obvious items such as risk exposure and opportunity cost related to a specific debt item.

Risk and opportunity costs often have more impact than the recurring maintenance and direct remediation costs. A technical debt item that might lead to severe security risks will normally be remediated quickly even if the cost of remediation outweighs the maintenance reduction. Think of outdated operating systems with known and unknown vulnerabilities that are no longer patched: Migrating to a new OS might be expensive, but you just cannot afford to risk a security breach. Conversely, an item that would at first sight make very much economic sense to remediate as soon as possible might still have to wait because you need the full development capacity to grasp an opportunity to beat the competition by creating some new functionality.

Risk Exposure

The proper way to calculate the total expected cost of uncertain failure is the well-known risk exposure formula: $E(S) = p(S) \times C(S)$, where $p(S)$ is the probability of failure scenario S occurring, and $C(S)$ is the cost incurred when S occurs. By summing up the risk exposure E over all possible failure scenarios S caused because of the technical debt, you come as close as statistically possible to an accurate prediction of the expected cost of failure. In practice, technical debt–related risk exposure can often be estimated only as an order of magnitude, but this level of estimation is often enough to make a business case.

I once encountered a situation in which a large transportation company was running some of its core business systems on ancient mini-computers. Spare parts were very hard to get, and the manufacturer had put severe limitations on its maintenance contract. The organization in question had a hard time making the business case for migrating the system to a modern, virtualized, blade-based solution: The cost of the old platforms was so low that the ROI for the migration looked negative. The risk of failure, however, was substantial: A single missing spare part could potentially break the company by disabling its core system for a few days. Including that risk exposure in the

(continued)

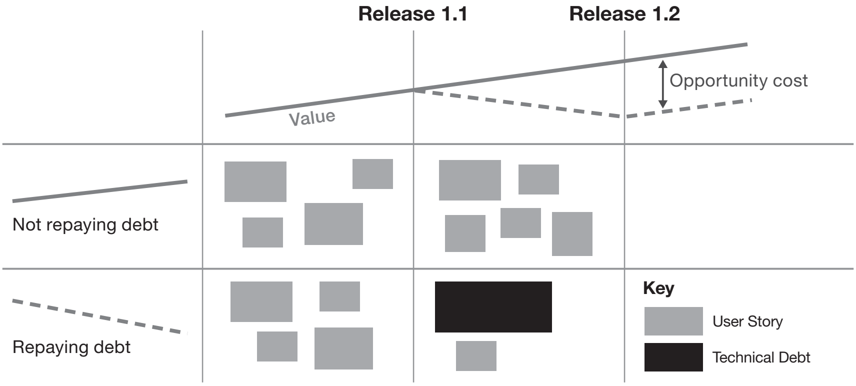
technical debt interest leads to a completely different business case; in this case, it pushed the company over the tipping point.

Opportunity Cost

The *New Oxford American Dictionary* defines *opportunity cost* as "the loss of potential gain from other alternatives when one alternative is chosen."

When a development team spends resources and time on reducing technical debt (upgrading, refactoring, repairing), the team will produce fewer end-user stories during that time. Opportunity cost represents the business value that those end-user stories would have yielded, as a way of accounting for the scarcity of the team's resources.

The literal term *opportunity cost* is seldom heard during technical debt discussions, but it is often a major factor in deciding when to reduce the debt. Whenever a stakeholder (for example, a product manager) says something like, "Yes, we should do something about this debt, but we cannot afford to do it *now*," she is probably referring to the business features that end users are waiting for or that have been promised by a certain deadline. In other words, the opportunity cost of reducing the technical debt—the potential gain from the alternative of delivering the business features on time—is higher than the interest on the technical debt incurred during that period.



This diagram illustrates opportunity cost by comparing two scenarios: in Scenario 1, the technical debt is not paid back, and in Scenario 2, the debt is paid back in Release 1.2. The value curve at the top of the figure makes a little dip in Scenario 2 (dashed line), compared to the continued growth of Scenario 1. The figure shows that in Scenario 1, Release 1.2 introduces five new user stories, while in Scenario 2, there is time for only one user story because a team has spent the rest of the resources on reducing the technical debt. The gap between the dashed line and the solid line represents the opportunity cost of reducing the technical debt. (If you are wondering why the dashed line goes down in Release 1.2, even though the team has added a user story, I use the practical rule of thumb that existing business features in a solution are subject to some type of value decay due to growing expectations and demands from end users.)

A good example of opportunity cost in architectural technical debt reduction was presented to me by architects attending an agile architecture course as part of an exercise. In their organization, a team had been developing business process automation features for 4 years. The organization had kept track of the labor cost savings attributed to that automation effort, which amounted to 9 FTE (full-time equivalent positions) per year on average. The platform the software was running on was due for a major overhaul because it could not easily be made compliant with new European Commission regulations (most notably the EU General Data Protection Regulation). During the overhaul, the team would not be able to develop new features—which meant an opportunity cost equivalent to 9 FTE per year, or 0.75 FTE per month spent exclusively on the overhaul. This was a significant opportunity cost, but in the end, it was determined that the total benefits, including the significant reduction of the risk of noncompliance and reduced maintenance cost, outweighed the total cost (opportunity cost plus the cost of the overhaul itself).

The bottom line is that if you need to draw up a complete business case for servicing a piece of technical debt, make sure you include not only the more obvious principal and interest but also the risk and opportunity cost. This will help facilitate a rational discussion about the impact of running risks and delaying features and, therefore, help you put the decision in its business context.

Let's use the Phoebe project to demonstrate how to refine issues in the techdebt registry impacting business goals (Table 9.2). Recall that Team Phoebe's source code analysis surfaced more than 10,000 violations. The team identified two major technical debt items: Phoebe #346: "Remove duplicate code" and Phoebe #345: "Remove empty Java packages." Reviewing the architecture to complement the code analysis revealed a major risk in the design: Phoebe #420: "Locked-in architectural choices in adapter/gateway separation." Looking beyond the symptom of a reported defect

Table 9.2 *Phoebe techdebt registry*

Techdebt	Landscape	Remediation ROI
Phoebe #345: Remove empty Java packages	Code	Low
Phoebe #346: Remove duplicate code	Code	Medium
Phoebe #420: Locked-in architectural choices in Adapter/Gateway separation	Architecture	Medium
Phoebe #421: Screen spacing creates unexpected crashes due to API incompatibility	Architecture	High
Phoebe #500: Improve build time	Production	Medium
Phoebe #501: Improve test infrastructure	Production	Low

about crashes to the root cause in the design yielded another techdebt: Phoebe #421: "Screen spacing creates unexpected crashes due to API incompatibility." Finally, reviewing the production infrastructure surfaced technical debt items related to building and testing: Phoebe#500: "Improve build time" and Phoebe #501: "Improve test infrastructure." Chapter 8, "Costing the Technical Debt," provides the means to understand the costs associated with these issues for the architecture, code, and infrastructure, including testing.

However, there is still more work to be done in sorting out the backlog of issues collectively, setting priorities, weighing costs and benefits, and planning releases that allocate resources among new feature development, necessary design tasks, routine defects to take care of, and technical debt items.

Paths for Servicing Technical Debt

We've discussed how technical debt repayment might play a role in what you can deliver, considering a fixed expenditure budget. But the picture becomes more complicated as you establish a roadmap for a project and define the content of future releases. When deciding what to do in upcoming iterations, you need to consider all the items on your backlog of things yet to do and their dependencies, including technical debt items.

If you want a system to evolve in a certain direction—for example, by adding a new feature or service—you need to analyze which parts of the system will be affected by this evolution. If those parts of the system contain technical debt items, you need to look at the consequences of those items relative to the proposed evolution. Will they prevent or slow the proposed new development? If yes, then maybe you will need to repay them.

Any plan to repay some technical debt will affect the cost of possible scenarios for evolving the system, which in turn may affect your decision to repay some debt or evolve the system. The choice may well be decided by how much technical debt you must repay before proceeding. This is true for any change, whether it be a request to add a new feature, resolve a problem, invest in architecture, or even consider another technical debt item.

Here is an approach for developing a plan to manage your technical debt while you maintain and evolve your system:

1. Identify the parts of the system that will be affected by a change.
2. Determine whether technical debt items are associated with these parts of the system.
3. Identify the consequences of technical debt on this and possibly other changes.
4. Estimate the cost of the debt repayment and add it to the cost of the change.
5. Estimate the benefit of the debt repayment in enabling the development of this and possibly other changes. (This can be difficult to do!)

This approach is contingent upon having a good grasp of which areas in the system have more technical debt as well as a few maintenance and evolution scenarios to compare potential outcomes. In addition, this approach is most practical when technical debt signals problems with the design.

Remediating hundreds of little code-level “smells” and other code quality issues might involve allocating a fixed percentage of resources to servicing technical debt. This is analogous to adding a buffer of time within a sprint for fixing defects. A fixed percentage gives a team the discretion to deal with code quality issues while controlling spending. In cases of extreme debt, you might allocate an entire sprint or two to work on paying back technical debt.

Experienced teams consider aspects of evolution as they debate design options, backlog grooming, and technology change. Conducting these discussions explicitly for the technical debt items will improve a team’s understanding of the consequences and help members make decisions based on the benefit gained by fixing the related technical debt items.

There are a number of decision points:

- Determining whether the debt is potential or actual
- Deciding to fix or not as new features are developed iteratively
- Deciding whether it is time to mitigate the risk by remediating the debt completely
- Declaring victory by writing off the debt
- Declaring bankruptcy

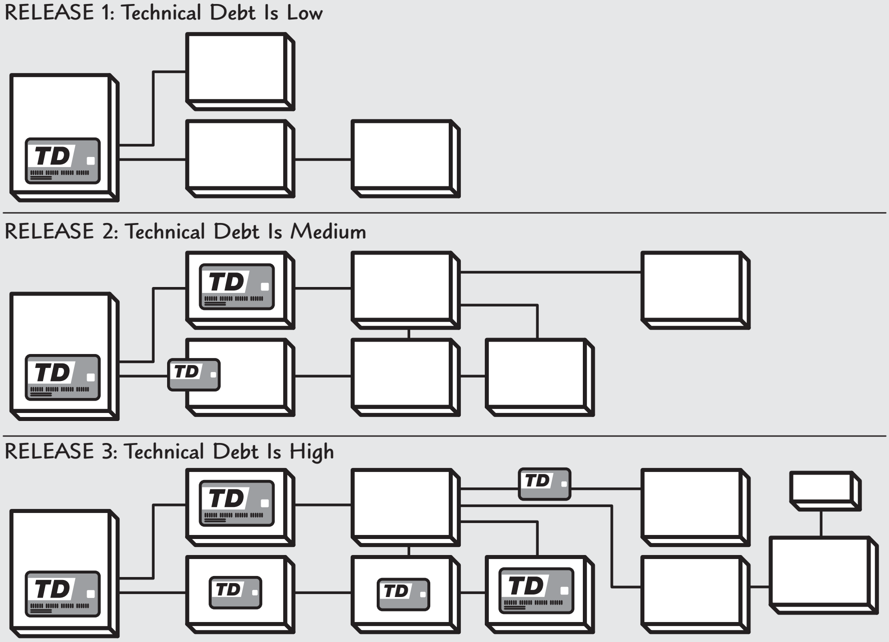
As your system reaches different points on the technical debt timeline, you will need to revisit whether the path you have selected is continuing to serve you in servicing your technical debt effectively. Let us look at these decision points along the way in more detail.

Is the Debt Potential or Actual?

If you have been thinking of a fixed-rate mortgage as a typical example of financial debt, then this is where the technical debt metaphor breaks down a bit. Your mortgage has defined principal and interest from the outset, and it's included in the paperwork you signed at the bank. Technical debt does not have defined principal and interest from the outset; it is tied to the current state of the system, and the current principal and interest are tied to your intentions for future changes. You might have a *potential* for technical debt, but it will be *actual* technical debt only if you have to evolve your system. You might also decide to walk away from your technical debt by walking away from your system.

The first step for any of the paths you can take is to determine whether the debt is potential or actual. If there is technical debt in parts of the system that do not need to evolve and do not have unintended business impact, you can just ignore it for now. In other words, the (long) list of technical debt items represents only potential debt. The actual debt at any point in time depends on how the system will evolve in the future. The more certain you are about the future evolution and the probability of change, the more confidently you can identify the actual debt and the payback strategies.

Principle 9: Technical Debt Depends on the Future Evolution of the System


RELEASE 1: Technical Debt Is Low
TD
RELEASE 2: Technical Debt Is Medium
TD
TD
TD
RELEASE 3: Technical Debt Is High
TD
TD
TD
TD
TD
TD

The value of debt as a strategic investment and its cost of remediation depend on the changes to be made on the system from now on. It is because of this principle that technical debt assessment and management are not one-time activities. They are strategic software management approaches that you should incorporate as ongoing activities for the development and sustainment of all software-reliant systems.

Should You Work on Debt, Feature Delivery, or Both?

Technical debt is an attribute of the state of a system at some point in time. While you may have identified technical debt items, you cannot associate to them any meaningful metric of cost until you look into the future and consider dependencies—dependencies among the technical debt items and dependencies of future features on them.

Team Atlas has some experience with this type of thinking:

> The Atlas project met their goal of time to market with a successful product launch. There is growing demand to evolve the system by adding a new feature. Designing the feature points out some dependency on an element of technical debt in the system.

The value of the feature is the same, regardless of how it is implemented. But the project may incur a different cost based on whether the team eliminates or mitigates the technical debt. In this case, the system had some *potential debt*, but because it is affected by a prospective evolution, it now has *actual debt*. Team members can determine the cost of repaying the debt and the cost of carrying each technical debt item, as described in Chapter 8. They should include the opportunity cost and risk liability in the decision to mitigate business risk.

The decision is determined by the cost–benefit trade-off of servicing the debt. The cost to fix includes the cost of repaying the technical debt and the opportunity cost of delaying features. The benefit includes the reduced recurring interest cost and reduced risk liability. You can compare these costs to see where the project is on the technical debt timeline and make a decision about repayment. Is the project still getting value from carrying the debt and the recurring interest, and is the risk liability still low? Or have you passed the tipping point, beyond which you are suffering from technical debt? If the cost to fix is reasonable for the given benefit, then it is sensible to proceed with the fix. If not, the debt may have passed the point where it is feasible to remediate, and you should consider other paths, such as declaring bankruptcy.

It is rare that a system would evolve one feature at a time. A feature could also depend on multiple technical debt items; many features together would therefore implicate many technical debt items. When you potentially have multiple features with different values, you may be able to combine them in different “packages” over a release timeline and maximize value over time at a given cost.

There is some invisible internal value generated by refactoring areas of high technical debt in the system, if it makes future evolutions of the system easier and therefore cheaper. This forethought takes a longer view into the future of the system. In planning the future of a project, you should include technical debt in the economic reasoning of the business, not just the value of features visible to users. And you must look at the benefit and cost of repaying or *not* repaying the debt that affects the possible evolution scenarios.

Is It Time to Mitigate Risk?

If your system is already deployed in the field, your backlog very likely contains defects that must be fixed urgently. These have *negative* value because they decrease the system’s usability and they make customers unhappy, and customers or users today can express their unhappiness widely through social media, which may affect future sales. So, your business people want at least some of the defects fixed in the upcoming release, and they may even be ready to compromise with some of the features, maybe to the detriment of the architecture, which you know you really need but for which the business people may not see much value. The Atlas project demonstrates this problem:

> The Atlas project is slowing in productivity, and team members spend more and more time fixing defects at the expense of adding new features. The time to fix the defects is increasing as well.

Over time, risk liability may be the most important factor in the case for debt reduction. Instead of pulling a feature off the backlog, you might consider a defect, an architecture investment, or a technical debt item to be of higher priority to mitigate risk. That backlog item might depend on refactoring parts of the system to eliminate other elements of technical debt.

The path is influenced by the cost–benefit trade-off of servicing the debt. The total cost includes the cost of the backlog item, the cost of repaying the technical debt, and the opportunity cost of delaying features. This approach provides the benefit of eliminating the recurring interest cost of the technical debt and reducing risk liability. This also supports the goal of lowering the probability and impact of failure in a system that is growing in complexity.

Is It Time to Write Off the Debt?

In the case of debt *amnesty*, you write off the accrued technical debt and do not have to repay it. Contexts for such a decision include developing a throwaway prototype or concluding that a feature or product is a failure and no longer needed for various reasons, such as a lack of customer interest or value. The written-off technical debt item will continue to appear in your registry, but it doesn't matter anymore because there is no longer any recurring interest and hence no benefit in reducing its cost.

Is It Time to Declare Bankruptcy?

Bankruptcy happens when the part of the software system that contains the technical debt item is no longer viable to support future development, and a complete rewrite is needed. In some extreme cases, the whole system may have reached a point where a complete rewrite is the only option. Bankruptcy is justified when the cost of rewriting the system is lower than the cost of maintaining it (that is, the sum of recurring interests).

After restructuring the software and emerging from bankruptcy, a project may elect to monitor technical debt more closely by implementing checks and tests that must pass, and if they don't, they break the build.

The Release Pipeline

The different paths to servicing technical debt can be used individually and in combination to sort out and prioritize the product backlog and assign the backlog elements to iterations or releases. Figure 9.2 shows an example of a plan for the next three releases. Each release contains issues from the product backlog that are a mix of desired features, architectural elements, defect fixes, and technical debt remediation. There will be more detail for the first release or two and less detail further out into the future as part of a long-term release plan showing what features, improvements, defect fixes, and technical debt payments will be part of each release. The arrows depict dependencies both within and across releases. In many cases, the architecture must be developed ahead of the features and technical debt remediation that depend on it.

