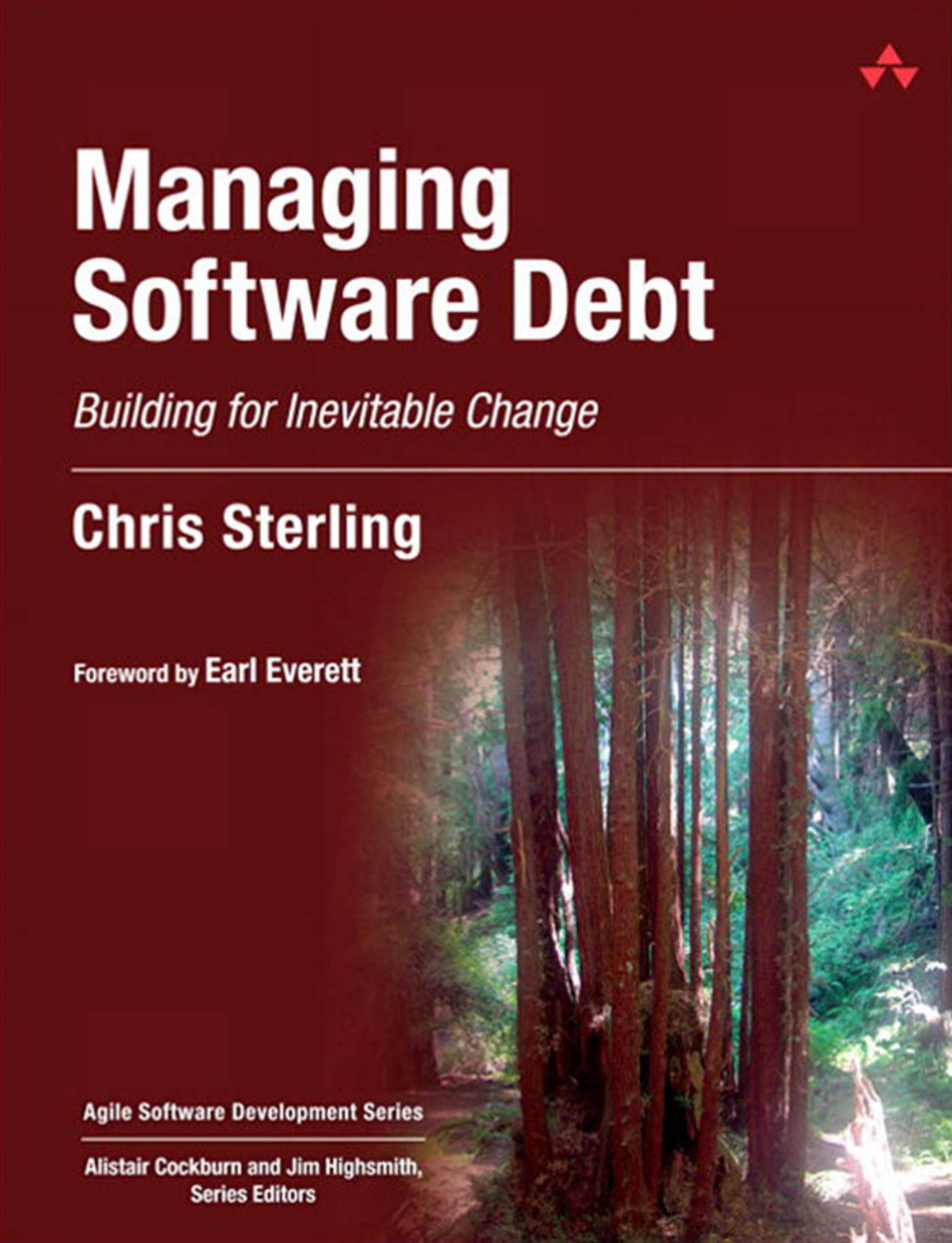


Managing Software Debt

Building for Inevitable Change

Chris Sterling

Foreword by **Earl Everett**

Agile Software Development Series

Alistair Cockburn and Jim Highsmith, Series Editors

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States please contact:

International Sales
international@pearson.com

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data
Sterling, Chris, 1973–
Managing software debt : building for inevitable change / Chris Sterling ; with contributions from Brent Barton.
p. cm.
Includes bibliographical references and index.
ISBN-13: 978-0-321-55413-0 (hardcover : alk. paper)
ISBN-10: 0-321-55413-2 (hardcover : alk. paper)
1. Computer software—Quality control. 2. Agile software development.
3. Software reengineering. I. Barton, Brent. II. Title.
QA76.76.Q35S75 2011
005.1'4—dc22

2010037879

ISBN-13: 978-0-321-55413-0
ISBN-10: 0-321-55413-2
Text printed in the United States on recycled paper at Courier in Westford, Massachusetts.
First printing, December 2010

Contents

This page intentionally left blank

Foreword

Of Fairy Rings, Cathedrals, Software Architecture, and the Tallest Living Life Form on Earth

The imposing, luxuriant, and verdant groves of the coast redwood (*Sequoia sempervirens*) are found today mostly in the foggy valleys along the Pacific Coast of North America from southern Oregon to just south of Monterey, California, in a strip approximately 450 miles long and 20 or so miles wide (725 by 32 km). They are the tallest living species on Earth, reaching heights of 300 to 350 feet (91 to 107 m). The tallest redwood known measures 367 feet (112 m), slightly taller than the *Saturn V* rocket. Redwood forests possess the largest biomass per unit area on Earth, in some stands exceeding 1,561 tons/acre (3,500 metric tons/hectare).

They can also be ancient, predating even the earliest FORTRAN programmers, with many in the wild exceeding 600 years. The oldest verified tree is at least 2,200 years of age, and some are thought to be approximately 3,000 years old. Redwoods first appeared on our planet during the Cretaceous era sometime between 140 and 110 million years ago, grew in all parts of the world, and survived the KT (Cretaceous–Tertiary) event, which killed off more than half the Earth's species 65 million years ago.

Clearly, the coast redwood has been successful. But how? After all, these trees require large amounts of water (up to 500 gallons or 1,893 liters per day), with the significant requirement of transporting much of that up to the height of the tree. Their height turns them into lightning rods, and many fires are struck at their bases during the dry season, fires that often spread underground along their root systems to their neighbors, as well. Rabbits and wild hogs would sneer disdainfully at their reproductive rate, because the trees produce seeds but once per year, and although a mature tree may produce as many as 250,000 seeds per year, only a minuscule number (0.23% to 1.01%) will germinate.

As you would guess, the redwoods have developed a number of architectural features and adaptations to survive and thrive.

The Pacific Coast region they inhabit provides a water-rich environment, with seasonal rains of 50 to 100 inches (125 to 250 cm) annually, and a coastal fog that helps keep the forests consistently damp. Adapting to this fog, redwoods obtain somewhere between 25% and 50% of their water by taking it in through their needles, and they have further adapted by sprouting canopy roots on their branches, getting water from lofty spongelike "soil mats" formed by dust, needles, seeds, and other materials trapped in their branches. Redwoods also have a very sophisticated pumping capability to transport water from their roots to their tops. In the tallest trees, the water's journey can take several weeks, and the tree's "pump" overcomes a negative pressure of 2,000,000 pascals, more than any human pump system is capable of to date.

This abundant fog and rainfall have a downside, however, creating (along with several other factors) a soil with insufficient nutrients. The redwoods have adapted by developing a significant interdependence with the whole biotic forest community to obtain sufficient nutrients, an interdependence that is only beginning to be understood.

The redwood has built bark that is tough and thick—up to 1 foot (30.5 cm) in some places—thickness that, among other things, shields against woodpeckers. Imbued with a rich cocktail of tannins, it is unappetizing to termites and ants. The bark also functions as an ablative heat shield, similar to the heat shields of the Mercury/Gemini/Apollo capsules, and protects the tree from fire.

Young redwoods use sunlight very efficiently (300% to 400% more so than pines, for example) and are capable of rapid growth. With optimal conditions, a sapling can grow more than 6 feet (2 m) in height and 1 inch (2.5 cm) or more in diameter in a growing season. Mature trees under optimal conditions can grow at a rate of 2 to 3 feet (.6 to 1 m), per year, but if the tops are exposed to full sun and drying winds, they will grow only a few inches/centimeters per year. They simply out-compete other trees for the sun's energy.

A major component in the coast redwoods' responsiveness to environmental conditions is their high genetic variability. Like most plants, they can reproduce sexually with pollen and seed cones, with seed production generally beginning at 10 to 15 years of age.

Yet the seeds don't get very far. While the seeds are winged and designed for wind dispersal, they are small, and light (around 5,600 to 8,500 seeds per

ounce [200 to 300 seeds/g]), and are dispersed a distance of only 200 to 400 feet (60 to 120 m) around the parent. When they land, the thick amount of duff (decaying plant debris on the ground) prevents most seeds from ever making it to the dirt.

This accounts for a part of the 1% or less seed germination rate, but a much more significant factor is at work. A large number of the seeds are actually empty! Scientists speculate this could be an adaptation to discourage seed predators, which learn that too much time is wasted sorting the "wheat" (edible seeds) from the "chaff" (empty seeds). It is estimated that only 20% of redwood reproduction occurs sexually through seeds.