When you have the mechanisms in place to service technical debt, you can run some what-if scenarios to adjust the technical debt remediation timeline:

- What if we want to be debt free? What is the cost of paying off all the debt now? Or what is the cost of not letting the debt increase while we figure out how to pay it down?
- What if we postpone a payment? What is the cost of living with the debt, and how will the repayment increase for each subsequent release?
- What if we need to conserve cash flow now, but we also want to be debt free by the end of the development phase? Given that the product will go into sustainment in three years, how can we structure payments so we will be debt free by then and ready to shift resources to new product development rather than support unnecessary maintenance?

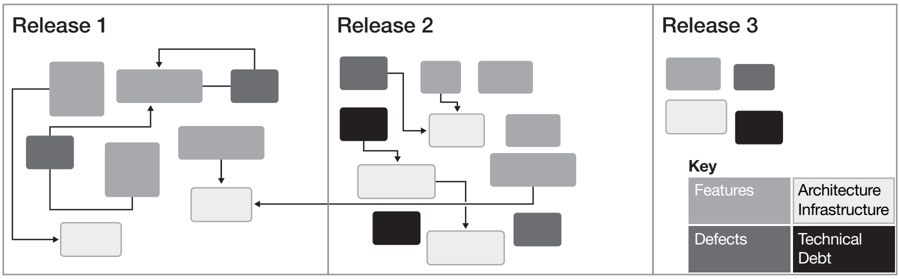


Figure 9.2 *Release planning*

While running what-if analysis on scenarios and comparing their implications will give you more information to make choices, you might still have the resources to select only a few of the debt repayment refactorings. How do you choose? Systems differ widely, and different quality attributes (security, fault tolerance, usability, performance, evolvability) matter more in different contexts than in others. This is especially true for major structural changes. It also applies to tackling some of the scattered code imperfection.

Returning to the Phoebe project, the team chose different payment strategies for the technical debt items in its registry. Team Phoebe prioritized fixing the defect in the user screen feature that was causing a very visible crash and had implications for security.

Team members implemented a patch to address their most immediate concern, and then they fixed the design in the next release to pay the debt fully. The issue with duplicate code concerned adapters and was related to the locked-in architecture choices in the adapter/gateway separation, so these issues were treated together.

To mitigate this major risk, team members better defined the responsibilities of the adapter and gateway and refactored the code to better separate the two components. They required new code to conform to the new design while they updated the existing duplicate code incrementally. Improving build time was the next issue to address.

A partial payment improved maintainability without sacrificing performance. Removing the empty Java packages was a more localized fix, and the team addressed this issue as part of the fixed buffer of time for dealing with defects and technical debt. The consequences of technical debt in the legacy test framework were diminishing as the framework was being used less and less, so the team deferred action on remediation for this issue.

The Business Case for Technical Debt as an Investment

We have sounded all the ills technical debt brings to software endeavors. However, when properly managed, technical debt can be a wise investment. This aligns with the basic financial metaphor of taking out a mortgage or borrowing money from a bank to start a new venture, which can be smart ways to build assets. When managed well, design choices bearing technical debt can be fruitful strategic investments and opportunities to investigate the market and learn new technologies if they are monitored for incurring interest.

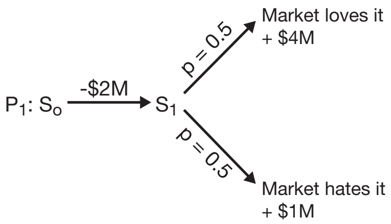
Let us illustrate the combination of paths a project team can take in its journey to make a wise software investment, this time using actual dollars and Atlas. As we map the team's choices, we will use the financial concept of net present value (NPV) to evaluate the implication of taking one path or another as Atlas decides whether to select another feature, mitigate risk, pay the debt, or declare bankruptcy. The NPV is the hypothetical value (estimated today) of an investment made today compared to its future returns—that is, its possible value in the future. We will use real options to model this. Real options include the decision—but not the obligation—for a business to pursue, defer, or abandon a capital investment.

Atlas envisions building a new software product, called alphaPlus, to put on the market. The initial plan is to invest $2 million to develop alphaPlus and ship Version 1. Market analysis shows a reasonable demand for alphaPlus, but in such a dynamic world, success is not guaranteed. The business analysts estimate that the product has a 50% chance of success, defined as the market loving it and bringing the company $4 million in revenue. There is also a 50% chance of a mediocre result: that the market hates it, and the return is only $1 million. As you can see in Figure 9.3, the NPV of the investment is $0.5 million. This is still positive overall, so the alphaPlus project is worth launching.

But what if Atlas were to take on an enormous amount of technical debt to bring a simpler prototype to market much earlier and use it to "test" the market? Then Atlas would invest only $1 million and take on $1 million of technical debt (mostly architectural, in limited scalability, in a single geographic locale, and with some internal ugliness).

If the market loves the product, then and only then would Atlas invest the other $1 million to complete alphaPlus. If the market hates it, Atlas will not pursue alphaPlus and will just walk away. As you can see in Figure 9.4, the NPV is now better: $1 million. So, taking on this technical debt is a more valuable investment!

But wait a minute. The decision is not this simple. Atlas will have to pay some interest on its technical debt. Estimating 50% interest on the debt to get to Version 2, Atlas must invest not the $1 million spared but $1.5 million. However, by releasing Version 1 much earlier, the company also increases its chance of success by



NPV (P₁) = -$2M + 0.5 x $4M + 0.5 x $1M = $0.5M

Figure 9.3 *NPV of alphaPlus*

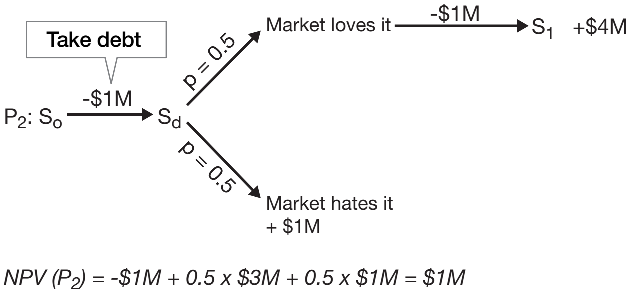


Figure 9.4 *NPV of alphaPlus with technical debt*

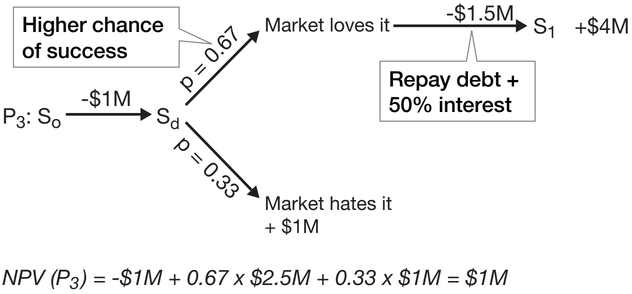


Figure 9.5 *NPV of alphaPlus with technical debt repayment*

leapfrogging the competition, from the business analysts' estimation of 50% to 67%. If the product is not successful, again, Atlas will not invest another dollar. The company will declare bankruptcy and walk away. As you can see in Figure 9.5, the NPV is still $1 million.

Even after the Atlas product has met success, the company is not bound to repay the technical debt and refactor the system for a clean Version 2. Team members still have the option of living with the debt and piling up more features. They can use the same reasoning again and again at each decision point in the future, based on what they know at that point in time (see Figure 9.6).

Is this "real options" strategy practical? Not quite yet. It might look good and might provide some rationale for making decisions. However, it requires many numbers about probabilities of events in the future, about which most software development organizations have no clue, so they have to make wild guesses. This approach could work in theory but not yet in practice. Nevertheless, the thought process and the act of building a simple decision tree can assist you in uncovering critical decision points on the technical debt timeline when you take on and plan to remediate debt. It also shows clearly that technical debt can be an asset—a good thing.

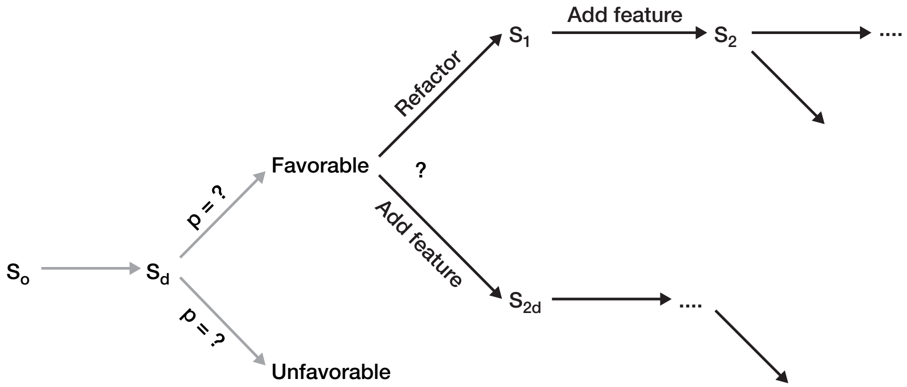


Figure 9.6 *Real options: The decision to add features or refactor*

What Can You Do Today?

At this point, it is important to incorporate the following basic rules of thumb into your decision making during iteration and release planning:

- Ensure sustainable team velocity by allocating time to servicing technical debt. Start by allocating 15% of your iteration budget. But know that there is no one-size-fits-all strategy. You might, for example, need to allocate a whole sprint to reducing technical debt; at other times, you may be able to tolerate more debt. Monitor your progress and learn from your experience.
- Put a context-dependent payment plan in place because repaying all debt, except in very small projects, is simply not feasible and also not the best use of resources.
- Show the value of technical debt reduction tasks by specifying how they support high-value change requests for new features or defect resolution.
- When choosing among refactorings, opt for the change that will offer more flexibility for the future and support more potential evolutions, when economically feasible.
- Prioritize technical debt items to fix by starting with the parts of your code that are the most actively modified. If a subsystem or module will not be modified as a result of a change scenario in the foreseeable future, do not fix any technical debt in it unless the change is a consequence of fixing the technical debt in a module it depends on.

- Recognize the time when the project is so far past the tipping point that future maintenance or evolution is no longer viable. That is the time to declare bankruptcy.
- Don't be afraid to take on technical debt strategically to your advantage when there is value in achieving a business objective and the servicing cost is predictable.

For Further Reading

Klaus Schmid first articulated the distinction between potential debt and actual debt (2013b), and then formally described it in mathematical terms (2013a). Eltjo Poort (2014, 2016) eloquently articulated the business case for technical debt reduction and architecture's role in risk management. Bankruptcy and amnesty have been identified by Eric Ries (2011) and Edith Tom and colleagues (2012). Highsmith (2010) showed the financial implication of technical debt manifesting itself as increased cost of change.

Understanding the value of software—in particular, the value of the design of software—is not trivial. Baldwin and Clark (2000) describe how modular designs create value in the form of future flexibility. Kruchten (2011) wrote about the value of software architecture in his blog.

Deciding whether to pay back technical debt is related to making solid software and business trade-offs. If you need a starter book for financial concepts applied to software, see Reifer's *Making the Software Business Case* (2001), which will help you work through them.

This page intentionally left blank

PART IV

Managing Technical Debt Tactically and Strategically

This page intentionally left blank

Chapter 10

What Causes Technical Debt?

Understanding the causes of technical debt is key to successfully controlling it. In this chapter, we examine the causes of technical debt that are common across many teams and organizations. These causes are associated with the business, change in context, development process, and people and team. Enabling development teams to clearly communicate about technical debt and selecting the right analysis techniques to focus on the concrete technical debt items that are accruing interest can empower teams to take action.

The Perplexing Art of Identifying What Causes Debt

When software professionals have a name for their pain, they are eager to talk about technical debt and look for causes. Getting to the root cause of technical debt can be a daunting task. Especially in long-lived systems, technical debt accumulates in several ways. Speculating about the project characteristics and organizational environment that contribute to technical debt very quickly becomes a frustrating and useless exercise for both software developers and managers.

Talking about their massive technical debt burden is almost like a therapy session for many software professionals. We have been there. We know how it feels! And we have heard the following proclamations from practitioners:

- "We have technical debt because our manager did not authorize us to migrate the system to the cloud!"
- "We have technical debt because the customers keep changing their minds!"

- “We have technical debt because we don’t know how to hire good developers!”
- “We have technical debt because we skip proper unit and automated testing when we are in a rush to finish a release!”
- “We have technical debt because we had no idea that we would need to scale up so soon!”

We know it feels good to get it out of your system, but as you talk about it, the debt keeps building. Talking about *who* failed is not sufficient to do anything about it. An important step toward getting ahead of technical debt is to understand the *realities and complexities of software development* that cause the debt. While understanding the causes will not provide a direct path to the precise location of the actual debt, it will provide a map of the environment and help you decide where to start looking more carefully. More importantly, it will help you eliminate future occurrences. Recall two of the principles we already introduced:

Principle 3: All systems have technical debt.
Principle 4: Technical debt must trace to the system.

Managing technical debt is not a one-time activity; it is an ongoing, integral part of the software development lifecycle. In this chapter, we discuss moving from a possibly speculative cause to a descriptive cause and the potential for injecting more technical debt into the system. This is the period leading up to the occurrence of technical debt in the timeline depicted in Figure 10.1. Software developers often confuse the *causes* that lead to accumulation of technical debt with the *system artifact* that has the debt and that should be fixed. You need to understand both the causes and the system artifacts, sometimes together and sometimes separately. We already talked about how to approach the most essential development artifacts of code, architecture, and production infrastructure in Chapters 5, “Technical Debt and the Source Code,” 6, “Technical Debt and Architecture,” and 7, “Technical Debt and Production.”

Having a good understanding of the causes of technical debt will later help you investigate its impact on the system and identify the areas of the system that will need to change. A suitable description of a cause will enable a team to articulate concrete actions to take, which include eliminating the cause, deciding how to analyze and tackle the debt, and possibly making broader changes in the organization and its processes. The ultimate goal of effective technical debt management is to reduce the unintended causes of technical debt and create an environment in which technical debt occurs mostly because it was taken on for a deliberate business need.

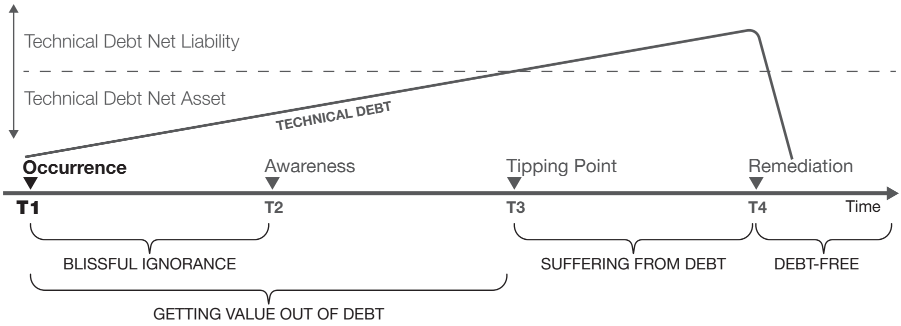


Figure 10.1 *The occurrence of technical debt on our timeline.*

The Roots of Technical Debt

In most horror stories about technical debt, the debt resulted from a combination of causes that built up to big problems. Unfortunately, we have seen far fewer examples of the debt that one consciously takes on, knowing all the terms of the debt, and having a strategy for paying it back, such as in buying a house.

Unintentional Debt

The causes of unintentional debt often confuse software development teams. These causes range from incompetence and reckless development behavior, to small inadvertent actions that result from lack of discipline and planning, to just not knowing any better. Most issues related to code and design quality have their roots in unintentional debt. Software developers and managers do not choose to introduce this technical debt; they do so accidentally. Symptoms emerge much later in the software development lifecycle, so the causes are hard to track down. Teams do not know when or how the debt began and, worse, they don't know how to get rid of it.

Intentional Debt

Managing technical debt with intention is a resource- and value-optimization activity, most often conducted to achieve a time-to-market goal. The causes of intentional debt are clear to the development team. This team has a business goal to satisfy within a short time frame: a release for a new customer, a feature to add that will let the product leapfrog the competition, a demo for prospective investors, or some other opportunities. Intentional debt involves careful and deliberate decision making by software developers and managers. At some point, they decide to introduce technical debt to achieve some objective. This intentional debt can be short term, when developers intend to rectify the technical debt within the next few releases. Or it could be long term.

When Ward Cunningham first used the debt metaphor in 1992, he implied intentionality. He used the example of shipping a system to a customer for the first time:

> Shipping first-time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite.…The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt.

What Causes Technical Debt?

A key aspect of any successful technical debt management strategy is to recognize that a cause contributes to the occurrence of technical debt in the system, and *the cause is not* the technical debt itself. To manage technical debt strategically, you must understand what led the technical debt to accumulate in the first place.

We sort the causes of technical debt into four major areas (see Figure 10.2):

- Nature of the business
- Change in context
- Development process
- People and team

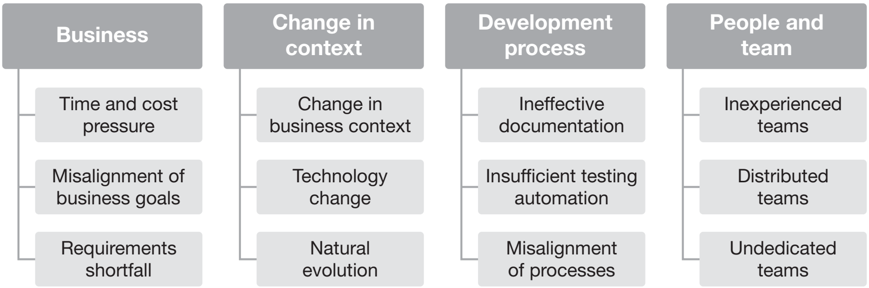


Figure 10.2 *Main causes of technical debt*

These are the causes we've identified, though there is great variability from project to project, and technical debt items are not evenly spread among the categories. And we do not claim to provide an exhaustive list, although we are pretty confident that we have included a significant number of common causes. When teams talk about causes, they often foreshadow, even if only in general, the actual technical debt items or symptoms as well. As we present sample scripts from Atlas, Phoebe, and Tethys, we will highlight the causes, symptoms, and potential technical debt items. The challenge these organizations faced initially was not always understanding what technical debt items resulted from these causes. You might find similar discussions in your organization as you start to implement technical debt management practices.