The other 80% comes from their capability to reproduce asexually through sprouting, even after severe damage, a feature that has likely played a large part in making the coast redwood such a vibrant and resilient species.

If a redwood is damaged by a fire, lightning strike, or ax, or is dying, a number of sprouts erupt and develop around the circumference of the tree. This nearly perfect circle is known colloquially as a "fairy ring." The sprouts can use the root system and nutrients of the parent and therefore can grow faster than seedlings, gaining 8 feet (2.3 m) in a single season. The sprouts are really "clone trees," and their genetic information may be thousands of years old, dating back to the first parent. Surprisingly, genetic analysis has found diverse gene stocks in the rings, where non-clones (seedlings) have "completed" the circle.

These fairy rings are found around the parent tree's stump, or a depression in the middle of the circle if the stump has decayed completely. They can also be found circling their still-alive and recovered parent tree. The stump of a parent tree inside a fairy ring is known colloquially as a "cathedral," and the fairy rings themselves are also known as "cathedral spires."

Walking through a redwood grove inspires awe. As one stands within the tall columnar trees, the canopy vault overhead dappling and chromatically filtering the light, enveloped in the pervasive and meditative quiet, it is not hard to appreciate the sense of being within a cathedral of nature.

The cathedral analogy is understandable but somewhat ironic, given that most of these trees existed long before the first cathedrals built by humans. Cathedrals are one of humans' archetypal and iconic architectures, with a unifying and coherent structure designed and built especially to be habitable by both the people and spirit of their region.

The concept of architecture has been adapted to include computer and software systems. At its mid-twentieth-century beginning, software system architecture meant algorithms and data structures. As our skills evolved and allowed increasing software system size and complexity, gotos came to be considered harmful, and domain-specific systems became mainstream. Just like the coastal redwoods, our systems are becoming rich and interconnected ecosystems, and our understanding of the richness and complexities of these systems and their development continues to evolve.

The *Encyclopedia Britannica* says this about architecture:

> *The characteristics that distinguish a work of architecture from other man-made structures are (1) the suitability of the work to use by human beings in general and the adaptability of it to particular human activities, (2) the stability and permanence of the work's construction, and (3) the communication of experience and ideas through its form.*[1]

The first two definitions adapt perfectly to software architecture. When judged successful, our system architectures are usable and adaptable by humans and offer stability in usage, although the concept of "permanence" is perhaps considered too lightly at times, as Y2K taught us.

Applied to software, the third definition may be the richest. Obviously, our system architectures communicate our experience and ideas, but they also reflect and embed the organizational structures that build them. Conway's Law states:

> *. . . organizations which design systems . . . are constrained to produce designs which are copies of the communication structures of these organizations.*[2]

Ultimately system architecture is a human activity, perhaps the most human activity. Perhaps Agile's biggest contribution is the recognition of the humanness of systems development. Agile organizations connect software and the business, and Agile processes provide communication patterns connecting the system-building teams, and the teams with the business, as well as role definitions. Like the forest architecture of the redwoods, Agile organizations evolve ecosystems in which experience and ideas live and grow and

1. www.britannica.com/EBchecked/topic/32876/architecture
2. Melvin E. Conway, "How Do Committees Invent?" *Datamation* 14, no. 5 (April, 1968): 28–31, www.melconway.com/research/committees.html.

enable shared understanding of the problems we're trying to solve, and thereby provide the foundation to architect, design, and build solutions to the problems we're addressing.

Good architecture and Agile organizations help us build systems that provide fitting, innovative, and exceptional solutions to functional and nonfunctional requirements and a sense of accomplishment and joy to the system's builders, maintainers, and users, and they represent, in the very best sense, the culture that designed, built, and lives in and around the system. They help our evolution beyond the observation that Sam Redwine made in 1988:

> *Software and cathedrals are much the same—first we build them, then we pray.*[3]

—Earl Everett

3. Sam Redwine, *Proceedings of the 4th International Software Process Workshop*, Moretonhampstead, Devon, UK, May 11–13, 1988 (IEEE Computer Society).

This page intentionally left blank

Introduction

The best architectures, requirements, and designs emerge from self-organizing teams.

—From "Principles behind the Agile Manifesto"[1]

Why This Book?

Just as Agile software development methods have become mainstream to solve modern, complex problems, practices of software architecture must change to meet the challenges of the ever-changing technology ecosystem. Good Agile teams have undergone a mind-set shift that enables them to deal with changing requirements and incremental delivery. A similar mind-set shift to manage larger software architecture concerns is needed to keep systems robust. Software architecture is as important as ever. Modern product requirements, such as scaling to Internet usage, extending the enterprise beyond the firewall, the need for massive data management, polyglot applications, and the availability of personal computing devices, continue to challenge organizations. To keep up with this ever-changing landscape, modern practices, processes, and solutions must evolve.

To set the foundation for architectural agility, we explore how architecture is accomplished. In any significant enterprise, and certainly on the Internet, architecture functions as a federated system of infrastructure, applications, and components. Thus, many people contribute to the architectures involved in providing a solution to users. Agile software development teams are an excellent example of where sharing architectural responsibilities is essential. To address multiple people collaborating and producing high-quality, integrated software effectively, we must understand how cross-functional, self-organizing teams are involved with and support effective software architectures.

Teams should have "just enough" initial design to get started. A team should also understand what aspects of the software architecture are not well understood yet and what risk that poses to a solution. In Agile, the sequence of delivery is

1. "Principles behind the Agile Manifesto," www.agilemanifesto.org/principles.html, 2001.

ordered by business priority, one of the main reasons Agile is now mainstream. Elements of the software architecture are built and proven early to support the business priorities, and other parts are delayed until they rise in priority. In this way, software architecture is reviewed, updated, and improved in an evolutionary way. Learning how to cope with this process and produce high-quality, highly valued, and integrated software is the focus of this book. To illustrate, the "Manifesto of Agile Software Development" values describe its biases by contrasting two attributes of software development. Either extreme is bad. For example, "responding to change" is valued over "following a plan." Both are important, but the bias is toward the ability to respond to changes as they come. Another example is valuing "working software over comprehensive documentation." It is OK to document an initial architecture and subsequent changes to a level of detail needed for delivering valuable software. This balance could be affected by operational constraints such as compliance and regulatory concerns.

While "everything" is important to consider in software architecture, the ability of architecture to accommodate change is most important. Architecture must define an appropriately open system considering its continued evolution beyond the first release. If it is closed, every change becomes progressively more expensive. At some point the cost per feature added becomes too expensive, and people start to talk about rewriting or replacing the software. This should rarely happen if the architecture establishes an open system that supports an adequate level of changeability. Agile approaches to software development promote this kind of open architecture, provided the team members are equipped with the knowledge and authority to build quality in throughout development and maintenance of the software.

Evolutionary Design

Rather than supporting the design of significant portions of the software architecture before the software is built, Agile methods identify and support practices, processes, and tools to enable evolutionary design. This is not synonymous with undisciplined or "cowboy" coding of software. Agile methods are highly disciplined. One principle behind the "Manifesto for Agile Software Development" in particular identifies the importance of design:

> *Continuous attention to technical excellence and good design enhances agility.*[2]

2. Ibid.

Because design is continuously discussed while implementing features, there is less focus on documentation and handoffs to capture design. People who have traditionally provided designs to project teams are expected to work more closely with the teams. The best way to do this is to be part of the team. When documentation is necessary or supports the continued maintenance of the software, it is created alongside the implementation of the features that made the need visible. Designers may also take on other responsibilities within the team when necessary to deliver working software.

Agile teams are asked to think more broadly than in terms of a single component or application when planning, implementing, and testing features. It is important that they include any integration with external applications in their incremental designs. The team is also asked to continually incorporate enhancements to quality attributes of the software, such as

- **Suitability:** Functionality is suitable to all end users.
- **Interoperability:** Functionality interoperates with other software easily.
- **Compliance:** Functionality is compliant with applicable regulatory guidelines.
- **Security:** The application is secure: confidentiality, integrity, availability, accountability, and assurance.
- **Maturity:** Software components are proven to be stable by others.
- **Fault tolerance:** The software continues operating properly in the event of failure by one or more of its components.
- **Recoverability:** The software recovers from failures in the surrounding environment.
- **Understandability:** People are able to use the software with little training.
- **Learnability:** Functionality is learned with little external interfacing.
- **Operability:** The software is kept in a functioning and operating condition.
- **Performance:** Perceived response is immediate.
- **Scalability:** The software is able to handle increased usage with the appropriate amount of resources.
- **Analyzability:** It is easy to figure out how the software functions.
- **Changeability:** Software components can be changed to meet new business needs.
- **Testability:** Repeatable and specific tests of the software can be created, and there is potential for some to be automated.
- **Adaptability:** Software component functionality can be changed quickly.

- **Installability:** Installation and reinstallation are easy.
- **Conformance:** The software conforms to industry and operational standards.
- **Replaceability:** The software is replaceable in the future.

Taking into consideration external integrations, software quality attributes, and the internal design of components and their interactions is a lot of work. Agile teams look for clarity about what aspects of these areas they should focus more of their effort on. For external integrations, find out who in the organization can support your application integrations and coordinate efforts between teams.

In the case of software quality attributes, work with your business owner to decide which quality attributes are most important for your application. As for the software's internal design, decide how large design changes will be undertaken. Also, figure out how these design decisions will be communicated inside and, if needed, outside the team to external dependents. In all cases, an Agile team looks for ways to consolidate its efforts into practical focus areas that are manageable from iteration to iteration as the application and its design evolve.

In a phase-gate approach, all of the design effort that occurs before construction begins is sometimes referred to as "big design up front" (BDUF). The reason for specifying business requirements and technical design before construction is to reduce risk. I often hear the phrase "We have to get it right" from teams using this phased approach. The BDUF approach to software development, however, creates problems:

- Customers don't know all of the requirements up front, and therefore requirements emerge during implementation. When customers touch and feel the software after implementation, they have feedback for the development team. This feedback is essential to developing software that meets the actual needs of the customer and can be in conflict with the original requirements.
- The people who create business requirements and design specifications are not easily accessible once construction of the software begins. They are often busy specifying and designing other software at that time.
- Development teams, who read requirements and design specifications well after they were created, often interpret business requirements incorrectly. It is common for testers and programmers to have conflicting understandings of requirement details as they interact with existing components and application logic.

- Business needs change frequently, and therefore the requirement details specified weeks or months ago are not necessarily valuable today. Any changes to the requirements must be reflected in the technical design specifications so that the "correct" solution is developed. An adversarial relationship develops between business and technology groups because of these changes. Scope must be managed, or fixed so that the business is not able to make any more changes. Any modifications that the business wants must go through a costly change control process to detail the changes and estimate the impact on the current design, construction, and testing efforts.

Generally, these problems with BDUF are symptoms of feedback cycles that are too long in duration. The time needed to analyze, specify, and design software before constructing it allows requirements and designs to grow stale before they are implemented. One important aspect of an Agile approach is shortening the feedback cycle between customers, the development team, and working software that can be validated. Agile teams manage their development efforts to get working software into the hands of their customers so they can touch it, feel it, and provide feedback. Short iterations and feedback from customers increase the possibility that the software will align with customer desires and expectations as development progresses. This shorter feedback cycle is established using *self-organizing*, *cross-functional*, and *highly collaborative project teams* delivering working software to their customers incrementally using *evolutionary design*.

Self-organizing, Cross-functional Teams

In the seminal paper that profoundly influenced the development of Scrum, "The New New Product Development Game,"[3] Takeuchi and Nonaka provided three characteristics exhibited by self-organizing project teams, which I summarize here:

- **Autonomy:** External involvement is limited to guidance, money, and moral support, and top management rarely intervenes in the team's work. The team is able to set its own direction on day-to-day activities.
- **Self-transcendence:** Teams seem to be continually striving for perfection. Teams set their own goals that align with top management objectives and devise ways to change the status quo.

3. Hirotaka Takeuchi and Ikujiro Nonaka, "The New New Product Development Game," *Harvard Business Review*, January–February 1986.

- **Cross-fertilization:** The team members' different functional specializations, thought processes, and behavior patterns enhance product development once team members start interacting effectively.

In Scrum, the entire team is a self-contained unit, including the Product Owner and the ScrumMaster. The Scrum team members are expected to make incremental improvements to transcend their existing software delivery process capabilities, resulting in better quality and faster throughput of feature delivery over time. Multiple functional roles are represented on a Scrum team. Their cross-fertilizing tendencies and knowledge sharing about different aspects of the delivery process each iteration enable them to figure out how to optimize their interactions over time.

Teams (as opposed to teamwork) self-organize in response to significant challenges—audacious goals—because it energizes them. Leaders help by building a strong performance ethic. Individual responsibility and individual differences become sources of collective strength rather than barriers to team self-organization.

Software development involves the work of multiple functional disciplines: design, testing, programming, analysis, user experience, database, and more, depending upon the project. Agile team members are able to carry out all the work to deliver what is necessary for the project. Instead of optimizing functional disciplines, the team looks for ways to optimize the delivery of a feature from user experience to testing to code to database.

Continuous interaction among team members taking on different functional roles makes good things happen. Team members find ways to interact better, so they are neither overloaded nor starving for work items to take on. When someone on the team is overwhelmed with work items, another team member can look for ways to help that person finish the work. This person could have additional work cycles, depending on the type of work the team took on, and will learn how to execute the easier aspects of a functional discipline with which they help.

Agile teams look for ways to implement features that are verified and validated every iteration. This entails a high degree of collaboration among people across functional roles during the iteration. When the appropriate functional roles for the project are not represented on the team, or are limited, software delivery slows down. Team members have to either cover the work conducted in this functional area or let the work pile up to be done

later. When work is left for later, it becomes more complicated, overwhelming, and error-prone.

Organizations taking an Agile approach must find ways for teams to increase collaboration across functional disciplines. Let's take the situation where a project has testers and programmers, but now they are on the same team. When both functional roles are represented on the team and are working together, a defect can be found much closer to the time that it was injected into the software. This reduces the amount of code created around the defect, puts the defect more into the context of current development efforts, and reduces the risk of unpredictability inherent in finding most defects at the end of a release. Highly collaborative teams enhance delivery, shorten feedback cycles, and improve quality.

Architectures should evolve toward simplicity. This is also true in a scaled environment that has many cross-functional teams. Simplicity emerges when teams spend time finding ways to deliver a complete feature, including the user interface and supporting infrastructure. If the architecture is too complicated, Agile teams make many small changes that lead to removal of unnecessary architectural components over time. This sort of simplification cannot be done in a vacuum by a single team because oversimplification can reduce business options too early and reduce the organization's ability to leverage existing assets for lower-cost solutions. Therefore, teams must work cross-organizationally to make architecture decisions that encompass diverse needs for similar assets. Also, there is a need in larger organizations to facilitate these cross-organizational interactions and communications to a larger audience.

WHY IS THIS TOPIC IMPORTANT?

Most books on Agile software development focus on either practices of the software development process, such as testing and programming techniques, or methods for project management. This book discusses how Agile software organizations can use these practices and methods with a holistic view of software development from team configurations to deployment and maintenance. Some might discuss parts of this book in terms of software or enterprise architecture, but this book is about how teams can take more responsibility for these aspects, taking software from vision to delivery and beyond. In this way, businesses can better understand the path to delivering valuable tools to users, and teams can build more integrity into what they deliver.

This Book's Target Audience

This book is for everyone who is involved in delivering and maintaining software for users. Senior software leadership can find better ways to support and manage delivery of value to stakeholders. Software management can find ways to organize and support the work of development teams. Teams can find out more about how they can build integrity into the full software development life cycle. And team members can take away specific techniques, heuristics, and ideas that can help them improve their own capabilities.

How This Book Is Organized

This book is made up of 11 chapters and an appendix.

Chapter 1, "Managing Software Debt," is a primer on the types of software debt that can impede software changes with age. The topic of software debt is prevalent throughout the book as the main focus for attaining more architectural agility. Five areas of software debt are described in the chapter: technical, quality, configuration management, design, and platform experience. These five areas of software debt are detailed further in the rest of the book.

Chapter 2, "Technical Debt," Chapter 3, "Sustaining Internal Quality," and Chapter 4, "Executable Design," focus on how the internals of software, mostly the code, can be delivered in a way that reduces the friction of future changes.

Chapter 5, "Quality Debt," discusses the problem inherent in the break/fix mentality common in software development organizations. This chapter discusses the use of automation and putting tests toward the front of software delivery to progress toward zero bug tolerance.

Chapter 6, "Configuration Management Debt," presents the need for teams to take care of their software's configuration management needs. The point of this chapter is that teams should have two scripts that do the heavy lifting: deploy and roll back. To do this, many aspects of configuration management must be attended to along the way.

Chapter 7, "Design Debt," Chapter 8, "Designing Software," Chapter 9, "Communicating Architectures," and Chapter 10, "Technology Evaluation Styles," focus on how software is designed for changeability, including its structure, alignment to current business needs, integrity, and design communication.

Chapter 11, "Platform Experience Debt," looks at how people fit into software development and provides team configuration patterns and knowledge-sharing approaches to enable teams to be more effective.