Causes Rooted in the Business

Business goals, requirements, resources, the amount of risk the organization is willing to take, and other business pressures all influence a product. Business problems cause technical problems that lead to technical debt.

Time and Cost Pressure

Development teams most often go into technical debt because of resource pressures, which usually boil down to time and developer costs in software development projects. A senior developer at Atlas described the nature of the time and schedule pressure in her project as customer-driven business pressure:

> Our customers and business leaders care only that we introduce new functionality to the users rapidly (**cause**). The customer gives almost no thought to what functionalities the users will need in the future or a vision for the end-game system. The customer's view is short-sighted and completely focused on tactical, immediate needs. So instead of taking the time to build a service layer for common functionality, we keep adding these services all over the system (**potential technical debt**).

Who would disagree that such constraints have demotivating and draining effects on a team? Defining success as delivering the required functionality within budget and schedule (fixing all three points on the classic quality triangle of scope, cost, and schedule) results in individual developer decisions that compromise intrinsic quality and introduce technical debt. The agile software development movement was born as a reaction to this problem and has succeeded in overcoming it to some extent by featuring communication with the customer early and often about value, quality, and constraints.

In this situation that Atlas is trying to manage, the developer recognized that the development artifact associated with technical debt is not the time and the cost pressures. It is the design decision to skip extracting common functionality to a service layer when developers are rushing against deadlines. And it is the flimsy software that results as functionality is instead repeated in multiple places.

Misalignment of Business Goals

Is your product solving the right problem for the business? A technical lead in the Phoebe project shared this example of technical debt in reference to a much-needed effort to change the technology stack to support portability:

> Poor business knowledge (**cause**) led to poor system design (**potential technical debt**), which led to poor user experience (**symptom**), which in turn led to a large amount of rework (**symptom**). And, most importantly, it led to the need to regain user satisfaction and acceptance of the system—I mean acceptance in the sense of the users agreeing to use it, not buyers' acceptance.

This is a clear case of "garbage in, garbage out." When the business side of the company does not understand the technical underpinnings of the system development or the business environment, the resulting problems in the system cannot easily and simply be resolved by labeling them as technical debt.

Lack of clear business goals inevitably will lead to technical debt when the as-designed-and-implemented system functionality and the expectations of the market do not match. A case like this example from Phoebe requires understanding the business priorities first and then tackling the system redesign.

Requirements Shortfall

Not articulating detailed requirements, not implementing expected functionality, and not understanding architecturally significant requirements such as security, performance, and availability that crosscut the system will all cause technical debt. The quality assurance manager from Tethys, the global giant, had to deal with this firsthand:

> A lot of business requirements from different divisions (**cause**) were implemented in an unstructured way (**potential technical debt**), which caused a lot of trouble with data flow (**symptom**). The system is a lot more complex than it should be (**potential technical debt**).

More often than not, developers respond to ambiguous and poorly understood requirements by either making narrow choices for the limited requirements they do understand or making overly general choices in the hope of anticipating the eventual requirements. Both responses add complexity that makes changing the system more costly.

However, these multiple issues with the requirements are not technical debt. A case like this example from Tethys requires first understanding the requirements and then understanding the problems it causes in the system and the complexity of the system that makes it costly to address these problems. Only then is it possible to start articulating the actual technical debt.

Causes Arising from Change in Context

Technical debt is a time-related concept. A design choice that did not create any technical debt at the time the decision was made can trigger re-architecting when the system context changes. This re-architecting is the result of technical debt that is caused by a change in business or technology or by natural evolution. Phoebe experienced this technological gap (introduced in Chapter 2, "What Is Technical Debt?") as the team started partnering with larger healthcare providers. Team members realized that their choice of a web services stack created a significant number of incompatibilities. They faced a hard decision to reconsider their design after almost three years of development.

Change in Business Context

Unanticipated external events create unanticipated changes in business goals. All the decisions for a system can be appropriate at the time they were made, but in many cases when the business context changes, they simply do not apply anymore. We could list countless examples: Introduction of the iPhone shook up the telecommunication market; advances in cloud computing facilitated infrastructure, platform, and Software as a Service models and shifted computing resource allocation priorities; and the open architecture initiatives by governments made some proprietary internal efforts obsolete.

When faced with such extreme changes, the right approach is not to ask, "How do we tackle our issue log?" The right approach is to ask, "Which of our business drivers will change in this new world? Which of these business drivers will require products to change?"

Technology Change

Technologies change—some at an anticipated rate and some in disruptive ways—triggering business changes. Lock-in to particular software, hardware, or middleware technology eventually will limit design options and accumulate technical debt in the form of unanticipated rework. Similarly, delays in upgrading technology, hence getting out of sync with the most recent version available, can create issues down the line. In fact, version mismatch is cited as one of the top causes of unanticipated security issues that create financial drains for organizations. Such mismatches often result in significant re-architecting efforts that might necessitate that entire releases be devoted to reducing technical debt, since a simple patch update will not do the trick. A wise approach is to plan ahead for technology change, balancing adaptation with anticipation and building those estimations into the system. How much cost you are willing to bear to anticipate future changes often depends on the level of uncertainty of future change. The more uncertain future changes are, the more it might make sense to adapt as you go and make changes as needed.

Phoebe's experience illustrates the cost of a technology change. As the Phoebe product increased its customer base, team members quickly realized that their earlier choice of web services stack created technology lock-in and limited their deployment options. The initial web services stack was an intentional design choice, or known technical debt, that would allow the product to be released quickly. Then new customers created a need for broader deployment options, which increased the amount of debt beyond the initial value of the original design choice (the tipping point in our timeline). This triggered payment to take higher priority: The development team needed to replace the web services stack with a technology choice that addressed broader requirements for Phoebe.

Natural Evolution

Systems age. As part of this natural evolution, systems change as they are maintained and new functionality is introduced. Such changes can eventually cripple a system. The consequence of such natural evolution, the increasing pervasiveness of software, and the sociotechnical complexities of creating software produce technical debt. We captured this as a principle that summarizes the unavoidable nature of technical debt:

Principle 3: All systems have technical debt.

A robust approach for managing technical debt that provides teams with effective tooling and incentivizes them to talk about technical debt and factor it into development plans can help mitigate the unavoidable effects of system aging. This is similar to healthy living, where making good choices about nutrition, exercise, and a supportive social environment will result in improved longevity and health. Understanding technical debt practices is similar. We address this phenomenon in Chapter 13, "Living with Your Technical Debt."

Causes Associated with the Development Process

Developers and managers often categorize flawed ways of executing software engineering practices and processes as technical debt. While team members might have problems following such processes, improving the processes will not fix the technical debt that has accrued in the system. Effective technical debt reduction involves understanding how undisciplined ways of executing processes influence the system, create unintentional system complexity, and result in technical debt. Preventing new debt from occurring requires a focused strategy, an organization process-improvement initiative, or both. There is plenty of available information about process improvement and how to choose and follow a sound software development process.

Ineffective Documentation

Documentation, in particular architectural design and test documentation, is often the elephant in the room. Existence of system documentation does not ensure that the system will be free of technical debt. The documentation must be *effective*: accessible, pertinent, and up to date. Ineffective or insufficient documentation creates a risk that the system will incur technical debt. An initial small group of developers, under pressure to deliver, will not see much value in spending time and effort documenting some of their design choices, constraints, guidelines, APIs, and other details. Subsequent developers may hesitate to change code that they are not confident they understand. Here is a situation from the Tethys project, as described by one of the developers:

> A portion of our product had little to no architecture documentation or descriptions of test procedures, and it was riddled with bugs (**cause**). The little documentation that existed was so outdated that it was not useful at all. Lack of acceptance of these facts, coupled with an overriding desire to maintain a schedule, led to a release that the customer rejected (**symptom**). The subsequent investigation and resolution of this problem required us to generate the design documentation that we should have created and maintained all along. It cost us months of time to re-create. Only after that effort were we able to locate where the problem was in the system (**technical debt**).

This developer articulates very clearly the impact on the system from the lack of clear, up-to-date, and usable documentation. Ineffective documentation often results in challenges in tracking existing issues, understanding their consequences, and preventing them from incurring more debt. Combined with schedule pressure, injection of new issues into the system becomes inevitable. It is likely that several causes are active simultaneously, as Tethys experienced. Creating the needed documentation did not resolve the project's debt, but it did provide a resource for the team to pinpoint the problems in the system.

Documentation becomes more important as a system experiences more success and the organization adds more people to a team. Without effective documentation, the process of bringing in the new team members becomes longer and more error prone. There is a limit to "the code is the documentation," especially when the codebase grows large and when designers need to communicate key architectural decisions. Some of the subtler design decisions are indeed embodied in the code, but they might not be obvious to the readers of the code. This is the case, for example, in a choice *not* to use a certain package or a certain feature of a library. The reasons against a choice do not appear in the code and, if they are not documented, the rationale is lost to members of the development team who weren't involved in the decision.

Because the Tethys team hires new developers frequently, lack of documentation creates an especially high risk that they will contribute to technical debt. A member of the Tethys team explains as follows:

> Documentation is spread over a set of Word/Excel documents and about four Application Lifecycle Management (ALM) database projects (**cause**). Helping someone new understand the product specification is very difficult because the details are hard to find.

A reasonable goal is to document for the needs of the reader, focusing on just enough documentation rather than documentation for the sake of process.

Insufficient Testing Automation

Test automation becomes especially critical when subsequent releases of a system add more and more functionality that begins making the original features erroneous. Adding new code can break code that originally worked as intended. Development teams focus on testing what they develop for the current release and not what they developed for previous releases. As a result, they introduce inconsistencies throughout the system that cause rework in the codebase, build scripts, and test suites. It takes a lot of effort to capture test cases systematically and instrument them in a form that can be run against the system in an orderly fashion with tools.

At the extreme, the *absence* of automated acceptance or regression testing is a major cause of technical debt. In 2004 Michael Feathers even defined "legacy code" as "code without tests" or code that has little or no automation in running these tests.

Absence of regression tests is also a major obstacle in technical debt remediation: Developers are concerned that refactoring may adversely affect the system's behavior and introduce undetected defects. Therefore, they may prefer to live with not-quite-right code that does the job rather than improve the internal structure of the code at the risk of altering its behavior.

Testing that is not aligned with the product goals can lead to over- or undertesting. Testing that is not relevant to the product can lead to unproductive work or developers ignoring the results, as described with the indiscriminate use of static code analyzers discussed in Chapter 5. Knowledge of business goals, requirements, and architectural risk is needed to guide analysis and testing of the system.

Automated testing has a multifaceted influence on technical debt management. An effective testing strategy, in particular for automated unit testing and regression testing, will influence the system design and uncover issues that may turn into technical debt in the long run. The following two scenarios described by an Atlas developer demonstrate this duality vividly:

> **Scenario A: Building the automated testing infrastructure**
>
> We were not able to fully stand up our automated testing at the beginning of the project (**cause**). As development proceeded, the volume of features to be tested grew until the automated testing infrastructure was fully implemented (**symptom**)**.** Even after clearing this backlog, new features and changes resulted in some older tests requiring rework so they would continue to pass (**potential technical debt**).
>
> **Scenario B: Building tests to catch issues and ensuring there is sufficient automated unit testing**
>
> Test coverage of the application wasn't fully assessed, and the sole test resource was removed earlier than anticipated (**cause**). Automated unit tests weren't completely defined. Developers were expected to perform adequate testing of their teammates' code but weren't always able to do this as thoroughly as they needed to because they were under tight time constraints to release the application. Consequently, we found ourselves overwhelmed with unexpected defects that the users kept finding (**symptom**). The root cause of these still lingers in the system (**potential technical debt**).

Scenario A describes an example of a technical debt item that introduced rework and needed to be fixed. The nuance in Scenario A is that it describes a causal chain that begins with the lack of automation that results in rework items in both the tests themselves and in the system due to misalignment between the tests and code (which we explored in Chapter 7). Scenario B also describes a chain of causes and their consequences. It is similar to ineffective documentation. When the tests did not exist, the developers did not catch unanticipated errors introduced into the system in a timely manner, which caused technical debt to accumulate. Similarly to documentation, testing often is present but may be very ineffective.

The goal of managing technical debt is to tease apart the root causes (in this case, lack of automation) from the technical debt development artifacts (reworked features and tests). Distinguishing these is necessary to devise clear remediation strategies for removing the debt.

Misalignment of Processes

All software development teams use processes. Members might deviate from the processes, which can cause technical debt. We refer to this deviation as *misalignment of processes*. Addressing the consequences of technical debt that result from misalignment of processes may require multiple actions at both the product level and the organization level. The Tethys developer who offers the following insight understands the difference between reckless, inadvertent practices that could and should be avoided and the strategic, intentional underpinnings of technical debt:

> Delays in our project are due to creeping requirements, insufficient resource allocation, and interdepartmental disagreements—and all these are due to poor process management and cause technical debt (**cause**). We should fix our process but also do a deep system analysis to understand our current technical debt.

The process used by developers could be well defined, such as Scrum, Rapid Application Development, or Scaled Agile Framework®. Or it could be a homegrown and implicit process with less-defined activities and roles. In a small team, members would not need an elaborate process description but would likely have some tribal knowledge of how they develop a system. Issues start arising when teams do not buy into the selected processes or do not understand them well.

Examples can range from simple oversights such as not checking the approved list of features with the customer (hence developing the wrong feature set) to not following the development procedures for checking in only tested code or pulling tasks from the backlog. Creating a common understanding of and getting buy-in for the procedures to follow among the team members will help them avoid unintentionally accumulating debt.

Causes Arising from People and Team

One of the critical and often overlooked influences on system development is the people developing the system. People make decisions, people implement the systems, and people use the systems. There are countless examples of ineffective teams or developers having widespread impact on a system that is only later recognized as technical debt. The causes can be traced back to inadequacies in transitioning new developers from different backgrounds, recruiting the right people, providing necessary training on new technology or the product environment, and so on.

Inexperienced Teams

Having one or two individuals with little experience or the wrong kind of skills is one type of problem. But with the increasing demand for software professionals, we see cases where organizations need to hire a substantial number of less experienced developers to put together a team, which almost immediately results in the project taking off with technical debt. The Phoebe team unfortunately had to go through this:

> At the beginning of this project, there were more than 20 developers, but nearly all of them were entry level (**cause**). It took us forever to finish even a simple task, and until a couple years ago, it really showed in the code (**symptom**). Functions were inefficient or needlessly complex, an entire class was copied and pasted from the Internet without regard for how well each part fit into the project, several functions were not actually being used or were unneeded, and duplicate code was found throughout the system (**potential technical debt**).

The critical point here is to recognize the issues caused by inexperienced teams and provide the right learning environment for them to succeed. In a situation such as Phoebe's, the organization should develop hiring and training strategies and use focused analysis tools to identify, prioritize, and fix the issues brought up in this anecdote. The development artifacts are the copied class, the functions with unneeded complexity, and the unused and duplicate code. To pay the existing debt, the team needs to fix the artifacts and ensure that the costs do not recur.

The knowledge and experience of stakeholders outside the team who have key decision-making power is also critical. For example, the Atlas team endured a period of frustration with the product owner as team members moved their product to a mobile environment:

> The cause of the technical debt can be traced to the product owner, who does not understand mobile applications or the systems that mobile applications work with (**cause**). The product owner also does not understand basic development processes or agile methods. We spend most of our time correcting or teaching the product owner about mobile technologies or development processes in order to do our job.

It is important to recognize that while lack of experience can inject unintentional technical debt into a system, getting caught up in the blame game will take the product or the team nowhere. Technical debt should be traced to the system artifact where it exists. Then the team can determine how to pay existing debt and rearrange responsibilities to ensure that the people with the right skill sets will have the right roles in the future, or needed skill development can be provided. Teams can build competence when they have the resources and opportunities to develop skills, share knowledge, and gain experience as they apply what they have learned in their projects.

Distributed Teams

Coordination issues can create misaligned assumptions about design decisions, which can cause technical debt. Distributed teams face task coordination challenges more often than not. Planning for handoffs should account for potential coordination issues. The Tethys architect offers an example of his project team's interaction with an offshore team:

> We learned the hard way that handing development to the offshore team before ironing out all the architectural wrinkles can accumulate significant debt, even when you are taking it on intentionally (**cause**). We allowed the offshore team to start developing, warning them that the API calls were incomplete in some areas because we needed to understand performance implications better. We assumed that we could coordinate these changes once we decided what to expose on the API. The offshore team had to make certain assumptions on the API to start, which included the incomplete parts, and our teams did not communicate well. Unfortunately, this resulted in redo in multiple areas (**technical debt**).

The Tethys architect assumed that his team in California could complete the API while the offshore team in Europe started implementation. Because his team arrived at work after the offshore team had gone for the day, it took the California team a few rescheduled meetings to realize that they would have to get to work early to communicate with the offshore team. Although the architect explained that there were missing API calls, the offshore team assumed that if the California team handed over development, the architecture must be good enough. From the perspective of the project lead of the offshore team, he did the right thing. Unfortunately, one of the missing API calls turned out to be key for optimizing performance. Incomplete API calls represented intentional technical debt for the California team. These avoidable misinterpretations between the two distributed teams regarding the status of the API caused the unintended consequence of a performance bottleneck to hit even harder. It took these two teams a while to understand where the issue was located.