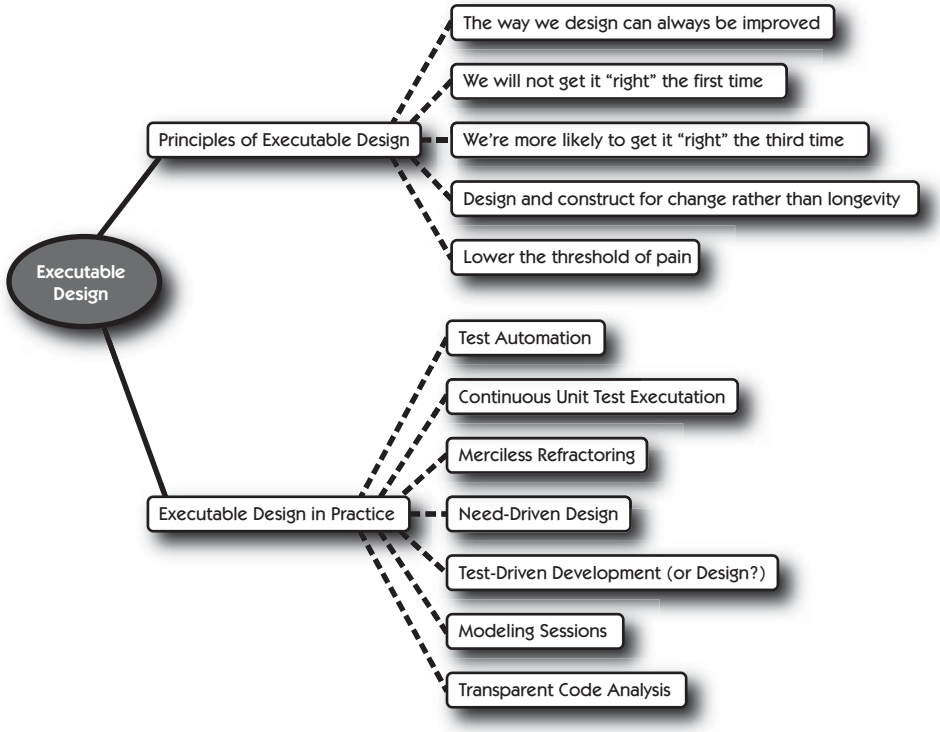
This book is heavily focused on Agile software development, and therefore the expectation is that the reader has experience using an Agile approach such as Scrum or Extreme Programming (XP). For those who want a primer, Appendix A discusses the history of Agile software development along with Scrum and XP in more detail.

I hope that you enjoy this book and that it gives you tools, techniques, and heuristics to improve your daily work life in software development.

This page intentionally left blank

Chapter 4

Executable Design



If we are not enhancing the design then we are just writing a bunch of tests.

—An anonymous developer in a meeting about a Test-First development implementation

Principles of Executable Design

Executable Design is an approach involving existing well-known practices, helpful principles, and a mind-set shift from traditional views on software

design. There have been many projects, conversations, and mistakes involved in defining Executable Design as described in this chapter. There is always room for improvement and perspective. Please take what you can from this chapter and apply it to your own context. In fact, this is an essential principle of Executable Design:

The way we design can always be improved.

This particular principle is not all that controversial. There is a continuous flow of writing about design methods and ideas in our industry. It does, however, suggest the notion that following a single design method is not recommended. By trying multiple methods of design, teams continue to learn and innovate for the sake of their applications.

People in our industry strive for the "best" design methods for software. This has led to many innovations that are in common use today, such as UML and Inversion of Control. In the development of these design methods many ideas were considered and changes were made along the way. In application development, teams also consider ideas and make changes based on their current understanding of the application's design. It is easy sometimes to choose an architectural design style such as Model-View-Controller (MVC), peer-to-peer, and service-oriented. But when these architectural design styles are put into practice to implement a solution of any size, many decisions must be made about specifics in the design. This has led to the following principle of Executable Design:

We will not get it "right" the first time.

This has been shown to be true in my career in the following situations:

- Abstracting web design from application behavior
- Creating a text formatting library for portable devices
- Using model-driven development on a project

Although we will not usually get the design "right" on the first attempt, the design does tend to settle out for most applications. It has come to my attention over the years that

We're more likely to get it "right" the third time.

I am positive that it is not always the third time, but that is not the point of this statement at all. The point is for team members to construct software so

that changes can be incorporated at any point in time. If we accept that we're more likely to get the design "right" closer to the third attempt, we will build software to support change. This gets us to our next principle of Executable Design:

Design and construct for change rather than longevity.

If software is designed and constructed for change, it will be technically and economically feasible to change the software for new needs rather than rewriting it. Designing and constructing with such discipline that changes are easily supported at any time is quite difficult. It takes tremendous discipline when patterns of technical debt, such as schedule pressure as discussed in Chapter 2, are introduced to a team. It is my opinion that we cannot rely on disciplined design over a long period of time by every team member. Therefore, our last principle of Executable Design is

Lower the threshold of pain.

At a workshop in Grand Rapids, Michigan, on technical debt, Matt Heusser proposed that technical debt could be an outcome of individual developers not having to deal with the consequences of their actions. The decisions that we make each day in developing software lead to technical debt because of a "moral hazard."

***Moral hazard* is the view that parties insulated from risk may behave differently from the way they would behave if they were fully exposed to the risk.**

This does not mean that team members act in a malicious or dishonest way. It means that if individual team members are insulated from the long-term effects of a decision, they will not take as much care in the short term. Matt's example was that a person may take a shortcut in developing a feature because that person is not going to be working on the code one year from now when it must be changed to support a new feature.

Immediately following Matt's discussion on moral hazard, Chet Hendrickson pointed out that a good way to minimize the moral hazard problem is by "lowering the threshold of pain." For instance, Chet brought up how many people approach doing their taxes in the United States. They could incrementally update their tax information for two hours each month. Instead, many of us wait until two weeks prior to the deadline to complete our tax forms. We push to complete our taxes by the deadline because the potential headache of tax evasion is strong enough that a pain threshold would be crossed.

Teams can agree to a threshold of pain they are willing to tolerate and put in feedback mechanisms to let them know when that threshold is crossed. In XP, there are multiple frequencies of feedback provided. *Pair programming* enables team members to provide feedback within seconds. *Test-Driven Development (TDD)* and *acceptance testing* provide feedback within minutes of the changes. By using *continuous integration* teams are provided feedback within tens of minutes on how all of their code works together. Having an *on-site customer representative* in close proximity can support getting feedback on an implementation detail within hours of starting work on it. Teams working in time-boxed iterations get feedback from stakeholders within weeks. Getting feedback as close to when an action has been taken is critical to the evolutionary nature of software development using XP.

Identifying a threshold for providing feedback to the team is also a critical aspect of Executable Design. Automating the feedback, when possible, enforces the team's threshold. The feedback could be automated in each team member's development environment, the continuous integration server, and promotion of software to servers exposed to stakeholders.

On some legacy development platforms there could be costs that make frequent feedback difficult or even seemingly impractical. Teams should work toward the shortest frequency of feedback at all levels of the software development process that is practical and feasible in their context. The frequency of feedback that a team can attain is probably more than initially thought.

To recap the principles that drive Executable Design:

- The way we design can always be improved.
- We'll get it "right" the third time.
- We will not get it "right" the first time.
- Design and construct for change rather than longevity.
- Lower the threshold of pain.

Taking on the Executable Design mind-set is not easy. I continually use these principles in design discussions for products, technology, code, tests, business models, management, and life. It has been helpful for me to reflect on situations where these principles have caused me to change my position or perspective. From these reflections I have found more success in setting proper expectations, learning from others in the design process, and designing better solutions for the situation.

Teams can tailor their practices and tools and still be in alignment with the Executable Design principles. The rest of this chapter will provide a set of

suggestions about practices and tools to support an Executable Design approach. By no means are these suggestions the only ways to apply the principles or the only ways that I have seen them applied. These suggestions are only examples to help clarify their application to the software development process. If they work for your current context, your team has a place to start. But please do not stop once you apply them successfully. Remember the first principle:

> **The way we design can always be improved.**

EXECUTABLE DESIGN IN PRACTICE

Executable Design involves the following practices:

- Test automation
- Continuous unit test execution
- Merciless refactoring
- Need-driven design
- Test-Driven Development (or Design?)
- Modeling sessions
- Transparent code analysis

The rest of this chapter will provide detailed information about all of these practices in terms of Executable Design.

Test Automation

This practice may seem implied by Executable Design, but the approach used for test automation is important to sustainable delivery. Also, teams and organizations sometimes think that automating tests is a job for the test group and not for programmers. The approach to test automation in Executable Design is based on the approach to testing in XP. Taking a *whole team* approach to testing is essential to having a successful and sustainable test automation strategy. This does not mean that all team members are the best test case developers and therefore are generalists. It does mean that all team members are able to understand the test strategy, execute the tests, and contribute to their development when needed, which is quite often. The following principle for automated test accessibility sums up the Executable Design suggested approach:

> **Everyone on the team should be able to execute any and all automated and manual test cases.**

This is an extremely important statement because it expresses the importance of feedback over isolation in teams. If any team member can run the tests, the team member can ensure the integrity of changes closer to the time of implementation. This lessens the amount of time between introducing a defect and when it gets fixed. Defects will exist in the software for a shorter duration on average, thus reducing the defect deficit inherent in traditional test-after approaches.

The focus on how automated tests are used is also important. Automated tests at all levels of execution, such as unit, acceptance, system, and performance, should provide feedback on whether the software meets the needs of users. The focus is not on whether there is coverage, although this may be an outcome of automation, but to ensure that functionality is behaving as expected. This focus is similar to that of Behaviour-Driven Development (BDD), where tests validate that each application change adds value through an expected behavior. An approach to automating tests for Executable Design could be the use of BDD.

In addition to the automating test development approach, understanding how the test infrastructure scales to larger projects is essential for many projects. Structure and feedback cycles for each higher layer of test infrastructure can make or break the effective use of automated tests for frequent feedback. Over time, the number of tests will increase dramatically. This can cause teams to slow down delivery of valuable features if the tests are not continually maintained.

The most frequent reason for this slowdown is that unit tests are intermingled with slower integration test executions. Unit tests should run fast and should not depend on special configurations, installations, or slow-running dependencies. When unit tests are executed alongside integration tests, they run much slower and cause their feedback to be available less frequently. This usually starts with team members no longer running the unit tests in their own environment before integrating a change into source control.

A way to segregate unit tests from integration tests is to create an automated test structure. In 2003, while working on an IBM WebSphere J2EE application with a DB2 on OS/390 database, our team came up with the following naming convention to structure our automated tests:

- *UnitTest.java: These tests executed fast and did not have dependencies on a relational database, JNDI (Java Naming and Directory Interface), EJB, IBM WebSphere container configuration, or any other

external connectivity or configurations. In order to support this ideal unit test definition, we needed to isolate business logic from "glue" code that enabled its execution inside the J2EE container.

- *PersistanceTest.java: These tests depended on a running and configured relational database instance to test integration of EJB entity beans and the database. Because our new architecture would be replacing stored procedure calls with an in-memory data cache, we would need these integration test cases for functional, load, performance, and stress testing.
- *ContainerTest.java: These tests were dependent on integrating business logic into a configured IBM WebSphere J2EE container. The tests ran inside the container using a framework called JUnitEE (extension of JUnit for J2EE applications) and would test the container mappings for application controller access to EJB session beans and JNDI.

In our development environments we could run all of the tests whose names ended with "UnitTest.java". Team members would execute these tests each time they saved their code in the IDE. These tests had to run fast or we would be distracted from our work. We kept the full unit test execution time within three to five seconds. The persistence and container tests were executed in a team member's environment before larger code changes—meaning more than a couple of hours of work—were checked in.

The full suite of automated programmer tests was executed on our continuous integration server each time code was checked into our source control management system. These took anywhere from 5 to 12 minutes to run. The build server was configured with a WebSphere Application Server instance and DB2 relational database. After the build and automated unit tests ran successfully, the application was automatically deployed into the container, and the database was dropped and re-created from scratch. Then the automated persistence and container tests were executed. The results of the full build and test execution were reported to the team.

Continuous Unit Test Execution

Automated programmer tests aren't as effective if they are not executed on a regular basis. If there is an extra step or more just to execute programmer tests in your development environment, you will be less likely to run them. Continuous programmer test execution is focused on running fast unit tests for the entire module with each change made in a team member's development environment without adding a step to the development process. Automating unit test execution each time a file is modified in a background process will help team members identify issues quickly before more software

debt is created. This goes beyond the execution of a single unit test that tests behavior of the code under development. Team members are continually regressing the entire module at the unit level.

Many platforms can be configured to support continuous unit test execution:

- In Eclipse IDE, a "launcher," similar to a script that can be executed, can be created that runs all of the unit tests for the module. A "launcher" configuration can be saved and added to source control for sharing with the entire team. Another construct in Eclipse IDE called "builders" can then be configured to execute the "launcher" each time a file is saved.
- If you are into programming with Ruby, a gem is available called ZenTest with a component named *autotest* for continuous unit test execution. It also continuously executes unit tests when a file is changed. Autotest is smart about which tests to execute based on changes that were made since the last save.
- Python also has a continuous unit test execution tool named *tdaemon* that provides similar functionality to ZenTest for Ruby.

As you can see, continuous testing works in multiple programming languages. Automating unit test execution with each change lessens the need for adding a manual step to a team member's development process. It is now just part of the environment. Teams should look for ways to make essential elements of their software development process easy and automatic so it does not become or appear to be a burden for team members.

Merciless Refactoring

Refactoring is an essential practice for teams developing solid software and continually evolving the design to meet new customer needs. The web site managed by Martin Fowler, who wrote the original book conveniently called *Refactoring*, says:

> *Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior.*[1]

It is important to understand that refactoring is not just restructuring code. Refactoring involves taking small, disciplined steps, many of which are docu-

1. Martin Fowler, "Refactoring Home Page," www.refactoring.com/.

mented in books and online resources on refactoring, to alter the internal structure of the software. (If you haven't done so already, please read the books and online resources on refactoring to learn how it is applied more effectively. This book will not describe specific refactorings in detail.)

When taking an iterative and incremental approach such as Scrum, it is imperative that the software continue to be changeable. Merciless refactoring is an approach that teams should adopt whether they are working on new or legacy software.

> ***merciless*—adj.: having or showing no [mercy—show of kindness toward the distressed]**

To refactor *mercilessly* means that the team will

> **Relieve distressed code through kindness and disciplined restructuring.**

Some teams wonder if they will be allowed to apply merciless refactoring in their project. It is important to understand that teams are not asked to develop software that does not allow for new changes to be easily added. Stakeholders do tend to want features quickly, but that is their role. Teams should understand that their role is to create quality software that does not accrue abnormal costs with each change. Robert C. Martin wrote in his book *Clean Code: A Handbook of Agile Software Craftsmanship* about a simple rule that the Boy Scouts of America have:

> *Leave the campground cleaner than you found it.*[2]

Teams that I work with use a variant of this simple rule in their own working agreements:

> **Always leave the code in better shape than when you started.**

Teams demonstrating this mind-set will continually improve the software's design. This leads to acceleration in feature delivery because the code will be easier to work with and express its intent more concisely. On a project that is well tended in terms of its design and structure, the act of refactoring can be elegant and liberating. It allows teams to continually inspect and adapt their understanding of the code to meet the customer's current needs.

2. Robert C. Martin, *Clean Code: A Handbook for Agile Software Craftsmanship* (Prentice Hall, 2009); www.informit.com/articles/article.aspx?p=1235624&seqNum=6.

On a legacy application or component, the act of refactoring can seem overwhelming. Although refactoring involves making small, incremental improvements that will lead to improvement in the software's design, figuring out where to start and stop in a legacy system is often unclear. How much refactoring is sufficient in this piece of code? The following questions should help you decide whether to start refactoring when you see an opportunity for it:

1. Does this change directly affect the feature I am working on?
2. Would the change add clarity for the feature implementation?
3. Will the change provide automated tests where there currently are none?
4. Does the refactoring look like a large endeavor involving significant portions of the application components?

If the answer to the first three questions is yes, I lean toward refactoring the code. The only caveat to this answer is when the answer to the fourth question, Does the refactoring look like a large endeavor?, is yes. Then I use experience as a guide to help me produce a relative size estimate of the effort involved in this refactoring compared to the initial estimate of size for the feature implementation. If the size of the refactoring is significantly larger than the original estimate given to the Product Owner, I will bring the refactoring up to the team for discussion. Bringing up a large refactoring to the rest of the team will result in one of the following general outcomes:

- The team thinks it is good idea to start the large refactoring because its estimated size does not adversely affect delivery of what the team committed to during this iteration.
- The team decides that the refactoring is large enough that it should be brought up to the Product Owner. The Product Owner could add it to the Product Backlog or decide to drop scope for the current iteration to accommodate the refactoring.
- Another team member has information that will make this refactoring smaller or not necessary. Sometimes other team members have worked in this area of code or on a similar situation in the past and have knowledge of other ways to implement the changes needed.

After starting a refactoring, how do we know when to stop? When working on legacy code, it is difficult to know when we have refactored enough. Here are some questions to ask yourself to figure out when you have refactored enough:

- Is the code I am refactoring a crucial part of the feature I was working on?
- Will refactoring the code result in crucial improvements?

Adopting a *merciless refactoring* mind-set will lead to small, incremental software design improvements. Refactoring should be identified in the course of implementing a feature. Once the need for a refactoring is identified, decide if it is valuable enough to do at this point in time, considering its potential cost in effort. If it meets the criteria for starting a refactoring, use disciplined refactoring steps to make incremental improvements to the design without affecting the software's external behavior.

Need-Driven Design

A common approach to designing application integrations is to first identify what the provider will present through its interface. If the application integration provider already exists, consumers tend to focus on how they can use all that the provider presents. This happens when integrating services, libraries, storage, appliances, containers, and more.

In contrast, *Need-Driven Design* approaches integration based on emergence and need. The approach can be summarized in the following statement:

> **Ask not what the integration provider gives us; ask what the consumer needs.**

Need-Driven Design, in its basic form, is based on the *Adapter* design pattern[3] as shown in Figure 4.1. There are two perspectives for integration in Need-Driven Design:

- **Consumer:** Ask what the consumer needs from the interface contract.
- **Provider:** A provider's interface emerges from needs expressed by more than one consumer.

From the consumer perspective, the idea is to depend only on what the software actually needs and no more. Instead of coupling the application to a web service directly, create an interface in between that defines only what the software needs, then implement the interface to integrate with the web service. This way, dependency on the web service is limited to the implementation of the interface and can be modified or replaced if the need arises in a single place. In the case of integrating a library, creating too much dependence on the library could make the application less changeable for new user needs. Wrapping the specific aspects of a library that the application uses could be an approach worth pursuing.

3. Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley, 1994).