Undedicated Teams

In many organizations, developers get pulled in several directions, especially the more experienced ones. This creates not only task switching but also priority shifts. An individual or a team that has competing priorities will focus attention on the most pressing items. In matrixed organizations, project managers should give special attention to setting priorities to avoid losing the attention of teams and individuals. The Atlas, Phoebe, and Tethys projects all suffered from the consequences of undedicated teams, especially as they started to grow. Building effective, focused, dedicated teams is a social and organizational challenge. The downside of not making team building a priority is that when teams are not given enough time, training, autonomy, and resources, they will inject unintentional technical debt as they make design trade-offs to manage priorities.

To Conclude

This chapter describes the causes of technical debt in four major areas: business, change in context, development process, and people and team. Many success stories, as well as failure stories, of technical debt can be traced to one or more causes in these areas. You will find that most of the time, schedule and cost pressure are the contributors that create a domino effect with other causes that pile up as well. Recognizing the causes will help you recognize technical debt, intentional or unintentional.

Knowing the business, understanding the technical underpinnings of the system, avoiding process churn, and building effective teams will help you manage technical debt. Understanding the causes helps you identify the elements of your software development process and organizational realities that create risks for injecting technical debt into your system.

What Can You Do Today?

For major kinds of technical debt, it is important to identify the root cause: schedule pressure, process or lack of process, people availability or turnover, knowledge or lack of knowledge, tool or lack of tool, change of strategy or objectives, and so on.

It is possible to understand and avoid or mitigate the causes of your technical debt immediately, with easy, low-cost actions such as the following:

- If you are a software developer, an architect, or a tester who is an active participating member of a development team, communicate the causes you observe with your team. Meanwhile, describe the rework to reduce technical debt that is the consequence of these causes.
- If you are a team lead, project manager, or Scrum master, start by asking your team what causes technical debt and what the team can do to avoid these causes.

- If you are a software development manager, director, or program manager who oversees multiple projects, create clear communication lines for the business goals and short-term and long-term vision for the product. Give power to your teams and invest in developing their skills. Do not jump to conclusions. Taking a day and a half to conduct a structured root-cause analysis discussion can save you a lot of headaches as your project advances.

When you understand the causes, you can plan specific actions to address them or mitigate their effects.

For Further Reading

In "Technical Debt Quadrant," Martin Fowler (2009) articulated the difference between deliberate (intentional) and inadvertent (unintentional) technical debt.

Pioneer thinkers in software engineering like Manny Lehman (1980, 1996) and David Parnas (1994) made us aware of the consequences of natural evolution and software aging decades ago. Natural evolution and software aging are two of the reasons all systems have technical debt.

Jim Highsmith (2002)—then of the Cutter Consortium and now with ThoughtWorks—has written extensively about agile project management and the tension between adaptation and anticipation. This tension has an impact on all four areas of technical debt causes that we have discussed: the nature of the business, change in context, development process, and people and team.

George Fairbanks (2010) introduced the idea of "just enough architecting," which also addresses just enough architecture and design documentation. He focused on the need for effective and sufficient information at the right time rather than architecting and documenting just for checking a box.

Stories from developers and software project managers help us understand how they categorize causes of technical debt. Studies by Lim and colleagues (2012) and Tom and colleagues (2012) provide anecdotes that map to types of causes similar to those we summarize in this chapter.

Chapter 11

Technical Debt Credit Check

You want to execute a deep analysis of your system and formulate a strategy for managing your technical debt. Beginning with a quick sanity check of the business goals against the system architecture, development practices, and organizational context will provide guidance for successfully executing that deeper analysis and determining actionable outcomes. In this chapter, we introduce a technique for assessing the context and state of your software development project to reveal the causes of your debt.

Identifying Causes: Technical Debt Credit Check

How do you begin to manage a complicated situation? Consider this scenario: You come home and find out your living room is flooded. What do you do first? Do you think about the best plumber to call? Do you mop the floor? Do you call your insurance agent? Or do you quickly look around to assess the situation, see if the water is continuing to come in, shut off the main water valve, move your belongings out of harm's way, and then figure out the cause, the source of the water?

Likewise, a quick check of a project and the software and system under development may reveal technical risks in the business vision, architecture, organization, and development practices that can potentially inject technical debt into the system. You can use these findings to define criteria for measuring technical debt and select techniques and tools that will help you measure against those criteria. You will find this information useful if you plan to conduct an overall technical debt analysis of your system to fully characterize its current state of technical debt.

The goal of the Technical Debt Credit Check is to identify the root causes of a system's existing technical debt and determine whether the debt will continue to grow. Understanding the causes of debt is essential for selecting the appropriate management practices and removing the debt. In Chapter 10, "What Causes Technical Debt?" we looked at common causes of technical debt. The Technical Debt Credit Check helps teams understand what might be causing their debt, especially if it is at a chronic level. This simple questioning technique enables teams to quickly review their business vision, the organization's capacity to support that vision, and the software development artifacts and practices. For the flooded living room, such a check would help you determine whether the pipes leaked, the dishwasher overflowed, or someone left a faucet on. Once you find the source, you can stop the flow of water and look for damage in other areas of the house that might not be as visible as the water in the living room.

In the next few subsections, we describe the purpose of this technique, who participates and when, what types of input you need, the steps to proceed, and the outcome.

Purpose

The Technical Debt Credit Check is a systematic approach to navigate through the context and state of a software development project, using four focus areas that are worthy of attention. By reviewing key criteria, an organization can quickly identify potential causes at risk for creating technical debt and that need further analysis. This initial technique is especially useful when an organization is dealing with the consequences of unintentional technical debt. In Chapters 5, "Technical Debt and the Source Code," 6, "Technical Debt and Architecture," and 7, "Technical Debt and Production," we described how to choose appropriate analysis approaches to further clarify the root causes and trace technical debt to the development artifacts associated with code, architecture, and deployment.

Who Is Involved?

This technique identifies the potential causes of technical debt from the perspective of the development team and project management. Developers have firsthand knowledge of the development artifacts associated with technical debt and its symptoms, and managers grasp the consequences for cost and value. A small group of two or three members of the project team will act as analysts and interview the key stakeholders of the project, focusing on areas of the business vision, architecture, development, and organization.

When Can You Conduct?

You can use this technique in two ways. An organization or a team may sense that technical debt is building up but find it hard to begin addressing it systematically. In such a case, you can use the technique as an intervention to bring awareness of technical debt into the organization. You can also use the technique as an ongoing activity that is part of the project's process for continuous improvement. In this case, you establish a baseline of areas that are most likely to contribute to technical debt and allocate analysis and management resources accordingly. Once the baseline is established, you can use the criteria to investigate causes periodically to keep technical debt under control.

Inputs

The inputs to this technique are the context and state of the software development project, focusing on the business vision, architecture, development practices, and organization. Focusing on these areas will help the team move quickly from overgeneralized causes that may express any project's struggles (summarized in Chapter 10) to project-specific causes that the team can use to identify concrete technical debt items. The inputs may be found in multiple artifacts as well as in the heads of developers and key stakeholders and in the tribal knowledge of the organization.

Steps

Here is a step-by-step description of how to conduct a Technical Debt Credit Check:

1. In collaboration with the project decision makers, select key stakeholders to interview. At a minimum, select junior and senior developers, the architect, the project manager, and key decision makers.
2. Discuss the state of the project, focusing on the business vision, architecture, development, and organization. Ask questions targeted to uncover the common causes of technical debt, as detailed later in this chapter.
3. Consolidate issues across the key focus areas to identify causes that lead to risks related to the business goals, the architecting activities, the development practices, and how people are supported by the organization. Issues may be similar or may overlap.
4. Present the results to all the stakeholders and the key decision makers.
5. Guide the stakeholders in prioritizing the identified causes. Estimate the probability of occurrence and the potential impact of each risk. Rank them according to their potential risk of triggering technical debt—high, medium, or low. Express the resulting list as typical risk areas but focus on technical debt, as in these templates:
    - If <bad thing that may happen>, then <negative consequence> might result.
    - <Factual statement of existing situation> may lead to <negative consequence>.

Output

The output is a *scorecard* that includes a list of causes of technical debt and the impact rating of each as high, medium, or low.

Four Focus Areas for Understanding the State of a Project

We suggest that you initially focus on four key areas to understand the context and state of a project. These four areas will help you filter the myriad causes to a handful of them to develop a simple, actionable strategy for technical debt triage.

Business Vision

Through a clear vision of the business goals for the system, a project team will understand the desired system qualities, the desired software development state, and the consequences of choices that developers make when diverging from that state. Without this clear vision, a system can suffer many unintended consequences that result in technical debt. Key criteria to investigate to ensure that the development effort is aligned with the business vision include the following:

- Are business goals clear, and do they reflect stakeholders' concerns?
- Are success strategies defined and clearly communicated (for example, roadmaps, product portfolios, key timelines)?
- Is funding secured, and are there related resource priorities that could affect the project?
- Does the product owner understand the dynamics of the business environment and changing market opportunities?
- Are consequences of key business decisions for product design and development clear?
- Has the development team established effective communication channels with the customer? And timely feedback cycles?

The focus on business vision will help you identify business-related causes (as discussed in Chapter 10), such as the magnitude of time and cost pressures, alignment of business goals, and clarity of requirements.

Architecture

Architecting activities that balance the short-term and long-term technical goals of a project must be integrated into the software development lifecycle to strategically manage technical debt. The view that these activities happen sequentially often creates silos, architecture conformance issues, and unexpected rework costs in later stages of the development effort. Teams must make architectural decisions with consideration for business goals, organizational needs, and the desired state of development. Key criteria to investigate include the following:

- Are architecturally significant requirements defined, tied to business goals, and communicated clearly across the business and technical stakeholders?
- Is evidence provided that the architecture satisfies key requirements?
- Are there known architectural issues, and are they tracked and managed?
- Is the timeline of key architectural decisions clear, considering both short-term and long-term business goals that the architecture needs to support?
- Is the impact of the changes in technology and their limitations clear?
- Are key build and integration, test, and deployment scenarios clear, well developed, and utilized in a timely manner?

Identifying causes of debt related to the architecture will uncover causes related to the context of the project, such as technology change, business shift, or market evolution. It will also give hints about where the most critical technical debt may reside in the system. In addition, you may uncover process-related causes—not only processes related to architecture but also processes related to documentation and software development. Process causes are important because they indicate how well your processes will guide the team to manage and service technical debt.

Development

The bottom line of any software engineering project is the quality of the running system. It is critical to align the development practices with the business goals and architecture to avoid unintentional debt. Investigating the following criteria will help uncover potential risks related to development and its processes and tools:

- Is the development infrastructure in place and aligned with the architecture?
- Are necessary quality control methods available and used (for example, code reviews, inspections, testing, continuous integration, deployment practices)?

- Does the development team have appropriate tools and use them effectively? Is training provided when needed?
- Is an environment in place to measure and monitor the implemented system quality and “done” criteria?
- Has the development team considered code maintenance and evolution?
- Does the team understand, embrace, and follow the established software development processes and practices?

Organization

Any successful organization operates within a culture and established procedures. When an organization’s underlying culture and processes do not support its people and embrace change, technical debt creeps in. Key criteria to investigate include the following:

- Does the organizational structure enable collaboration? Do the development team, project management, and architects effectively support each other?
- Are necessary procedures and technology in place to respond to change?
- Has the organization determined the impact of cost of delay and rework and decided how to manage trade-offs?
- Has the organization acknowledged the impact of uncertainty on the project?
- Has the organization provided training for skills needed to succeed in the project?
- Has the organization provided the team with sufficient resources?
- Is there a procedure to bring new team members up to speed with the project?
- Have teams and team members established clear communication channels?

Diagnosing the Causes of Technical Debt in Phoebe

The example we gave in Chapter 10 from the Phoebe project showed that technology change was the cause of an architectural issue:

> The open-source web services stack that we rely on went through several versions, but we did not upgrade. Our customers require new features that we cannot support if we do not upgrade soon.

BUSINESS VISION (▼)

- ✓ business goals
- ✓ success strategies
- ▼ resources
- ▼ customer communication
- ▼ consequences of business decisions
- ▼ feedback cycles
- …

ARCHITECTURE (-)

- ⊖ architecturally significant requirements
- ⊖ architecture fitness
- ▼ architecture issues
- ⊖ short-term and long-term architecture goals
- ▼ impact of technology change
- ⊖ build, integration, test, and deployment alignment
- …

DEVELOPMENT (-)

- ▼ development infrastructure
- ▼ quality assurance
- ⊖ development tools
- ⊖ done criteria
- ⊖ code maintenance and evolution
- ▼ software development processes and practices
- …

ORGANIZATION (▼)

- ▼ collaboration
- ✓ change management
- ▼ cost of delay and rework
- ⊖ uncertainty
- ▼ development team resources
- ✓ new employee onboarding
- ✓ team communication
- …

Legend

- ✓ No issues causing technical debt (green)
- ▼ Can improve, can contribute to technical debt (yellow)
- ⊖ Significant issues contributing to technical debt (red)

Figure 11.1 *Scorecard for causes of technical debt in the Phoebe project*

Figure 11.1 shows the scorecard for causes of technical debt in the Phoebe project. We recommend rating the outcomes as red, yellow, and green, where red indicates that the issue area is causing technical debt, yellow indicates that the area is a potential cause of technical debt if not managed better, and green indicates that the area is being managed adequately. A red rating implies that the answers to the questions under that focus area were mostly negative or insufficient. Phoebe clearly needs to better manage its short-term and long-term architectural issues.

Diagnosing the Causes of Technical Debt in Tethys

Let us look in detail at Tethys, an organization that needs to focus on aligning its business goals and organizational processes. As Tethys grew into the global giant it is today, management decided to separate the responsibilities of development and quality assurance. The development teams operated in an iterative and incremental delivery tempo, and the quality assurance team followed a waterfall approach to the software development lifecycle. In safety-critical and avionics environments, where products must conform to industry and safety standards, such over-the-wall handoff between development and quality assurance is not uncommon.

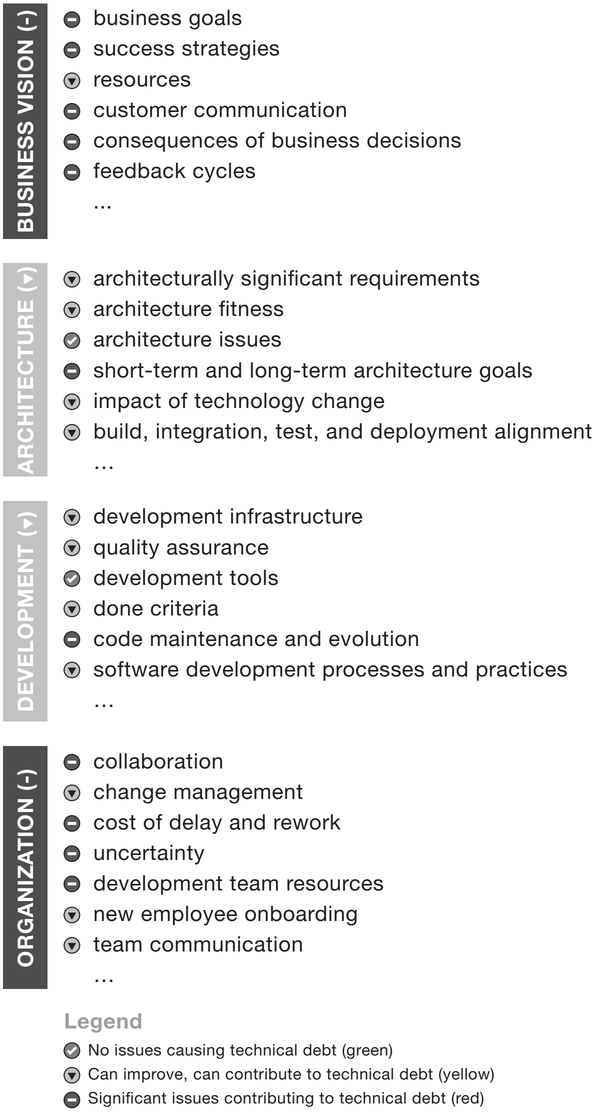
However, the delivery schedule of the development teams did not align with the quality assurance teams' expectations and schedule. The development teams delivered features in increments. The quality assurance team did not test feature increments even though other features relied on them, waiting instead to test each entire feature once it was complete. The inevitable consequence of this practice was that quality assurance found defects, and developers' time became consumed with fixing them. While the project manager prioritized new features above all other tasks, new feature development started slipping, and time pressure to deliver became the development team's top priority; the development team had to navigate conflicting priorities of fixing defects and developing new features. In these ways, misalignment of business goals and organizational processes created substantial roadblocks for Team Tethys.

In addition, another organizational issue had causes rooted in inexperienced teams, an area that contributes many causes of potential technical debt across the software industry. One Tethys developer reflected on this issue:

> We have a very high turnover rate, but we do not allocate the time to bring the new hires up to speed with the system and our development practices. New hires inject a lot of defects because we do not onboard them properly. When more experienced staff members mentor new hires, they succeed, and we do not see the chaos, but often no one takes enough time to do this, and then issues go unnoticed. This lack of training will eventually slow down our velocity, which is already showing signs of the problem.

The team lead of Tethys negotiated with his manager and customer to postpone new features as the inexperience of junior members was becoming more problematic with each iteration. While the team fixed some of the immediate issues caused by these defects, he conducted a Technical Debt Credit Check and reported the results in the scorecard shown in Figure 11.2.

A closer look at the Tethys business vision revealed a misalignment of business goals. The project team did not understand the product-line opportunity. While the



BUSINESS VISION (-)

- ⊖ business goals
- ⊖ success strategies
- ▼ resources
- ⊖ customer communication
- ⊖ consequences of business decisions
- ⊖ feedback cycles
- ...

ARCHITECTURE (▾)

- ▼ architecturally significant requirements
- ▼ architecture fitness
- ✓ architecture issues
- ⊖ short-term and long-term architecture goals
- ▼ impact of technology change
- ▼ build, integration, test, and deployment alignment
- ...

DEVELOPMENT (▾)

- ▼ development infrastructure
- ▼ quality assurance
- ✓ development tools
- ▼ done criteria
- ⊖ code maintenance and evolution
- ▼ software development processes and practices
- ...

ORGANIZATION (-)

- ⊖ collaboration
- ▼ change management
- ⊖ cost of delay and rework
- ⊖ uncertainty
- ⊖ development team resources
- ▼ new employee onboarding
- ▼ team communication
- ...

Legend

- ✓ No issues causing technical debt (green)
- ▼ Can improve, can contribute to technical debt (yellow)
- ⊖ Significant issues contributing to technical debt (red)

Figure 11.2 *Scorecard for causes of technical debt in the Tethys project*

long-term goal was to serve multiple markets with the same product, the short-term goal was to serve a pressing time-to-market requirement. The development teams went out of their way to create a general architecture, and they missed the immediate product-specific delivery needs, further adding time and cost pressure.

Tethys began its journey to the market with significant risks that the project team would inject technical debt into the product because of its confusion about short-term goals and the business vision. The goal of creating a solution that envisioned all potential variations of the product resulted in an over-parameterized architecture. In an effort to create an infrastructure robust enough to handle the natural evolution of the products and product line, the team added unnecessary complexity for the immediate customer need. Both the overgeneralized architecture and unnecessary complexity created several technical debt items as development progressed. The team got lost within the variation parameters, so many of the features they implemented were incomplete. Alternatively, focusing exclusively on the short-term goals of the immediate customer would have introduced a different set of issues. Getting clarity on the trade-offs would have helped team members recognize where they would need to take on technical debt with intention so that they could manage it strategically. The Tethys project would have taken on technical debt either way, but it missed an opportunity to take on the debt strategically and not only acquired the wrong type of debt but also did not recognize it until it almost got the project canceled.

The organizational structure of Tethys provides clues about how a number of causes related to process contributed to the accumulating technical debt. Tethys was not able to align the multiple processes of the iterative and waterfall models across the distributed development and quality assurance teams, which operated at different tempos due to the challenges of fulfilling compliance requirements in the safety-critical domain.

The Tethys project used three different cycles: annual releases to the customer, quarterly testing performed by a quality assurance team, and monthly sprints conducted by the development team. The goals of each cycle were different, yet they had important dependencies. For example, by the time the quarterly testing found issues with product features in a given release, the development team had already implemented three other releases on top of the system, increasing its size and complexity. This made it harder for the team to locate issues, so they spent an extensive amount of time in bug-fixing mode. Consequently, by the time the developers realized that they could not make progress while using three unaligned cycles, it was too late to redesign the overly complex architecture that resulted in unmaintainable and buggy code.

After conducting the Technical Debt Credit Check, team members realized that their misaligned business goals created the overly complex architecture. They conducted an analysis of their codebase, using some static analysis tools, focusing on security (as discussed in Chapter 5). To understand the impact of that debt, they conducted an architecture review (as discussed in Chapter 6). Consequently, the

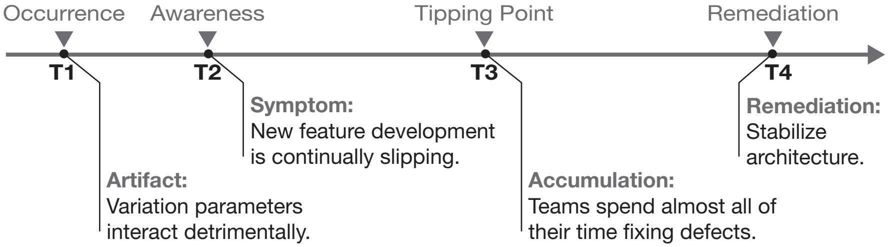

Occurrence
Awareness
Tipping Point
Remediation
T1
T2
T3
T4
Symptom:
New feature development is continually slipping.
Artifact:
Variation parameters interact detrimentally.
Accumulation:
Teams spend almost all of their time fixing defects.
Remediation:
Stabilize architecture.

Figure 11.3 *Tethys and the technical debt timeline*

team decided to take the following corrective actions immediately to reduce the debt and avoid its further accumulation: Reduce the number of variant parameterizations in the architecture, add guidelines for architecture conformance, and have everyone work from the same architecture. There were other consequences of the Technical Debt Credit Check, such as replanning the release cycles to better align the development and testing cycles and revisiting the testing strategy completely (as discussed in Chapter 7).

Mapping the events onto the technical debt timeline as shown in Figure 11.3 allowed the Tethys team to assess the consequences of its debt. Team members continued to mitigate risk by remediating the debt, as discussed in Chapter 9, "Servicing the Technical Debt." They decided to stop delivering new features for at least a quarter until they repaid some of the debt. Distinguishing the causes of their debt and the current debt that they needed to fix allowed them to recognize that if they kept fixing defects, they would never get ahead of the problem. They needed to correct the course of their product-line architecture.

What Can You Do Today?

Teams that do not follow established software engineering practices will take on reckless and unintentional technical debt. In this chapter, we introduced a technique to help you identify where you may be diverging from established practices and introducing technical debt.

With the right stakeholders in the room and good facilitation skills, you can conduct your own Technical Debt Credit Check and create a scorecard indicating the causes that contribute most to your technical debt accumulation. Then you can begin developing a plan to manage your debt strategically.

For Further Reading

Technical risk assessment is a routine practice in many organizations. The Technical Debt Credit Check we describe is inspired by these approaches, but it is meant to provide a lightweight approach to assessing technical debt risks. The Architecture Tradeoff Analysis Method by the Software Engineering Institute (Bass et al. 2012), for example, similarly walks through the architecture of a system to uncover technical risks against business goals and architecturally significant requirements.

The guidelines that the Agile Alliance Technical Debt Initiative has developed for executives, managers, and developers summarize code quality rules that, when violated, generate technical debt. In particular, they propose an *Agile Alliance Debt Analysis Model* (A2DAM) (Fayolle et al. 2018).

Chapter 12

Avoiding Unintentional Debt

In this chapter, we summarize software engineering practices that any team should incorporate into its software development activities to minimize unintentional technical debt. These practices are essential for organizations and teams to institutionalize an integrated approach to managing technical debt.

Software Engineering in a Nutshell

Managing technical debt requires a broad understanding of software engineering practices—and that is exactly the goal of this chapter: providing starting points for practices that are essential for establishing a well-rounded approach to technical debt management so you can spend your time on strategic technical debt rather than fighting avoidable fires. Because these practices are described in many software development books, we only summarize them here and explain how they support technical debt management or how they relate to technical debt.

Not using sound and proven practices to run a software engineering project is likely to bring you a lot of technical debt. We discussed aspects of this phenomenon in detail when we teased apart the causes of technical debt in Chapter 10, "What Causes Technical Debt?" More importantly, using recommended software engineering practices will help you avoid violating the key principles of technical debt that we've introduced in this book.

If you do not institutionalize good coding standards and code quality checking practices, in time your code will inevitably degrade. You will start getting lost in accumulated defects. Your architecture will also eventually start degrading.

If you do not know your architectural decisions and trade-offs and review them continuously, you will not react to architectural changes in a timely way. You will not be able to determine what to fix, where to fix it, or what caused the issue in the first place. Keep in mind these two principles:

Principle 5: Technical debt is not synonymous with bad quality.
Principle 6: Architecture technical debt has the highest cost of ownership.

If you do not know your short-term and long-term organizational and project goals and do not institute practices to establish a roadmap toward them, you will get caught in the "blame game." As we pointed out in Chapter 3, "Moons of Saturn—The Crucial Role of Context," only the most trivial systems escape technical debt, and it is better to manage it deliberately than to have it manage you accidentally. Keep in mind this additional principle:

Principle 3: All systems have technical debt.

Good coding, architecture, and production practices are essential components of good software engineering and lead to greater responsiveness to business needs and quality code that is easier to evolve and maintain. Making your software "observable" in some way—through techniques such as static code analysis, monitoring, and logging—will allow you to collect data and use it to interpret system behavior and how it correlates with the evolution and maintenance challenges you experience. We take a deeper look at these practices next.

Code Quality and Unintentional Technical Debt

The following four fundamental practices are critical for creating high-quality and maintainable code:

- Establishing and following sound coding standards
- Establishing and following secure coding standards
- Writing maintainable code
- Refactoring

If you abandon fundamental principles of software craftsmanship, your code will drown in recurring interest payments throughout the life of the project.

Sound Coding Standards

Coding standards are guidelines for specific programming languages that recommend programming style, practices, and methods for each aspect of a program written in that language. The most common form of scattered and unintentional technical debt results from not following such coding standards.

Most software development organizations adopt some form of coding standard that specifies acceptable and objectionable code idioms. These standards are development language specific. Their main objectives are as follows:

- Increasing programmers' and maintainers' understanding of the code
- Avoiding common coding mistakes
- Preventing the use of dangerous, error-prone, or costly forms of implementation constructs

These guidelines include naming conventions, formatting of code, and permissible language constructs. Other areas of concern include file organization and documentation in the form of comments to improve understandability of the overall codebase. Examples of commenting guidelines include the minimum amount of documentation for every public class and public method and what does not need comments within the code. An effective style guideline often describes phrases that avoid confusion and key phrases that increase ease of navigation. "Basics" go a long way, especially in projects where large teams need to be orchestrated, such as when establishing and following naming conventions for public, private, and protected attributes, classes, and method calls.

Integrated development environments help enforce standards and style guides. All the developers on your team should be intimately familiar with and follow the standards and style guides to be used for the project. These can be company specific, or you can adopt an industry practice, such as *Google Java Standard Guide* or *Oracle Code Conventions for the Java Programming Language*.

Secure Coding Standards

Secure coding is the practice of developing software in a way that guards against the accidental introduction of logic flaws and implementation mistakes that result in commonly exploited software vulnerabilities. A combination of security issues, especially when caught late, will accumulate and become technical debt.

The timeline of the Phoebe team mirrors the journey of many teams we interact with. As their product matured, team members started to realize that they would have to do more to demonstrate the security aspects of the product, especially given the needs of their government customers. In anticipation of coming requirements, they decided to be proactive and run a security analysis tool through the codebase. They added several technical stories and tasks to their backlog:

> *Task*: Execute security scan on Phoebe code and document findings.
>
> *Technical story*: As a Phoebe developer, I want to resolve all the security scan findings with Critical or High priorities.
>
> *Technical story*: As a Phoebe contributor, I want to address all Medium and Low security scan issues so that the code quality is improved.

Team Phoebe elected to use a security scanning tool called Fortify, which offers features in static and dynamic application security testing through automating the checking of conformance to secure coding standards based on commonly found security issues. One reason for selecting Fortify was the fact that Phoebe was implemented with Java, with extensive use of J2EE libraries for which Fortify offered up-to-date conformance checks at the time.

As a result of the security scan, team members added 69 more issues to the backlog. In isolation, none of these issues were technical debt. Indeed, most of the issues were minor. However, when analyzed together, it became clear that the Phoebe project had technical debt related to security in the code. A significant number of the issues that were returned by this scan included poor error handling where null pointer exceptions were not caught properly or exceptions were not thrown or caught properly. These were symptomatic of an underlying design limitation in the treatment of exception handling. The list of violations included other common examples, such as the following:

- J2EE bad practice:
  - Leftover debug code
- Poor error handling:
  - Overly broad throws
- Poor logging practice:
  - Use of a system output stream
- Poor style:
  - Value never read
  - Non-final public static field
  - Confusing naming
  - Redundant null check

All these vulnerabilities in time will create security risks that can crash the system, be exploited, or both. After team members addressed these issues, they also educated the rest of the team to follow secure coding practices.

A number of resources can guide you in improving your secure coding practices. For example, the Open Web Application Security Project maintains a document that summarizes secure coding rules and practices. Some of these resources provide general guidance, such as "protect server side code being downloaded by a user," without specifying the kind of protection mechanism to use. Others enforce very specific rules, such as those included in the SEI CERT Secure Coding Standards. There are also tools that implement these and other rules. The MITRE Corporation maintains a universal Common Weakness Enumeration (CWE) database as well as a Common Vulnerabilities and Exposures (CVE) database. These are only some of the ample resources available to educate your teams in secure coding and help them implement best practices. Teams should review secure coding practices at the beginning of a project, when they are establishing coding standards.

But secure coding gets a bit tricky from the perspective of technical debt. Security often gets top priority, and when such issues are found, they are the first to be fixed. Sometimes these are random patches that introduce technical debt. Treating each issue in isolation will often not address the technical debt. Often combinations of security issues that relate to architectural design consequences both create technical debt items and constrain the approaches to fix them. Not following known secure coding practices and standards increases the odds of introducing technical debt to the system, and it will make finding the roots of problems harder as the system grows.

Maintainable Code

Maintainable code and architecting for maintainability are closely related to each other. Following well-established best practices will enhance code maintainability, such as establishing common criteria for class sizes, guiding the use of external libraries, and selecting architectural patterns that promote maintainability.

The ISO/IEC 25000 standard (which evolved from ISO 9126) on software product quality describes system quality characteristics. Maintainability incorporates such concepts as changeability, modularity, understandability, testability, and reusability. Many source code properties affect maintainability. Characteristics that are relevant to maintainable code include unit size, unit complexity, unit interface, duplication, coverage, coupling, cycles, propagation, and types of dependencies. Units can be groupings defined by the artifacts in the development environment (such as lines of code and the number of files, directories, packages, or projects) or semantic constructs of the software asset (such as functions, blocks, classes, statements, and accessors). Organizations such as the Object Management Group and Consortium for IT Quality have recommended standards specifically related to maintainability.

Writing maintainable code is part of developing high-quality code. In the same way, understanding maintainability is part of architecting the system. Establishing clear baselines for these practices will help you avoid the kind of technical debt that is most commonly seen and most costly, yet least likely to be fixed.

Refactoring

Refactoring is a behavior-preserving transformation that improves the overall code quality. Refactoring is not simply cleaning up code; it is a technique involving applying known patterns of improvement. While each refactoring does a little, a series of transformations can introduce both needed restructuring and improvements in code quality and complexity. Basic refactoring guidance is widely available. There are resources that describe how to make small, local transformations and some tools that implement them. There are also catalogs of generic refactoring patterns as well as catalogs of patterns specific to programming languages.

Before refactoring code, you need a solid set of automated unit tests. The unit tests should pass both before and after the code has been refactored. Unit tests safeguard against introducing new issues unintentionally. Savvy developers ensure that unit tests are used and passed during refactoring activities.

The Atlas team relied on refactoring to manage its technical debt within short iteration cycles. Here are two of the team's technical debt items:

> *Atlas #102:* Placeholder: I changed the code and made the tests pass, but the tests are not testing the code. I will fix this tomorrow.
>
> *Atlas #623:* We should create a toolbar superclass /ui/toolbar/bottom_toolbar.mm. And reading_list_toolbar.mm, clear_browsing_bar, and bookmark_context_bar should be based on the superclass. This way, we can reduce the redundant code and technical debt and make sure the style, font, and spacing of the toolbars are always consistent.

In the first example, the developer knew that she made the code work after refactoring but that she also introduced another problem. She created a technical debt item to alert everyone on the team, assigned it to herself, and went back and fixed it the next day. In the second example, the developer had a solution, opened a technical debt item, and described its benefits along with the recommended refactoring.

Refactoring is an approach commonly used by teams to bundle known technical debt issues with other changes and reduce them as code is improved. While refactoring does not resolve deeply rooted architectural issues, it can be an effective technique for improving maintainability and code quality and eliminating some common problems before they become costly.

Architecture, Production, and Unintentional Technical Debt

We have established that the most expensive technical debt is at the architecture level. Today, a good architecture practice can be summarized as a deliberate and continuous focus on architecture issues, not a massive up-front design. Architecture design is not a point in time but an activity that is integrated with a project and that may continue while the system is in operation. Your choices of technology, frameworks, integration, and deployment pipeline will all encapsulate architectural decisions and enable or hinder quality attribute requirements. Here we call out some practices that are essential to understanding the design trade-offs that are part of architecting:

- Eliciting quality attribute requirements that drive the software design and quality
- Incorporating iterative incremental design into release planning
- Aligning the architecture and production infrastructure
- Documenting to address stakeholder needs
- Incorporating lightweight analysis and conformance checking throughout

Quality Attribute Requirements

Producing high-quality systems and managing their technical debt closely depend on understanding their architecturally significant requirements. Quality attribute requirements are the architecturally significant requirements for the system that affect its run-time behavior, system design, and long-term evolvability. There is no shortage of taxonomies and definitions to guide you in requirements specification (for example, IEEE 830-1998: *Recommended Practice for Software Requirements Specifications*).

Establishing a common understanding of quality attribute requirements allows teams to design for them and, more importantly, understand the short-term and long-term architectural weakest links. Designing for these requirements that drive the system structure and behavior is often an ad hoc practice. Organizations often do not allow time to explicitly focus on quality attribute requirements, and this can result in significant amounts of technical debt as the project progresses. Designing with security, scalability, and maintainability in mind is not a trivial task.