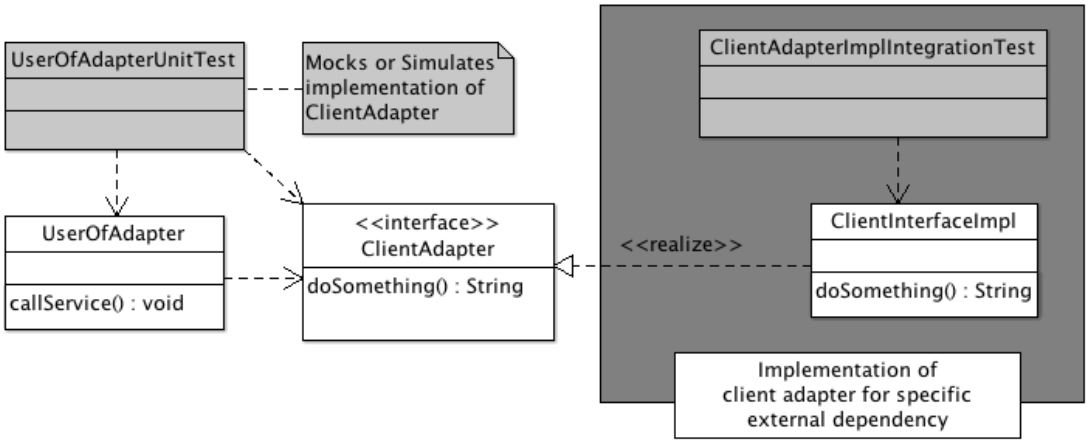


Figure 4.1 Example implementation of the Need-Driven Design approach for exploiting an external component using the Adapter design pattern

From a provider perspective, generalizing an interface should be done only after there is more than one consumer of the provider's capabilities. This contrasts with how many organizations approach application integration. They might have a governance program that identifies services before construction of software begins. The Need-Driven Design approach is to wait for more than one consumer to need access to a particular capability. Once the need to create a provider interface is identified, it is promoted to a reusable asset.

The Need-Driven Design approach is best applied in conjunction with automated unit tests. The automated unit tests describe what each side of the adapter, the consumer interface and provider interface, will be responsible for. The application using the adapter should handle general usage of the client interface and should not care about specialized concerns of the provider interface. This can be explained with the following real-world example where Need-Driven Design was applied.

Instead of designing software toward what an external dependency can provide, decide what the application needs. This need-driven approach focuses on adding only what is necessary rather than creating dependence on the external component. Need-Driven Design has the following steps:

1. **Assess the need:** Add external dependencies to your project only when the value outweighs the integration costs.
2. **Define the interface:** Create an interface that will provide your application with the capabilities that it needs.

Container Trucks and RFID

The application we were working on tracked containers on trucks being loaded on and off of ships in the port. Radio-frequency identification device (RFID) was becoming the way containers were tracked coming in and out of port. Ports in the United States were finding ways to support reading these RFID tags on the containers.

Our team was employed to implement the software that would take an RFID tag identified on a container and relay that information to port workers. A separate vendor was hired to create the RFID-reading hardware since none existed to our knowledge that could handle the corrosive nature of being near salt water.

Since the hardware did not exist, we asked the other vendor for an example of what the message would look like so that we could implement a parser for it. When we got the example message, we found that the XML it contained was malformed. This was mentioned to the hardware vendor, but we still had to make progress because of our contractual obligations. We were all supposed to be finished with our application changes, integrated with the RFID hardware, in three months.

Our team decided to figure out what the application needed rather than what it would receive from the other vendor's hardware. In one of the integration scenarios, the application needed an RFID tag and a timestamp, so we created an interface to access these pieces of information. To ensure that the application was able to handle implementations of this interface, we wrote automated unit tests that used mock objects to run scenarios through the interface. For instance, what if the RFID could not be read?

After understanding how the application would use the interface, we created an initial implementation of the interface for the vendor's hardware based on what we thought would be close to the actual message. Since the hardware was providing the XML through a web service, the interface implementation caught any web-service-specific errors and exceptions so that the application did not have to be coupled to even the technical aspects of it. Automated unit tests validated that the interface implementation handled such conditions.

It turned out that the hardware vendor was more than three months late. Our team created a simulator that would send multiple versions of an example XML message with random RFID tags and random exception conditions. One item was still needed for release when we finished our work: integration with the actual hardware through the interface implementation.

As an epilogue, our clients sent us a note on the day they got the hardware. They modified the interface implementation to accommodate the actual XML message from the working hardware in only one hour and ran all of the automated tests. It worked! They put it into production that same day.

3. **Develop executable criteria:** Write automated unit tests for expected scenarios your application should handle through the defined interface; mock up and/or simulate the various scenarios.
4. **Develop the interface implementation:** Create automated unit tests and the interface implementation to integrate external components, and make sure to handle conditions that the application does not need to be aware of.

By driving integration strategies through the steps defined in Need-Driven Design, we can decrease integration costs, reduce coupling to external dependencies, and implement business-driven intentions in our applications.

Test-Driven Development (or Design?)

Test-Driven Development (TDD) is a disciplined practice in which a team member writes a failing test, writes the code that makes the test pass, and then refactors the code to an acceptable design. Effective use of TDD has been shown to reduce defects and increase confidence in the quality of code. This increase in confidence enables teams to make necessary changes faster, thus accelerating feature implementation throughput.

It is unfortunate that TDD has not been adopted by the software development industry more broadly. The TDD technique has been widely misunderstood by teams and management. Many programmers hear the name and are instantly turned off because it contains the word *test*. Teams that start using TDD sometimes misinterpret the basics or have difficulty making the mindset shift inherent in its use. Using tests to drive software design in an executable fashion is not easy to grasp. It takes tremendous discipline to make the change in approach to design through micro-sized tests.

The following statement summarizes how I describe TDD to teams that are having difficulty adopting the approach in their development process:

TDD is about creating a supportable structure for imminent change.

Applications and their components change to meet new business needs. These changes are effected by modifying the implementation, improving the design, replacing aspects of the design, or adding more functionality to it. Taking a TDD approach enables teams to create the structure to support the changes that occur as an application changes. Teams using a TDD approach should maintain this structure of unit tests, keeping the unit tests supportable as the application grows in size and complexity. Focusing on TDD in

this manner helps teams understand how they can apply it to start practicing a design-through-tests approach.

Automated unit tests are added through a test-driven approach and tell us if aspects of each component within an application are behaving as expected. These tests should be repeatable and specific so they can be executed with the same expected results each time. Although the tests are important, teams should not lose focus on how these tests drive the software design incrementally.

A basic way to think about TDD is through a popular phrase in the TDD community:

Red, Green, Refactor.

This simple phrase describes the basic steps of TDD. First, write a failing test that describes the scenario that should work at a micro level of the application component. Then write just enough code to make it pass, and no more. Finally, refactor the implementation code and tests to an acceptable design so they can be maintained over time. It is important to emphasize once again to write only enough code to make the current failing test pass so no untested code is written. Untested code is less safe to change when it's time to make necessary refactorings. Figure 4.2 shows the basic steps involved in the TDD approach.

These three basic steps are not always sufficient to do TDD effectively. Uncle Bob Martin wrote "The Three Laws" of TDD as follows:

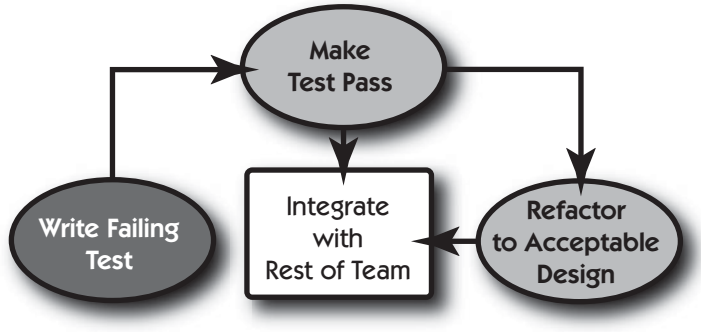


Figure 4.2 The basic steps of Test-Driven Development are to write a failing test, write only the code that makes the test pass, and refactor to an acceptable design.

Test-Driven Development is defined by three simple laws.

- *You must write a failing unit test before you write production code.*
- *You must stop writing that unit test as soon as it fails; and not compiling is failing.*
- *You must stop writing production code as soon as the currently failing test passes.*[4]

He goes on to say that software developers should do TDD as a matter of professionalism. If, as software developers, we do TDD effectively, we will get better at our craft. Uncle Bob Martin provides an initial list of things we could improve by doing TDD:

If you follow the three laws that seem so silly, you will:

- *Reduce your debug time dramatically.*
- *Significantly increase the flexibility of your system, allowing you to keep it clean.*
- *Create a suite of documents that fully describe the low level behavior of the system.*
- *Create a system design that has extremely low coupling.*[5]

Difficulty in Introducing Test-Driven Development

TDD is a highly disciplined approach. The discipline involved is difficult for some people to apply each day. Following is a list of team environmental issues that lower the chances for effective use or adoption of TDD:

- Pressure from management and stakeholders to release based on an unreasonable plan: Integrity of the software is always sacrificed when the plan is inflexible and does not incorporate reality.
- When there is a lack of passion for learning and implementing effective TDD practices on the team: The high degree of discipline required in TDD makes passion for working in this way extremely helpful.
- Not enough people on the team with experience doing TDD in practice: Without any experience on the team, success in adoption is lower.
- On an existing code base, if the software's design is poor, low cohesion, and high coupling, or is implemented in a way that is difficult to test, then finding a starting point could seem nearly impossible.