Several established techniques can augment existing team requirement management practices that focus on quality attribute requirements. Those that are elicited from key stakeholders and represented as scenarios provide a quantifiable definition and specific prioritization of the architecturally significant requirements. Agile software development processes can incorporate quality attribute requirements as user stories when a system's run-time qualities are visible to the user or as technical stories when a team is focused on internal structural issues.

Iterative, Incremental Design in Release Planning

Reasoning about architecture alternatives and using the architecture to guide implementation choices during release planning provide opportunities for handling technical debt strategically. Explicitly defining tasks related to realizing quality attribute requirements in development iterations and release planning is key. Failing to allocate time blocks for architecting is a recipe for unintentional technical debt.

Modern software development approaches acknowledge the critical and strategic importance of architecting. For example, the Scaled Agile Framework (SAFe®) defines the *architectural runway* as the production infrastructure, architecture, and code that are essential for near-term features and functionality. It recommends allocating time in sprints to create and extend the runway as needed to support the development of the features that depend on it.

Systems with a smaller scope and smaller teams, such as Atlas, may need a shorter architectural runway. Especially in the face of uncertain requirements for technology or features, it may be more efficient for the team to try something out, get feedback, and refactor as needed than to invest more time in trying to discern requirements that are in flux.

Systems with a larger scope and larger teams, such as Tethys, need a longer runway. Building infrastructure and re-architecting the software take longer than a single iteration or release cycle. Delivering planned functionality is more predictable when the structure for the new features is already in place. This requires looking ahead in the planning process and investing in architecture work in the present iteration that will support future features that the customer needs.

An explicit focus on allocating architecture tasks driven by quality attribute requirements will support the development team in making design trade-offs wisely and taking on technical debt strategically. Understanding the state of a development effort focuses teams on architectural design. It is desirable for development teams to reach a software development tempo in which each release delivers value as new functionality or improvement to the stakeholders. Initially this state does not exist. Teams need to build platforms and frameworks, establish architectural patterns, and make decisions about structure and its implementation.

A key enabler to achieving iterative, incremental design requires the following:

- **Understanding the short-term and long-term goals of the business and, therefore, the key quality attribute requirements:** Quantitative response measures and priorities for quality attribute requirements will help teams establish design strategies for these requirements.
- **Eliciting quality attributes as early as possible in the project:** They should be prioritized based on technical difficulty and value to the business and revisited at least at each release point.
- **Understanding the dependencies between technical constraints, products used, and these requirements:** This is an ongoing activity because dependencies are often not immediately apparent as the devil is in the details. Lightweight analysis approaches incorporated into sprint retrospectives will help uncover these dependencies.

Aligning the Architecture and Production Infrastructure

Another essential aspect of the architectural runway is the production infrastructure and the tooling needed to achieve continuous integration, continuous deployment, and monitoring. Recognizing how the software aligns with the release process and production infrastructure will make continuous delivery and its tooling easier to achieve. At a minimum, employing parameterization, self-monitoring, and self-initiating version updates will enable teams to avoid technical debt in production environments:

- Parameterization focuses on environmental variables relevant to the production infrastructure, such as databases and server names. It allows a team to defer binding time and change aspects of the build and production environment without having to change the build.
- Self-monitoring allows for monitoring the system performance and faults as it runs and when it gets out of sync. Both the production infrastructure and the architecture of the system can take advantage of load balancing, logging, and redundancy tactics to realign the allocation and improve system behavior.
- Self-initiated version updating allows a team to run scripts that update the relevant versions of the software in production. Versioning becomes an issue particularly at scale and when continuous integration and deployment is a goal. The clients and the main applications may get out of sync, as may the supporting tooling environment.

Documentation

For many systems, some documentation exists, but it has rapidly become disconnected from the running software. Under schedule pressure, it is all too common for a team to jettison updates to the documentation and use that time to fix one last defect. Consequently, documentation suffers from several problems:

- It rarely helps authors immediately ("I know this, and I can remember it for several weeks or months"), so they have no immediate incentive to spend the time and effort writing documentation.
- What is obvious to one developer may be counterintuitive to another.
- Diagrams take time to create and are tedious to update, even though they provide high value for the reader.
- Documentation is not trusted because it is assumed to be out of date. For some organizations, this is a cultural issue.

Make sure you document what is actually useful. Developers can read the code, so do not create massive amounts of documentation that just paraphrases what is in the code. However, new developers may have a hard time understanding a large body of code, and they can benefit from "roadmaps" to help them navigate the code and get the big picture of how it works. They also need explanations of key design decisions, so they can integrate this original reasoning into their own designs. This is the role of a *software or system architecture document*, along with some accompanying design guidelines. The architecture document should include documentation about key interfaces in the system: the APIs. Another key document should describe the development process, from end to end, including production.

Project management discipline is the key to writing and maintaining documentation. Here are a few heuristics for deciding what documents a development team should produce and how to maintain them:

- **No write-only documents:** If no one will ever use it, do not waste time developing it and maintaining it.
- **Single point of maintenance:** Do not force developers to change information in multiple places. Part of the documents can be generated by tools; for example, diagrams representing the structure can be "decompiled" from the code.
- **Version control:** Documentation should be under configuration management, just like the rest of the system.
- **Mandatory updates:** Release to production should be blocked if vital documentation steps are not completed.

Lightweight Analysis and Conformance

Analyzing the codebase for conformance to the architecture and design should be part of the routine iteration and sprint reviews. A focus on quality attribute requirements will provide the goals to be met and a strategic perspective on design trade-offs. If appropriate, listing the trade-offs and risks identified by analysis as technical debt items will provide additional elements to monitor and manage on the product backlog.

At a minimum, the team should establish the following:

- Module interfaces and responsibilities
- Conformance guidelines, from module to code
- Key design decisions, architecture decisions, and technical constraints

Lightweight analysis allows a team to assess the trade-offs that may turn into technical debt. Every architecture approach used to improve one quality attribute can negatively impact others:

- Putting everything that needs to change in one place may introduce unnecessary dependencies to other components. This is bad for security and other types of changes.
- Data structures with generic interfaces may impose a performance penalty.
- Versioned interfaces increase complexity, which is more difficult to test and increases the chance of system crashes.

The team needs to be aware of these issues, mitigate them, and document them. Lightweight review of the system with regard to quality attribute requirements uncovers such issues early and gives the team opportunities to address them before they become technical debt or to explicitly take them on as intentional technical debt items.

At a minimum, the team should understand the principles of lightweight architecture and design analysis:

- Important quality attribute properties of the architecture need to be evaluated. The important qualities are derived from the business goals.
- Quality attribute scenarios translate business goals into required quality attribute properties.

- Quality attribute scenarios help identify relevant components of the architecture to analyze.
- Architectural approaches with their quality attribute properties should be clear to the team, as should the side effects and trade-offs of those architectural approaches.
- Mismatches between architecture properties and scenarios become risks to the business goals and potential technical debt items.

Leveraging Agile Practices to Manage Technical Debt at Scale *by Robert Eisenberg*

I spent over 30 years in software development for large, high-reliability and long-lived systems in the defense industry. Programs such as these have particular challenges related to technical debt. The challenges begin with the long-lived nature itself, which can span decades and is thus subject to "software decay."

Software decay occurs when a fairly clean architecture, design, or implementation slowly degrades over time as successive changes are made, each one implemented with a focus on getting it done in the cheapest, quickest, least-likely-to-have-an-impact way possible to meet the customer's immediate needs. Each change creates a little more technical debt, and the debt compounds over time, slowly corrupting the original architecture, design, and implementation. Once the debt becomes burdensome, customers (and internal management) often resist remediation since they expect to be paying for new features, not debt remediation, which they often mistakenly blame on prior shoddy work.

You might think that the inherent nature of these high-reliability systems would lead to less technical debt, but that is not necessarily the case. Formal requirements and reliability objectives focus on externally visible characteristics, not the intrinsic product quality and maintainability of the underlying software (unless by chance there are requirements for such, which is rare). These systems can also be subject to extreme schedule pressure against a fixed set of requirements and cost baseline. Thus, technical debt can be as significant an issue on these types of systems as any other.

So, suppose you're on a large, long-lived program, and you think you have a technical debt problem. Now what? What are some strategies for getting started? Here is some of what I've learned. First, let's start with what not to do. Don't try to perform a comprehensive analysis of your multimillion-line software product in order to determine total technical debt and develop a comprehensive remediation plan. Getting an overall perspective can be insightful, but you can also quickly get wrapped around the axle in terms of actionable outcomes. The data can be mind-numbing, and the scope of the problem can seem insurmountable, especially for program managers concerned with the cost and schedule for new features. Showing that you've built up millions of dollars in technical debt (principal) isn't useful or actionable; estimating future interest is guesswork at best. Trust me, I've tried this approach without much success.

I believe in agile values and their applicability to the technical debt challenge. You should approach debt identification and remediation in an incremental and iterative manner, growing your practices and methods based on experience in execution. Programs applying agile frameworks and practices have many opportunities for "baking in" technical debt practices and growing them over time, including the following:

- **Definition of done criteria:** Initially include criteria for identification of any existing debt uncovered during feature and story development. This will help identify the "visible debt"—that is, the debt that became a visible hindrance during normal development. These debt items should be documented, ideally in the program backlog tracking tool, as "debt stories." A "no new debt" criterion can also be included to prevent debt from growing. And if new debt is unavoidable for some reason, then it, too, should be documented with a debt story. What is considered technical debt is often initially subjective based on collective team experience with the program and good software craftsmanship, but it may later be augmented with measurables (for example, from static analysis, other tools, or formal analysis methods). I encourage teams to apply the "scout rule" for camping to software: Always leave the code a little cleaner than you found it. I recommend including debt actions (stories) in the same tracking tool as all other work and as part of a single program or team backlog. Our agile-inspired phrasing is "all work is work; all work goes on the backlog."
- **Definition of ready:** Before starting a new feature or story, check the backlog to identify any known debt items that should be considered during implementation because they impact the same area of the code or would otherwise impede its development. In addition, make debt consideration part of the standard design and product size estimation processes during planning (for example, feature and story pointing). These processes will help make debt prevention and remediation more proactive since they occur before development has begun.

(continued)

- **During planning:** Planning occurs at many levels. I'll start at the lowest level and work up. During team-level planning (for example, sprint planning), look to include previously identified debt remediation stories associated with the code being updated. When discussing story size, be sure to consider the effort to prevent any new debt. I've seen some teams allocate a small percentage of capacity each sprint or program increment (using SAFe® terminology) to debt remediation. During program increment (or equivalent) planning is the time to consider larger debt prevention and remediation items, such as those associated with more substantial architecture or design changes. Here again, consider necessary debt remediation and prevention when sizing the bigger chunks of work (such as features). At the highest level of program planning, one of the key factors I've seen is to evaluate the existing debt within any planned reuse code (for example, code being reused from a prior program or Independent Research And Development (IRAD) project). Too often programs fail to consider and include the costs necessary to refactor the reuse code to maintain internal product integrity during the necessary modification and enhancement.
- **During retrospectives:** Ask team members if they uncovered any debt during the development. This is often the time when debt is identified related to the development infrastructure or other facets that aren't directly associated with the primary codebase. Once again, create debt remediation stories as required. Teams can use retrospectives to look for trends in prior debt stories (for example, common root causes). They can track and monitor the overall volume of debt stories, considering questions such as "Is our debt getting too high?" and "Is our debt backlog growing, maybe because we never prioritize those stories?" and "Is increasing debt measurably affecting our velocity?" Retrospectives, especially at longer intervals (such as program increments), can also be an opportunity to look at things like the occurrences of bad fixes (that, perhaps, introduced a new problem or undesirable behavior) or new capabilities that had high debt rates after the team thought the system was done. Both can be indicators of high debt in the underlying code.

Collectively these steps help make debt consideration part of normal development rhythms and practices and not something separate. I have found this type of integration of debt management important for success. The steps also help bring focus to the debt that affects the developers the most. A portion of the product with high debt in terms of principal but that functions correctly and requires no modification will generally be ignored by this approach, which is appropriate because that debt effectively has no interest payment.

So, if you aren't sure where to start or if you have a lot of barriers, start small. Add a few basic debt identification, quantification, remediation, and prevention techniques to your normal development rhythms. Then use what you've learned to improve over time, increasing the use of techniques and strategies that provide the most value.

What Can You Do Today?

Not following established software engineering practices will result in reckless and unintentional technical debt. This chapter highlights essential practices that you can incorporate into any development effort. Doing so will help you avoid unintentional technical debt and take on intentional debt strategically. Embrace and educate your teams continuously on bread-and-butter software engineering practices such as code review, unit testing, and coding standards. And consider automating these practices, including static analysis of the codebase.

For Further Reading

The concept of the architectural runway is a key practice in SAFe (Leffingwell 2007). In addition, Stephany Bellomo et al. (2014) describe how to "agilely" design an architecture, an approach in which reducing the Big Up-Front Design should minimize the premature technical debt that is incurred.

Adopting the right software development practices for your project will definitely have a payoff in the long run. For example, Forsgren, Humble, and Kim emphasize the benefits of adopting a DevOps practice in their 2017 DORA report.

Refactoring existing code enables a development team to get ahead of unintentional technical debt. Scott Ambler (2017) lists this as one of his 11 strategies for dealing with technical debt as well. Three books provide in-depth review of related strategies: Martin Fowler's *Refactoring* (2018), Joshua Kerievski's *Refactoring to Patterns* (2004), and Michael Feathers' *Working Effectively with Legacy Code* (2004).

Meeting high standards and enforcing good software craftsmanship are increasingly important in our software-intensive world. The ideas presented in this chapter are embodied in work that discusses how developers should understand and enforce high standards for their implementation practices. Robert Martin's *Clean Code: A Handbook of Agile Software Craftsmanship* (2008) explains the fundamentals of writing clean, maintainable code and provides examples. See also Sandro Mancuso's *The Software Craftsman: Professionalism, Pragmatism, Pride* (2014).

Chapter 13

Living with Your Technical Debt

In this final chapter, we describe how you can continue exploring the technical debt landscape and how to make the management of technical debt an integral part of your product development activities.

Your Technical Debt Toolbox

By now, you've developed a more comprehensive idea of what technical debt is about and how it affects your software development projects. It is likely that your projects suffer from technical debt to some extent. But if managed well, technical debt can be an effective design strategy.

Throughout the book, we have recommended activities that you can do today. But you may still be asking how all these activities fit together in your particular situation. The answer is "It depends!" It depends on your role, on the impact the debt has on the project, on the age of your system, and primarily on where the project needs to go from where it stands today. While the range of actions you can take is wide, here is a generic path:

1. **Become aware:** Ensure that all the people involved have a common understanding of what technical debt is and how it affects any project.
2. **Assess the information:** Understand the state of the project, what debt you are currently facing, what causes it, and what its consequences are.
3. **Build a registry:** Build some form of inventory of technical debt.

4. **Decide what to fix:** Look over the technical debt registry as you plan a release for items that will reduce your technical debt and that you will actually tackle.
5. **Take action:** Include technical debt identification and management in all software development and business governance practices.

Repeat this process, as it is unlikely that you will escape completely debt-free in one shot.

Does this sound daunting? It does not have to be. At the end of each chapter, we have introduced some ideas for simple undertakings. Here we revisit them so you can think more carefully about living with your debt going forward. Having these practices in your technical debt toolbox will assist you in managing your technical debt.

Depending on the context of your development effort (mostly size, age of the system, and external factors, such as the domain), your software development lifecycle process may be more or less explicit or formal. If you want to explicitly manage the roles, activities, and artifacts of technical debt in your organization, do not make dealing with debt a separate process. Integrate it into your process to complement your current practices.

Proceed in steps, not as a massive change. As the first step, choose what can bring the most immediate benefits. Consider the effect this choice will have on the people involved and any learning required. Keep the technical debt timeline in mind as a guide (see Figure 13.1).

Become Aware

Put a name to your technical debt. Ensure that all the people involved in the project or close to it have a common understanding of technical debt: what it is, what it is not, and how it affects the project. This is important because today many people

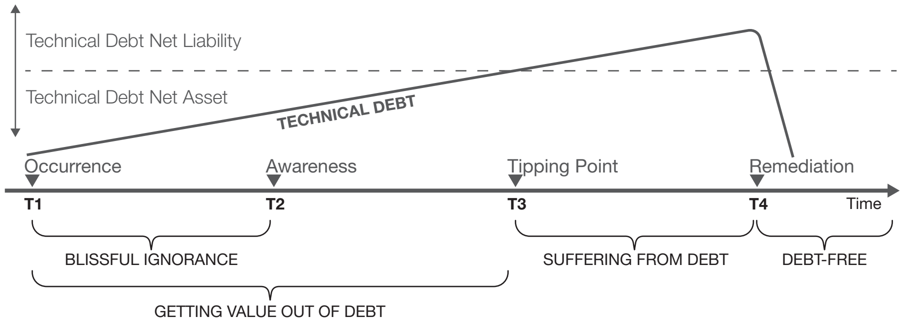


Figure 13.1 *Timeline for an organization incurring unintentional technical debt*

have heard about technical debt from various sources and developed their own ideas of what it means. Use the definition we give in Chapter 2, “What Is Technical Debt?”

The following are some ways to raise awareness:

- Provide a clear, simple definition of technical debt in the context of your project.
- Educate the team about technical debt and its causes.
- Educate the people in the immediate project environment (management, analysts, and product managers) about technical debt.
- Create a “techdebt” category in your issue tracking system, distinct from defects or new features.
- Include known technical debt as part of your long-term technology roadmaps.
- Extend awareness activities to any external contractors who are part of the project.

Use approaches that increase team communication and help get everyone on the same page conceptually. You might want to organize a lunch-and-learn session with your team to introduce the concept of technical debt. Illustrate this with examples from your own project. A Technical Debt Credit Check can provide a quick look at the overall project and guidance on where to start (see Chapter 11, “Technical Debt Credit Check”).