4. Robert C. Martin, "Are You a Professional?" *NDC Magazine*, Norwegian Developers Conference 2009, Telenor Arena, Oslo, June 17–19, p. 14. Reprinted with permission.
5. Ibid.

- If the existing code base is large and contains no or minimal test coverage, disciplined TDD will not show valuable results for some time.
- Managers tell team members they don't believe TDD is effective or directly prohibit its use on projects.

To successfully adopt a TDD approach, it is important to manage these environmental issues for the team. This could include managing expectations, providing the team with support from a coach, and allowing sufficient learning time to understand the tools and techniques.

Modeling Sessions

When team members get together and discuss software design elements, they sometimes use visual modeling approaches. This usually happens at a whiteboard for collocated team members. While the team names model elements and their interactions, the conversation revolves around how the model enables desired functionality. The points discussed can be thought of as scenarios that the solution should support. These scenarios are validated against the model throughout the design conversation and can be easily translated into one or more test cases.

As the modeling session continues, it becomes more difficult to verify the number of scenarios, or test cases, that have already been discussed. When an interesting scenario emerges in conversation and causes the model to change, the group must verify the model against all the scenarios again. This is a volatile and error-prone approach to modeling because it involves manual verification and memorization. Even so, modeling is a valuable step since it helps team members arrive at a common understanding of a solution for the desired functionality. Minimizing the volatile and error-prone aspects of this technique improves the activity and provides more predictable results. Using TDD to capture the test cases in these scenarios will eventually make them repeatable, specific, and executable. It also helps to ensure that the test cases providing structure to the solution's design are not lost. Without the test cases it is difficult to verify the implementation and demonstrate correct and complete functionality.

By no means would I prescribe that teams eliminate quick modeling sessions. Modeling sessions can provide a holistic view of a feature or module. Modeling only becomes an issue when it lasts too long and delays implementation. The act of designing should not only be theoretical in nature. It is good to timebox modeling sessions. I have found that 30 minutes is sufficient for conducting a modeling session. If a team finds this amount of time insufficient, they should take a slice of a potential solution(s) and attempt to implement it

before discussing the rest of the design. The act of implementing a portion of the design provides a solid foundation for further exploration and modeling.

Modeling Constraints with Unit Tests

To reduce duplication and rigidity of the unit test structure's relationship to implementation code, teams should change the way they define a "unit." Instead of class and method defined as the only types of "unit," use the following question to drive the scenario and test cases:

> **What should the software do next for the intended user?**

The approach for writing unit tests I follow is that of Behaviour-Driven Development (BDD).[6] Thinking in terms of the following BDD template about how to model constraints in unit tests helps me stay closer to creating only the code that supports the desired functionality:

```
Given <some initial context>
When <an event occurs>
Then <ensure some outcomes>.
```

By filling in this template I can generate a list of tests that should be implemented to supply the structure that ensures the desired functionality. The following coding session provides an example of applying this approach. The fictitious application is a micro-blogging tool named "Jitter." The functionality I am working on is this:

> **So that it is easier to keep up with their child's messages, parents want shorthand in the messages to be automatically expanded.**

The acceptance criteria for this functionality are:

- LOL, AFAIK, and TTYL are expanded for a parent.
- It should be able to expand lower- and uppercase versions of the shorthand.

The existing code is written in Java and already includes a `JitterSession` class that users obtain when they authenticate into Jitter. Parents can see their child's messages in their session. The following unit test expects to expand "LOL" to "laughing out loud":

6. An introduction to Behaviour-Driven Development (BDD) can be found at http://blog.dannorth.net/introducing-bdd/, and the Given, When, Then template is shown in the article in a section named "BDD Provides a 'Ubiquitous Language' for Analysis."

```
public class WhenParentsWantToExpandMessagesWithShorthandTest {


    @Test
    public void shouldExpandLOLToLaughingOutLoud() {
        JitterSession session = mock(JitterSession.class);
        when(session.getNextMessage()).thenReturn("Expand LOL");
        MessageExpander expander = new MessageExpander(session);
        assertThat(expander.getNextMessage(),
            equalTo("Expand laughing out loud"));
    }

}
```

Before we continue with the unit test code example, let's look more closely at how it is written. Notice the name of the programmer test class: `WhenParentsWantToExpandMessagesWithShorthandTest`.

For some programmers, this long name might seem foreign. It has been my experience that it is easier to understand what a programmer test has been created for when the name is descriptive. An initial reaction that programmers have to long names for classes and methods is the fear they will have to type them into their editor. There are two reasons why this is not an issue:

- Because this is a unit test, other classes should not be using this class.
- Modern integrated development environments have code expansion built in.

Also notice that the name of the test method is `shouldExpandLOLToLaughingOutLoud`. This naming convention supports how we drive design through our unit tests by answering the question "What should the software do next for the intended user?" By starting the method name with the word *should*, we are focusing on what the software should do for the user identified in the unit test class name. This is not the only way to write unit tests. People have a wide variety of preferences about how to write their tests, so please find the way that fits your team's intended design strategy best.

The `MessageExpander` class does not exist, so I create a skeleton of this class to make the code compile. Once the assertion at the end of the unit test is failing, I make the test pass with the following implementation code inside the `MessageExpander` class:

```
public String getNextMessage() {
    String msg = session.getNextMessage();
    return msg.replaceAll("LOL", "laughing out loud");
}
```

This is the most basic message expansion I could do for only one instance of shorthand text. I notice that there are different variations of the message that I want to handle. What if LOL is written in lowercase? What if it is written as "Lol"? Should it be expanded? Also, what if some variation of LOL is inside a word? The shorthand probably should not be expanded in that case except if the characters surrounding it are symbols, not letters. I write all of this down in the unit test class as comments so I don't forget about it:

```
// shouldExpandLOLIfLowerCase
// shouldNotExpandLOLIfMixedCase
// shouldNotExpandLOLIfInsideWord
// shouldExpandIfSurroundingCharactersAreNotLetters
```

I then start working through this list of test cases to enhance the message expansion capabilities in Jitter:

```
@Test
public void shouldExpandLOLIfLowerCase() {
    when(session.getNextMessage()).thenReturn("Expand lol please");
    MessageExpander expander = new MessageExpander(session);
    assertThat(expander.getNextMessage(),
        equalTo("Expand laughing out loud please"));
}
```

At this point, I find the need for a minor design change. The `java.lang.String` class does not have a method to match case insensitivity. The unit test forces me to find an alternative, and I decide to use the `java.util.regex.Pattern` class:

```
public String getNextMessage() {
    String msg = session.getNextMessage();
    Pattern p = Pattern.compile("LOL", Pattern.CASE_INSENSITIVE);
    Return p.matcher(msg).replaceAll("laughing out loud");
}
```

Now I make it so that mixed-case versions of "LOL" are not expanded:

```
@Test
public void shouldNotExpandLOLIfMixedCase() {
    String msg = "Do not expand Lol please";
    when(session.getNextMessage()).thenReturn(msg);
    MessageExpander expander = new MessageExpander(session);
    assertThat(expander.getNextMessage(), equalTo(msg));
}
```

This forces me to use the `Pattern.CASE_INSENSITIVE` flag in the pattern compilation. To ensure that only the code necessary to make the test pass is created, I match only "LOL" or "lol" for replacement:

```
public String getNextMessage() {
    String msg = session.getNextMessage();
    Pattern p = Pattern.compile("LOL|lol");
    return p.matcher(msg).replaceAll("laughing out loud");
}
```

Next, I make sure that if "LOL" is inside a word it is not expanded:

```
@Test
public void shouldNotExpandLOLIfInsideWord() {
    String msg = "Do not expand PLOL or LOLP or PLOLP please";
    when(session.getNextMessage()).thenReturn(msg);
    MessageExpander expander = new MessageExpander(session);
    assertThat(expander.getNextMessage(), equalTo(msg));
}
```

The pattern matching is now modified to use spaces around each variation of valid "LOL" shorthand:

```
return Pattern.compile("\\sLOL\\s|\\slol\\s").matcher(msg)
    .replaceAll("laughing out loud");
```

Finally, it is important that if the characters around LOL are not letters, such as a space, it still expands:

```
@Test
public void shouldExpandIfSurroundingCharactersAreNotLetters() {
    when(session.getNextMessage()).thenReturn("Expand .lol!
please");
    MessageExpander expander = new MessageExpander(session);
    assertThat(expander.getNextMessage(),
        equalTo("Expand .laughing out loud! please"));
}
```

The final implementation of the pattern-matching code looks like this:

```
return Pattern.compile("\\bLOL\\b|\\blol\\b").matcher(msg)
    .replaceAll("laughing out loud");
```

I will not continue with more of the implementation that would expand other shorthand instances. However, I do want to discuss how the focus on "What should the software do next?" drove the design of this functionality. Driving the code using TDD guides us to implement only what is needed. It also helps us approach 100% code coverage for all lines of code. For programmers who have experience writing object-oriented code, the modules will likely have high cohesion, focused on specific responsibilities, and maintain low coupling to other code. The failing unit test represents something

that the software does not do yet. We focus on modifying the software with the simplest implementation we can think of that will make the unit test pass. Then we focus on enhancing the software's design with the refactoring step. It has been my experience that refactoring takes most of the effort when applying TDD effectively. This does not mean refactoring is used with each TDD cycle. It means that overall, programmers spend more time refactoring to enhance the design.