Assess the Information

Before attempting technical debt remediation, objectively assess the state of the project. Depending on where you are on the technical debt timeline, you may consider the following possibilities:

- Establish the goals and criteria against which technical debt will be assessed (see the section “Understanding the Business Context for Assessing Technical Debt” in Chapter 4, “Recognizing Technical Debt”).
- Monitor your portfolio by analyzing code, architecture, and production infrastructure to understand the technical debt responsible for the symptoms the team is experiencing (see Chapters 5, “Technical Debt and the Source Code,” 6, “Technical Debt and Architecture,” and 7, “Technical Debt and Production”).
- Incorporate lightweight checks to continuously monitor technical debt (see Chapter 12, “Avoiding Unintentional Debt”).

A wide range of activities, including the following, can help you assess the information you gather about technical debt:

- Understand the business context to guide the use of source code and analysis tools as input for technical debt analysis.
- Create coding, architecture, and production infrastructure standards to serve as the yardstick against which technical debt is measured. Establish thresholds to identify when the debt level is becoming too high.
- Develop tests and traceability: requirements, design/code, tests, and test results.
- Use tool support to check and enforce some of these guidelines or standards. Deploy a static code analyzer to detect code smells. And do not panic in the face of large numbers of warnings. We gave you some strategies to prioritize these in Chapter 5.
- Review the architecture. If it is not documented, glean insights from team knowledge, source code, and the issues being tracked. Use your knowledge of architectural risk to guide automated analysis of the source code.
- When fixing a defect or adding a new feature request, look beyond the immediate implementation to see if longer-term design issues could lead to technical debt.
- Organize one-hour brainstorming sessions around the question "What design decision did we make that we now regret because it is costing us more than we estimated?" or "If we had to do it again, what should we have done?"

Assessing the information is not a blame game or a whining session; just identify high-level structural issues, the key design decisions from the past that have turned into technical debt today. Later the results will help you determine the impact that technical debt has on your project.

Build a Technical Debt Registry

Introduce a gradual means to build an inventory of technical debt items (see the section "Writing a Technical Debt Description" in Chapter 4):

- Refine the "techdebt" category in your issue tracker into a technical debt description. Point at the specific software artifacts involved: code, architecture, or production infrastructure.

- Include at least the most common two technical debt subcategories: (1) simple, localized, code-level debt and (2) wide-ranging, structural, architectural debt. Point at the software artifacts involved: code, architecture, or production infrastructure.
- Standardize on a single form of "Fix me" or "Fix me later" comment in the source code to mark places that should be revised later. With such comments, they will be easier to spot by using a tool.
- Analyze the code and the architecture for the presence of unintentional technical debt and describe the findings in the technical debt registry.
- Develop a strategy for prioritizing debt remediation and ensuring that it isn't starved.
- Include technical debt discussions during your iteration reviews and retrospectives and note any technical debt items. Prioritize them as part of your backlog.
- As you pursue development, make sure to introduce *intentional* technical debt items in your registry at the point where you make the decision to incur such debt.

You may have to do a little bit of this inventory work as you gather information to get some concrete examples and input data for your assessment.

Decide What to Fix

If you are facing a large and somewhat varied registry of technical debt, you need to decide what to fix and when. You should base these decisions on your assessment of the situation, including where you want to bring your software product next. To make your decisions, you need to gather additional information about remediation strategies, cost, and the trade-offs involved (see Chapter 8, "Costing the Technical Debt").

Review items in the technical debt registry to ensure that it contains the appropriate items and that they are prioritized to help with the decision of what to deliver next:

- Refine technical debt items to the level of "story cards" on your backlog and make them an integral part of your release planning and iteration planning. Organize your backlog to reflect the four categories of items it contains (see the sidebar "What Color Is Your Backlog?" in Chapter 4).

- Estimate not only the cost to pay the technical debt items but also the cost to not pay them: How much will deferring repayment slow current progress? If you are not able to provide an actual cost, use some “t-shirt sizing” strategy.
- Allocate time to service technical debt. You might start with 15% of your iteration budget, but you need to keep in mind that one size does not fit all. At times you might need to allocate a whole sprint to technical debt reduction; at other times you might need a lot less time. Monitor your progress and learn from your experience.
- Some complex technical debt will have aspects that relate to rework in code, architecture, and infrastructure. Reduction may require more significant rearchitecting or systemwide refactoring, which you may have to spread across several iterations.
- Put a context-dependent payment plan in place; repaying all debt, except in very small projects, is simply not feasible and is also not the best use of resources.
- Show the value of technical debt reduction tasks by how they support high-value change requests for new features or defect resolution. When choosing among refactorings, opt for the change that will offer more flexibility for the future and support more potential evolutions, when economically feasible.
- Prioritize technical debt items to fix by doing them first in the parts of your codebase that are the most actively modified. If a subsystem or module will not be modified as a result of a change scenario in the foreseeable future, do not fix any technical debt in it, unless the change is a consequence of fixing the technical debt in a module it depends on.

What you will do now about technical debt can be fully integrated with your release and iteration planning. As you pay down unintentional debt, you may also begin to keep or take on some debt with intention as you seek to proactively manage technical debt (see Chapter 9, “Servicing the Technical Debt”).

Take Action

An integrated technical debt management approach uses your knowledge of your project context to apply practices specific to your situation. It adds proactive measures to understand the causes of technical debt and control the introduction of new debt (see Chapters 10, “What Causes Technical Debt?” and 11, “Technical Debt Credit Check”). Software engineering practices that are essential to any development effort will not only help you address the causes of unintentional technical debt but will also help you avoid it (see Chapter 12).

Take action to identify and manage technical debt in software development and business governance practices. The following are some example of the action you might take:

- Aim to reduce technical debt at each development cycle by bringing some technical debt items into your iteration backlog to keep the level of technical debt low.
- Develop an approach for systematic regression testing so that fixing technical debt items does not risk breaking the code. (This will remove the objection that "It is not really badly broken, so I won't fix it.")
- Assess your current practices from the perspectives of business, architecture, development, and organization and identify sources of technical debt to eliminate.
- Factor technical debt into business decisions about the opportunity cost of delaying features and reducing risk liability.
- Consider taking on intentional technical debt as a short-term or long-term investment, where appropriate, and plan to manage it.
- Include indicators of technical debt in any management or overview project dashboard.
- Gather key measures of effort or cost associated with technical debt elements to assist in future decision making.

Start simple and iterate through these activities, incrementally improving the process at each iteration and adding sophistication.

Integrating technical debt into your current software development process does not need to change the lifecycle. You may not create new artifacts, except guidelines or standards, if you are missing them. Do not create new roles, apart from a possible "technical debt evangelist" or "technical debt champion" at the beginning. You may, however, deploy new techniques and tools to support some of the activities, and we have enumerated a few throughout the previous chapters.

On the Three Moons of Saturn…

Let's look at what happened in the three companies we have been using as examples in this book and what we recommend they do now.

Atlas: The Small Startup

The Atlas company became aware of technical debt when it grew to be 15 developers and experienced some difficulties evolving its product for more diverse consumers. The 4 founders had not been aware of the accumulation of debt due to the constant "pivoting" during their early years. More recent hires made the team aware of this, but the debt remained for many months a vague concept, not brought to any concrete action and only a subject of occasional discussion.

One of the founders brought in a consultant who made an initial assessment and delivered presentations to the whole team. The concept of technical debt became clearer for everyone. Atlas acquired a static analysis tool (SonarQube) and a structural analysis tool (Structure 101) and employed a summer student intern to gather data about debt in the project. The results of this side project showed that the team could take some easy actions to mitigate some of the major technical debt. These actions were introduced into two development iterations and involved explicitly putting technical debt story cards on the backlog. But failing to go further led to conflicting priorities later, when the team tried to resolve the debt: Some of the major structural technical debt items are still there, and remediating them seems too daunting to actually tackle.

The Atlas team should now plan to do the following:

- Refine its development process to systematically capture technical debt items.
- Integrate the big-ticket technical debt items into the planning for future major releases by evaluating the costs associated with remediation and nonremediation. Some of these items will require allocating several sprints of effort to complete.
- Systematically reduce technical debt items for code smells in each iteration or most iterations. Train the developers to ensure that they do not inject new code smells.

Phoebe: The Agile Shop with a Successful Product

Phoebe evolved within the "agile" movement, using an improved version of Scrum. Most developers were aware of technical debt, and from the beginning, they systematically included small technical debt items on the backlog. But as the product became successful and the core team began to shrink in size, with much of the development done by external partners, technical debt grew, especially at the structural level. Today the Phoebe team struggles to manage multiple stakeholders with diverse requirements, get ahead of changing technology, and sustain a viable product. As a result, technical debt is accruing, in most cases intentionally.

While the Phoebe team has been trying to repay the debt by prioritizing technical debt reduction in major releases, technology lock-in has become a major hindrance to meeting this goal; decreasing staff size has also been a hindrance. There is also a lot of inconsistency in how the core team identifies and manages the technical debt. For example, the team tried using some tools to look into the code quality but did not sustain their use. Major refactoring releases have eliminated some of the existing technical debt or made it obsolete, but Phoebe has not communicated this broadly to its stakeholders, and it is not clear what the team is handling as the top-priority issues.

The Phoebe team should now plan to do the following:

- Raise awareness of technical debt and its impact with partners in its ecosystem.
- Explicitly integrate technical debt management in the process used with and by the other partners in the ecosystem, including specified tools.
- Add technical debt indicators in the project dashboard.
- Integrate the big-ticket technical debt items in the planning for future major releases by evaluating the costs associated with remediation and non-remediation.

Tethys: The Global Giant

For many years at Tethys, technical debt was the elephant in the room. Most senior developers who had been with the project for several years were acutely aware of it, even though they did not call it "technical debt." They would in private gladly discuss with visitors or newcomers some of the technical debt and when it was intentionally incurred. But from a planning perspective, technical debt and its possible remediation were never on the table and never discussed with the company technical leadership or the business part of the company.

The code is of rather high quality. The company routinely uses various tools to assess the code quality and conformance to its coding and design standards. The technical debt is the result of both intentional structural debt accumulated over the years and a technological shift: Major design choices now look bad due to the evolution of technology over some 15 years.

The Tethys team should plan to do the following:

- Agree on a systematic way to identify and capture technical debt, particularly complexity of the architecture and technological shift.
- Develop simple means, like t-shirt sizing to start with, to associate remediation and non-remediation cost to major technical debt items.

- Make upper levels of management aware of technical debt and its impact on the business.
- Involve the product management team in decision making about technical debt reduction and decisions about intentionally taking on more technical debt.
- Conduct a Technical Debt Credit Check and include management and product management in the assessment.

Technical Debt and Software Development

Your software development organization will become gradually “technical debt aware.” Ultimately, managing technical debt will become an integral part of your software development.

If your organization is just beginning the journey toward managing technical debt, your startup costs will be higher. To get started, at a minimum, you should plan to do the following:

1. Understand the state of the development process and its alignment with business goals (see Chapter 10).
2. Identify technical debt items (see Chapters 5, 6, and 7), including selecting practices and tools to support this activity.
3. Incorporate technical debt as a major input into software development decisions (see Chapter 9).
4. Educate all stakeholders inside the development team and at its immediate periphery about technical debt and its consequences (see Chapter 4).

If your organization is already technical debt aware, you can plan to do the following:

1. Readily identify intentional debt at the point of occurrence and record it (see Chapters 10 and 12).
2. Regularly monitor the design and the code for new and accumulating technical debt and record any detected technical debt items (see Chapters 5, 6, 7, and 11).
3. Spread debt reduction across the development lifecycle (see Chapter 9).
4. Collect metrics on indicators that may point to symptoms of technical debt, such as velocity, lingering defects, and high development estimates (see Chapter 4).

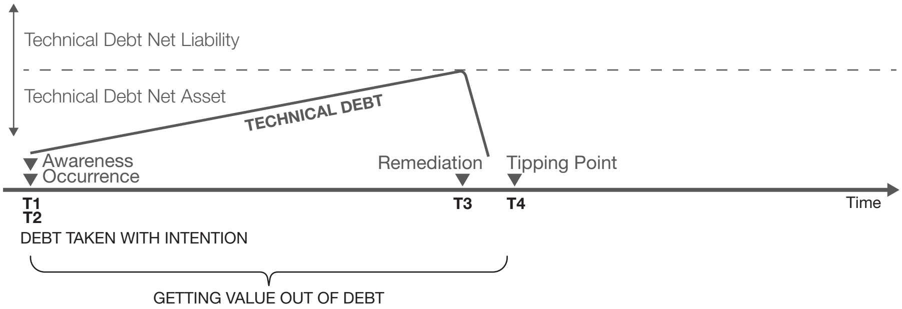


Figure 13.2 *Timeline for a technical debt-aware organization*

The timeline for a technical debt-aware organization shown in Figure 13.2 shows technical debt being taken on, monitored, and remediated with intention.

An iterative development lifecycle, which is a core feature of agile approaches, offers better opportunities to manage technical debt continuously. The repayment of small technical debt items can be spread over multiple iterations in a single release cycle. However, larger technical debt items—such as architectural ones—may not fit easily in a short iteration cycle. You may be tempted to defer them because they are too disruptive to the rapid pace of development. (This is often the case with architectural activities and is not specific to the architectural debt reduction.) Resist the temptation to incur lots of technical debt in an effort to be more responsive to change requests.

Finale

Technical debt is at the root of the friction we described in Chapter 1, “Friction in Software Development”: the phenomenon that gradually slows down software development teams. Technical debt is simply unavoidable, especially in large and long-lived systems, and even more so in successful systems.

Technical debt has proven to be a useful concept for helping developers and managers approach these issues. Drawing from a financial metaphor, the concept of technical debt shifts decision making from a strict economic standpoint or a pure technical standpoint to a place where various parties can better understand the trade-offs and compromises, assess the current state of development, and determine the way forward.

Technical debt can be an effective tool to sprint to a major short-term milestone—achieving some success very rapidly by borrowing time from the future—and, in this sense, it looks more like debt used as an investment. The problems start to arise later, when the debt is quickly forgotten and not repaid promptly.

In this book, we have identified a small number of principles to help you better understand and deliberately manage technical debt:

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.

Sustaining the pace of innovation while ensuring software quality involves establishing technical debt management as a core software engineering practice. There is growing interest in research, practice, and tool support for managing technical debt. Most of this research can be done only in an industrial environment; the kind of issues we are dealing with cannot be reproduced in a small laboratory experiment. We invite you to join the technical debt community and contribute case studies, stories of technical debt, and practices associated with managing it. A good starting point is the website techdebtconf.org.

Glossary

accruing interest—Additional costs incurred by building new software that depends on an element of technical debt, a nonoptimal solution. These costs accumulate over time into the initial principal to lead to the current principal (accretion).

artifact—*See* development artifact.

business goal—A high-level objective that a stakeholder in a software product development effort wants to achieve.

cause—The process, decision, action, lack of action, or event that triggers the existence of a technical debt item.

compounding interest—*See* accruing interest.

consequence—The effect on the value, quality, or cost of the current or the future state of a system associated with technical debt items.

context—The set of economic, sociological, cultural, and technical factors that are not strictly under control of the project but have a strong influence on the project evolution.

cost—The cost of developing or maintaining a product, which mostly consists of paying the people who work on it. Cost is often approximated for planning purposes by using a system of points as a proxy for actual financial cost.

developer—Any person involved directly with the development of software: architects, designers, coders, testers, and so on.

development artifact—An element of a system or the supporting work products: design, code, documentation, tests, defect records, and so on.

feature—A chunk of functionality that delivers business value.

interest—*See* accruing interest and recurring interest.

point—A unit of measure of the development cost of a planned system.

principal—The cost savings gained by taking some initial approach or shortcut in development (the initial principal) or the cost it would take now to develop a solution (the current principal).

product—A complete system that is ready to be delivered or commercialized.

quality—The degree to which a system, component, or process meets customer or user needs or expectations (IEEE Standard 610).

recurring interest—Additional costs incurred by a project in the presence of technical debt, due to reduced productivity (or velocity), induced defects, or loss of quality (maintainability and evolvability). These are sunk costs that are not recoverable through remediation.

registry—For a software system, an inventory of technical debt items typically stored in a tool, such as an issue tracker or project backlog management system.

remediation—The removal of a technical debt item. Its cost is the associated current principal and any accrued interest associated with it.

stakeholder—Any party—person or organization—that is affected by or that influences a development project.

symptom—An observable qualitative or measurable consequence of technical debt items.

system—A set of connected, engineered artifacts that form a complex whole. In this book, *system* refers to the software-intensive system under development that will ultimately become the product.

technical debt—1. The complete set of technical debt items associated with a system. 2. In software-intensive systems, 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.

technical debt description—A systematic way of capturing a technical debt item and its (known) properties.

technical debt item—One atomic element of technical debt connecting a set of development artifacts with consequences for the quality, value, and cost of the system and triggered by one or more causes related to process, management, context, business goals, and so on.

value—The business value derived from the ultimate consumers of the product: its users, or acquirers, the people who are going to pay to use it, and the perceived utility of the product.

References

Alan, D. R., Wixom, B. H., & Tegarden, D. (2012). *Systems analysis and design with UML, Version 2.0: An object-oriented approach* (4th ed.). Hoboken, NJ: Wiley.

Al-Barak, M., & Bahsoon, R. (2016). Database design debts through examining schema evolution. In *Proceedings of the IEEE Eighth International Workshop on Managing Technical Debt* (pp. 17–23). Piscataway, NJ: IEEE Computer Society Press.

Albrecht, A. J., & Gaffney, J. R. (1983). Software function, source lines of code, and development effort prediction: A software science validation. *IEEE Transactions on Software Engineering, 9*(6), 639–648.