Software Design beyond TDD

Most software design approaches are concerned with documenting design artifacts. Agile teams look for ways to reduce documentation to only what is necessary. Because of this statement, many teams and organizations mistakenly think Agile means no documentation. This is an inappropriate interpretation of Agile software development and is not corroborated by thought leaders and books from the Agile community.

To better enable cost-effective and high levels of support for applications deployed in production, teams ought to be aware of artifacts that assist ongoing maintenance. Software development goes beyond just writing code. It also includes demonstrating the integrity of component integration, alignment to business objectives, and communication of the software's structure for continued maintenance. Some of these aspects can be validated through integration tests. As pointed out in the section on test automation earlier in this chapter, integration tests are not executed as frequently as fast unit tests, such as in each team member's environment. Instead, they are executed in an integration environment when changes are integrated into a common stream of work in source control.

Teams that must consider some or all of the aspects listed above should have processes and tools that support effective maintenance. On top of automated integration testing, they might also benefit from

- Frequent and enhanced compliance auditing
- A team member with specific knowledge or appropriate training and practice
- Push-button deployment and rollback capability to all associated environments
- Production-like staging and test environments for more realistic integration testing

As a team, think about which aspects of software design you should be concerned with and then figure out how you will manage them in the software development process.

Transparent Code Analysis

Code coverage tools measure whether all discernible paths through the code have been tested. It is impossible, except in the most basic instances of code, to validate that all paths through the code have been tested with every potential input. On the other hand, it is possible to ascertain whether each line of code in a module has been tested. This involves measuring test coverage by some basic metrics:

- **Statement coverage** checks how many lines of code have been executed.
- **Decision coverage** checks if each path through a control structure is executed (i.e., "if/else if/else" structures).
- **Condition coverage** checks if each Boolean expression is evaluated to both true and false.
- **Path coverage** checks if combinations of logical code constructs are covered, including sufficient loop evaluation (i.e., executing a loop 0, 1, and more than 1 time).
- **Relational operator coverage** checks if inputs in relational operator evaluation are sufficiently verified (i.e., executing $a < 2$ with $a = 1$, $a = 2$, $a = 3$).

Executing code coverage tools inside each development and continuous integration environment can be helpful feedback to identify lapses in test-driven discipline. As a programmer gains more experience with TDD, it becomes practical to approach 100% coverage in most instances. Tools can provide feedback about code coverage, as shown in Figure 4.3.

It is a common belief that striving for 100% code coverage is too expensive and does not provide enough value to offset its costs. In my experience, approaching 100% code coverage increases confidence and accelerates delivery

Element	Coverage	Covered Instructions	Total Instructions
TisUgly	97.3 %	729	749
src/main/java	97.3 %	729	749
com.gettingagile.tisugly	100.0 %	106	106
DesignAssertion.java	100.0 %	58	58
MessageConsole.java	100.0 %	25	25
Module.java	100.0 %	23	23
com.gettingagile.tisugly.analyzer	99.6 %	284	285
ASMAnalyzer.java	99.6 %	284	285
org.objectweb.asm.depend	94.7 %	339	358
DependencyVisitor.java	94.7 %	339	358

Figure 4.3 The Eclipse IDE plug-in, EclEmma, showing a view of the project's code coverage inside the Eclipse IDE

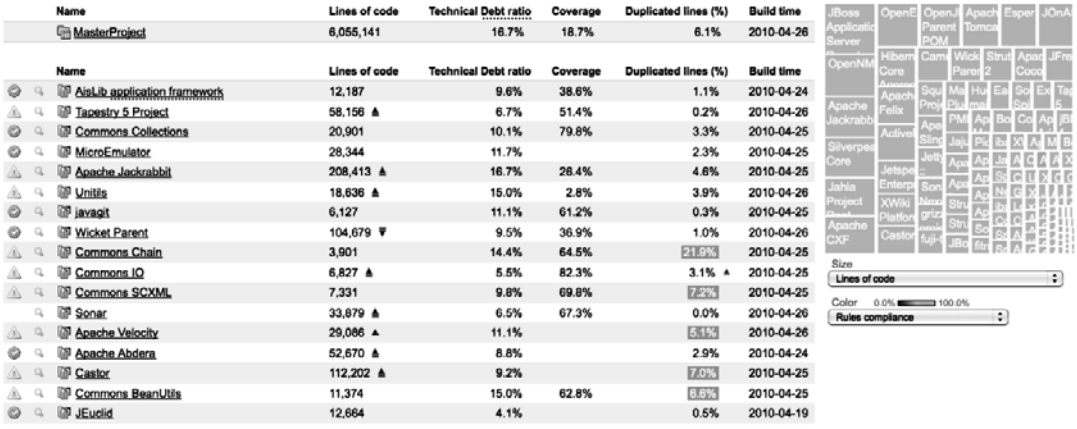
of working software. There are some factors that inhibit teams from approaching 100% code coverage:

- Working with an existing code base that has significantly less than 100% code coverage: If this is the case, track increases in code coverage rather than whether it approaches 100%.
- A brittle component or application where finding a place to put a valid unit test is difficult, or nearly impossible: In this case, using a black-box testing tool that executes the code in its packaged or installable form could be a better option until the code is less tangled. The packaged or installed application could be instrumented sometimes so that code coverage is evaluated during the black-box test execution.
- When code is generated: Teams should look for ways to isolate generated code from code implemented by team members and evaluate code coverage only for the latter. In some circumstances, our team has been able to generate the unit and integration tests for generated code when we had access to the code generation templates.
- When code integrates with third-party components: Although the third-party component probably will not have 100% code coverage, integration tests can be developed that verify how the code is expected to integrate. Evaluate code coverage on only code that your team created. See the Need-Driven Design section earlier in this chapter.

When working with an existing code base, approachiìng 100% is probably not attainable. In this case, make sure that the current code coverage does not deteriorate. Teams should look for ways to slowly increase code coverage of the existing software over time.

Tools to determine code coverage are not the only code analysis tools available. There are many tools in the static code analysis arena as well. Static code analysis tools can provide feedback through a dashboard, such as in Figure 4.4, on aspects of the software such as

- Team's preferred coding rules
- Lines of code
- Cyclomatic complexity
- Duplicated code
- Lack of cohesion
- Maintainability
- Dependencies
- Design
- Architecture
- Technical debt



Name	Lines of code	Technical Debt ratio	Coverage	Duplicated lines (%)	Build time
MasterProject	6,055,141	16.7%	18.7%	6.1%	2010-04-26

Name	Lines of code	Technical Debt ratio	Coverage	Duplicated lines (%)	Build time
AisLib application framework	12,187	9.6%	38.6%	1.1%	2010-04-24
Tapestry 5 Project	58,156	6.7%	51.4%	0.2%	2010-04-26
Commons Collections	20,901	10.1%	79.8%	3.3%	2010-04-25
MicroEmulator	28,344	11.7%		2.3%	2010-04-25
Apache Jackrabbit	208,413	16.7%	28.4%	4.6%	2010-04-25
Unitils	18,636	15.0%	2.8%	3.9%	2010-04-26
javagit	6,127	11.1%	61.2%	0.3%	2010-04-25
Wicket Parent	104,679	9.5%	36.9%	1.0%	2010-04-26
Commons Chain	3,901	14.4%	64.5%	21.9%	2010-04-25
Commons IO	6,827	5.5%	82.3%	3.1%	2010-04-25
Commons SCXML	7,331	9.8%	69.8%	7.2%	2010-04-25
Sonar	33,879	6.5%	67.3%	0.0%	2010-04-26
Apache Velocity	29,086	11.1%		5.1%	2010-04-26
Apache Abdera	52,670	8.8%		2.9%	2010-04-24
Castor	112,202	9.2%		7.0%	2010-04-25
Commons BeanUtils	11,374	15.0%	62.8%	8.6%	2010-04-25
JEuclid	12,664	4.1%		0.5%	2010-04-19

Figure 4.4 The Sonar dashboard showing metrics for lines of code, technical debt ratio, code coverage, duplicated lines of code, and build time[7]

I do not suggest that the metrics generated for all of these aspects of the software are exact. They do provide feedback about our software internals and can help guide our development efforts. Also, it is sometimes easier to drill down into the static code analysis dashboard for an area of code that will be involved in changes for the next feature. Looking at higher-level metrics for the code can provide useful information to guide implementation and opportunities for code improvement.

SUMMARY

Executable Design is a method for driving the implementation of software functionality. It focuses on the following principles:

- The way we design can always be improved.
- We'll get it "right" the third time.
- We will not get it "right" the first time.
- Design and construct for change rather than longevity.
- Lower the threshold of pain.

Going beyond how we write automated tests, Executable Design also involves how they are structured inside projects, how they are executed in different

7. Sonar is an open platform to manage code quality available at www.sonarsource.org/. The picture is from their demo web site at http://nemo.sonarsource.org/.

environments, and a way to think about what the next test should be. When we answer the question "What should the software do next for the intended user?" our programmer tests will be more directly focused on delivery of value to the user. Using transparent code analysis, team members can get feedback on the health of the code as they implement in an incremental fashion.

INDEX

C

Q

S

U