Ambler, S. (2017). 11 strategies for dealing with technical debt. Retrieved from http://www.disciplinedagiledelivery.com/technical-debt/

Ambler, S. W. (2011). Agility at scale. Become as agile as you can be. Retrieved from https://www.ibm.com/developerworks/community/blogs/ambler?lang=en

Ampatzoglou, A., Ampatzoglou, A., Chatzigeorgiou, A., Avgeriou, P., Abrahamsson, P., Martini, A., …Systä, K. (2016). The perception of technical debt in the embedded systems domain: An industrial case study. In *Proceedings of the 2016 IEEE 8th International Workshop on Managing Technical Debt* (pp. 9–16). Piscataway, NJ: IEEE Computer Society.

Arcelli-Fontana, F., Ferme, V., Zanoni, M., & Roveda, R. (2015). Towards a prioritization of code debt: A code smell Intensity Index. In *Proceedings of the IEEE Seventh International Workshop on Managing Technical Debt* (pp. 16–24). Piscataway, NJ: IEEE Press.

Arulraj, J. (2018). SQL Check. https://github.com/jarulraj/sqlcheck

Avgeriou, P., Kruchten, P., Ozkaya, I., & Seaman, C. (Eds.). (2016). Managing technical debt in software engineering (Dagstuhl Seminar 16162). *Dagstuhl Reports* (Vol. 6, pp. 110–138). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik. http://dx.doi.org/10.4230/DagRep.6.4.110

Baldwin, C. Y., & Clark, K. B. (2000). *Design rules: The power of modularity.* Cambridge, MA: MIT Press.

Basili, V. R., Caldiera, G., & Rombach, D. (1994). The Goal Question Metric approach. In R. Van Solingen (Ed.), *Encyclopedia of Software Engineering* (pp. 528–532). Hoboken, NJ: Wiley.

Bass, L., Clements, P., & Kazman, R. (2012). *Software architecture in practice* (3rd d.). Reading, MA: Addison-Wesley.

Bass, L., Weber, I., & Zhu, L. (2016). *DevOps: A software architect's perspective*. Boston: Addison-Wesley Professional.

Bavota, G., & Russo, B. (2016). A large-scale empirical study on self-admitted technical debt. In *Proceedings of the 2016 IEEE/ACM 13th Working Conference on Mining Software Repositories* (pp. 315–326). New York: ACM.

Becker, C., Chitchyan, R., Duboc, L., Easterbrook, S., Penzenstadler, B., Seyff, N., & Venters, C. (2015). Sustainability design and software: The Karlskrona Manifesto. In *Proceedings of the 37th International Conference on Software Engineering, Vol. 2* (pp. 467–476). Piscataway, NJ: IEEE Press.

Bellomo, S., Kruchten, P., Nord, R. L., & Ozkaya, I. (2014). How to agilely architect an agile architecture. *Cutter IT Journal, 27*(2), 12–17.

Bellomo, S., Nord, R. L., Ozkaya, I., & Popeck, M. (2016). Got technical debt? Surfacing elusive technical debt in issue trackers. In *Proceedings of the 2016 IEEE/ACM 13th Working Conference on Mining Software Repositories* (pp. 327–338). New York: ACM.

Bergey, J., Cohen, S., Donohoe, P., & Jones, L. (2005). *Software product lines: Experiences from the Seventh DoD Software Product Line Workshop* (CMU/SEI-2005-TR-001). Pittsburgh: Software Engineering Institute.

Beyer, B., Jones, C., Petoff, J., & Murphy, N. R. (Eds.). (2016). *Site reliability engineering*. Sebastopol, CA: O'Reilly.

Boehm, B., Clark, B. K., Brown, A. W., & Abts, C. (2000). *Software cost estimation with Cocomo II*. Upper Saddle River, NJ: Prentice Hall.

Booch, G. (2000). The future of software (abstract). In *Proceedings of the 22nd International Conference on Software Engineering* (p. 3). New York: ACM.

Brooks, F. P. (1986). No silver bullet—Essence and accident in software engineering. In H.-J. Kugler (Ed.), *Proceedings of the IFIP Tenth World Computing Conference* (pp. 1069–1076). Amsterdam: Elsevier Science B.V.

Brooks, F. P. (1995). *The mythical man-month: Essays on software engineering* (anniversary ed.). Reading, MA: Addison-Wesley.

Brown, S. (2018). *Software architecture for developers*. Vancouver, BC: Leanpub.

Cervantes, H., & Kazman, R. (2016). *Designing software architectures: A practical approach*. Boston: Addison-Wesley Professional.

Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R.,…Stafford, J. (2011). *Documenting software architectures: Views and beyond* (2nd ed.). Reading, MA: Addison-Wesley.

Clements, P., Kazman, R., & Klein, M. (2001). *Evaluating software architectures: Methods and case studies*. Reading, MA: Addison-Wesley.

Cohn, M. (2016). What are story points? Retrieved from https://www.mountaingoatsoftware.com/blog/what-are-story-points

Consortium for IT Software Quality (CISQ). http://it-cisq.org/

Cunningham, W. (1992). The WyCash Portfolio Management System. Presented at OOPSLA 1992, Vancouver. Available at http://dl.acm.org/citation.cfm?id=157715

Davis, A. M. (1995). *201 principles of software development*. New York: McGraw-Hill.

Dijkstra, E. W. (1972). The humble programmer. *Communications of the ACM, 15*(10), 859–866.

Fairbanks, G. (2010). *Just enough software architecture: A risk-driven approach*. Boulder, CO: Marshall & Brainerd.

Fayolle, J. P., Coq, T., & Letouzey, J.-L. (2018). The Agile Alliance debt analysis model (A2DAM). Agile Alliance. Retrieved from https://www.agilealliance.org/the-agile-alliance-debt-analysis-model/

Feathers, M. (2004). *Working effectively with legacy code*. Upper Saddle River, NJ: Pearson Education.

Ford, N., Parsons, R., & Kua, P. (2017). *Building evolutionary architectures: Support constant change*. Sebastopol, CA: O'Reilly Media.

Forsgren, N., Humble, J., & Kim, G. (2017). *Forecasting the value of DevOps transformations: Measuring ROI of DevOps*. Beaverton, OR: DevOps Research and Assessment (DORA).

Fowler, M. (2003). Technical debt. Retrieved from http://martinfowler.com/bliki/TechnicalDebt.html

Fowler, M. (2009). Technical debt quadrants. Retrieved from https://martinfowler.com/bliki/TechnicalDebtQuadrant.html

Fowler, M. (2018). *Refactoring: Improving the design of existing code*. Reading, MA: Addison-Wesley.

Freeman, S., & Matt, C. (2014). Is unhedged call options a better metaphor for bad code? InfoQueue. Retrieved from https://www.infoq.com/news/2014/12/call-options-bad-code

Gibbs, W. W. (1994). Software's chronic crisis. *Scientific American, 271*(3), 72–81.

Glass, R. L. (2003). *Facts and fallacies of software engineering*. Boston: Addison-Wesley.

Google. (2018). Google Java standard guide. Retrieved from https://google.github.io/styleguide/javaguide.html

Gorton, I. (2006). *Essential software architecture* (2nd ed.). Berlin: Springer.

Grenning, J. (2002). Planning poker. Retrieved from https://wingman-sw.com/papers/PlanningPoker-v1.1.pdf

Guo, Y., Spínola, R. O., & Seaman, C. B. (2016). Exploring the costs of technical debt management: A case study. *Empirical Software Engineering, 21*(1), 159–182.

Hastie, S. (2010). What color is your backlog? *InfoQ Magazine*. Retrieved from http://www.infoq.com/news/2010/05/what-color-backlog

Highsmith, J. A. (2002). *Agile software development ecosystems*. Boston: Addison-Wesley.

Highsmith, J. A. (2010). The financial implications of technical debt. Retrieved from http://jimhighsmith.com/the-financial-implications-of-technical-debt/

International Organization for Standardization/International Electrotechnical Commission (IEC). (2009). *ISO/IEC 20926:2009 Software and systems engineering—Software measurement—IFPUG functional size measurement method*. Geneva, Switzerland: ISO.

International Organization for Standardization/International Electrotechnical Commission. (2011). *ISO/IEC 25010:2011 Systems and software engineering—Systems and software Quality Requirements and Evaluation (SQuaRE)—System and software quality models*. Geneva, Switzerland: ISO/IEC. https://www.iso.org/standard/35733.html

Kazman, R., Cai, Y., Mo, R., Feng, Q., Xiao, L., Haziyev, S.,…Shapochka, A. (2015). A case study in locating the architectural roots of technical debt. In *Proceedings of the International Conference on Software Engineering (ICSE '15)*, Vol. 2. (pp. 179–188). Piscataway, NJ: IEEE Press.

Kerievski, J. (2004). *Refactoring to patterns*. Boston: Addison-Wesley.

Kim, G., Behr, K., & Spafford, G. (2013). *The Phoenix Project: A novel about IT, DevOps, and helping your business win*. Portland, OR: IT Revolution Press.

Kim, G., & Debois, P. (2016). *The DevOps handbook: How to create world-class agility, reliability, and security in technology organizations*. Portland, OR: IT Revolution Press.

Klotins, E., Unterkalmsteiner, M., Chatzipetrou, P., Gorschek, T., Prikladnicki, R., Tripathi, N., & Pompermaier, L. B. (2018). Exploration of technical debt in start-ups. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice* (pp. 75–84). New York: ACM.

Knodel, J., & Naab, M. (2016). *Pragmatic evaluation of software architecture*. Berlin: Springer.

Kruchten, P. (2011). The (missing) value of software architecture. Retrieved from http://philippe.kruchten.com/2013/12/11/the-missing-value-of-software-architecture/

Kruchten, P. (2013). Contextualizing agile software development. *Journal of Software Evolution and Process, 25*(4), 351–361.

Kruchten, P., Nord, R., & Ozkaya, I. (2012). Technical debt: From metaphor to theory and practice. *IEEE Software, 29*(6), 18–21.

Kruchten, P., Nord, R. L., Ozkaya, I., & Falessi, D. (2013). Technical debt: Towards a crisper definition; report on the 4th International Workshop on Managing Technical Debt. *SIGSOFT Software Engineering Notes, 38*(5), 51–54.

Leffingwell, D. (2007). *Scaling software agility: Best practices for large enterprises*. Boston: Addison-Wesley.

Lehman, M. M. (1980). On understanding laws, evolution, and conservation in the large program lifecycle. *Journal of Systems and Software, 1*(3), 213–221.

Lehman, M. M. (1996). Laws of software evolution revisited. In C. Montangero (Ed.), *Software Process Technology: Fifth European Workshop* (pp. 108–124). Berlin: Springer.

Letouzey, J.-L. (2016). The SQALE method for managing technical debt: Definition document (Version 1.1). Retrieved from http://www.sqale.org/wp-content/uploads/2016/08/SQALE-Method-EN-V1-1.pdf

Letouzey, J.-L., & Ilkiewicz, M. (2012). Managing technical debt with the SQALE method. *IEEE Software, 29*(6), 44–51.

Li, Z., Liang, P., & Avgeriou, P. (2015). Architectural technical debt identification based on architecture decisions and change scenarios. In *Proceedings of the 12th Working IEEE/IFIP Conference on Software Architecture* (WICSA '15) (pp. 65–74). Piscataway, NJ: IEEE.

Lim, E., Taksande, N., & Seaman, C. B. (2012). A balancing act: What software practitioners have to say about technical debt. *IEEE Software, 29*(6), 22–27.

Mancuso, S. (2014). *The software craftsman: Professionalism, pragmatism, pride*. Upper Saddle River, NJ: Prentice Hall.

Martin, R. (2008). *Clean code: A handbook of agile software craftsmanship*. Upper Saddle River, NJ: Prentice Hall.

McConnell, S. (2007). Technical debt. Retrieved from https://www.construx.com/resources/whitepaper-managing-technical-debt/

Morris, K. (2016). *Infrastructure as code: Managing servers in the cloud*. Sebastopol, CA: O'Reilly.

Object Management Group. (2017). Automated technical debt measure, Version 1, Beta 2 (OMG Document Number: admtf/2017-03-01). Retrieved from http://www.omg.org/spec/ATDM/

Parnas, D. L. (1994). Software aging. In *Proceedings of the 16th International Conference on Software Engineering* (pp. 279–287). Los Alamitos, CA: IEEE Computer Society.

Poort, E. (2014). The business case for technical debt reduction. Retrieved from https://eltjopoort.nl/blog/2014/01/27/the-business-case-for-technical-debt-reduction/

Poort, E. (2016). Just enough anticipation: Architect your time dimension. *IEEE Software, 33*(6), 11–15.

Potdar, A., & Shihab, E. (2014). An exploratory study on self-admitted technical debt. In *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution* (pp. 91–100). Piscataway, NJ: IEEE Press.

Ramasubbu, N., & Kemmerer, C. (in press). Integrating technical debt management and software quality management processes: A normative framework and field tests. *IEEE Transactions on Software Engineering*.

Redgate Software Ltd. (2018). SQL Code Guard, 3.0. https://www.red-gate.com/products/sql-development/sql-code-guard/

Reifer, D. (2001). *Making the software business case: Improvement by the numbers*. Upper Saddle River, NJ: Addison-Wesley.

Ries, E. (2011). *The lean startup*. New York: Crown Business.

Rozanski, N., & Woods, E. (2012). *Software systems architecture: Working with stakeholders using viewpoints and perspectives* (2nd ed.). Upper Saddle River, NJ: Addison-Wesley.

Sadowski, C., Aftandilian, E., Eagle, A., Miller-Cushon, L., & Jaspan, C. (2018). Lessons from building static analysis tools at Google. *Communications of the ACM, 61*(4), 58–66.

Scaled Agile, Inc. (n.d.). Scaled Agile Framework (SAFe). Retrieved from https://www.scaledagileframework.com/

Schmid, K. (2013a). A formal approach to technical debt decision making. In *Proceedings of the Ninth International ACM Sigsoft Conference on Quality of Software Architectures* (pp. 153–162). New York: ACM.

Schmid, K. (2013b). On the limits of the technical debt metaphor: Some guidance on going beyond. In *Proceedings of the Fourth International Workshop on Managing Technical Debt* (pp. 63–66). Washington, DC: IEEE Computer Society Press.

Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D.,…Dennison, D. (2015). Hidden technical debt in machine learning systems. In C. Cortes, D. D. Lee, M. Sugiyama, & R. Garnett (Eds.), *Proceedings of the 28th International Conference on Neural Information Processing Systems, Volume 2* (pp. 2503–2511). Cambridge, MA: MIT Press.

Séguin, N., Tremblay, G., & Bagane, H. (2012). Agile principles as software engineering principles: An analysis. In C. Wohlin (Eds.), *LNCS 111*, pp. 1–15. Berlin: Springer.

Shafer, A. C. (2010). Infrastructure debt: Revisiting the foundation. *Cutter IT Journal, 23*(10), 36–41.

Sharma, S. (2017). *The DevOps adoption playbook: A guide to adopting DevOps in a multi-speed IT enterprise*. Indianapolis: Wiley.

Software Engineering Institute. (2018). SEI CERT secure coding standards. Pittsburgh: SEI. https://www.securecoding.cert.org

Tom, E., Aurum, A., & Vidgen, R. (2012a). A consolidated understanding of technical debt. In *Proceedings of the European Conference on Information Systems* (Paper 16).

Tom, E., Aurum, A., & Vidgen, R. (2012b). An exploration of technical debt. *Journal of Systems and Software, 86*(6), 1498–1516.

Tornhill, A. (2018). *Software design x-rays: Fix technical debt with behavioral code analysis*. Raleigh, NC: The Pragmatic Bookshelf.

Visser, J., Rigal, S., van der Leek, R., van Eck, P., & Wijnholds, G. (2016). *Building maintainable software: Ten guidelines for future-proof code*. Sebastopol, CA: O'Reilly Media.

Weber, J., Cleve, A., Meurice, L., & Ruiz, F. J. B. (2014). Managing technical debt in database schemas of critical software. In *Proceedings of the Sixth International Workshop on Managing Technical Debt* (pp. 43–46). Piscataway, NJ: IEEE Computer Society Press.

Wikipedia. (2018). Software architecture. https://en.wikipedia.org/wiki/Software_architecture

Zazworka, N., Vetro, A., Izurieta, C., Wong, S., Cai, Y., Seaman, C. B., & Shull, F. (2014). Comparing four approaches for technical debt identification. *Software Quality Journal, 22*(3), 403–426.

Zimmermann, T., Premraj, R., Bettenburg, N., Just, S., Schröter, A., & Weiss, C. (2010). What makes a good bug report? *IEEE Transactions on Software Engineering, 36*(5), 618–643.

This page intentionally left blank

Index

This page intentionally left blank

Pearson
livelessons™

Technical Debt Description

Name	**What** is it? This field is a shorthand name for the technical debt item.
Summary	**Where** do you observe the technical debt in the affected development artifacts, and where do you expect it to accumulate?
Consequences	**Why** is it important to address this technical debt item? Consequences include immediate benefits and costs as well as those that accumulate later, such as additional rework and testing costs as the issue stays in the system and costs due to reduced productivity, induced defects, or loss of quality incurred by building software that depends on an element of technical debt.
Remediation approach	Describe the rework needed to eliminate the debt, if any. **When** should the remediation occur to reduce or eliminate the consequences?
Reporter/assignee	**Who** is responsible for servicing the debt? Assign a person or team. While in most cases the **who** aspect can be trivial, in some situations the debt resolution may need to be assigned to external parties. If remediation is significantly postponed, this field can communicate that decision.