Lean Architecture

for Agile Software Development

by James O. Coplien & Gertrud Bjørnvig

WILEY

Lean Architecture for Agile Software Development

James Coplien
Gertrud Bjørnvig

A John Wiley and Sons, Ltd, Publication

This edition first published 2010

Registered office
John Wiley & Sons Ltd, The Atrium, Southern Gate, Chichester, West Sussex, PO19 8SQ, United Kingdom

For details of our global editorial offices, for customer services and for information about how to apply for permission to reuse the copyright material in this book please see our website at www.wiley.com.

Reprinted with corrections December 2010

A catalogue record for this book is available from the British Library.

ISBN 978-0-470-68420-7

Typeset in 11/13 Palatino by Laserwords Private Limited, Chennai, India.
Printed in Great Britain by TJ International, Padstow, Cornwall

Dedication

To Trygve Mikkjel Heyerdahl Reenskaug, also a grandfather

Publisher's Acknowledgments

Some of the people who helped bring this book to market include the following:

Editorial and Production
VP Consumer and Technology Publishing Director: Michelle Leete
Associate Director – Book Content Management: Martin Tribe
Associate Publisher: Chris Webb
Executive Commissioning Editor: Birgit Gruber
Assistant Editor: Colleen Goldring
Publishing Assistant: Ellie Scott
Project Editor: Juliet Booker
Content Editor: Nicole Burnett
Copy Editor: Richard Walshe

Marketing:
Senior Marketing Manager: Louise Breinholt
Marketing Executive: Kate Batchelor

Composition Services:
Compositor: Laserwords Private Limited, Chennai, India
Proof Reader: Alex Grey
Indexer: Annette Musker

Contents

About the Authors

Gertrud Bjørnvig is an agile requirements expert with over 20 years' experience in system development. She is a co-founder of the Danish Agile User Group and is a partner in Scrum Training Institute.

Jim Coplien is a software industry pioneer in object-oriented design, architecture patterns, and agile software development. He has authored several books on software design and agile software development, and is a partner in the Scrum Training Institute.

Preface

What my grandfather did was create options. He worked hard to allow my father to have a better education than he did, and in turn my father did the same.

Danny Hillis, quoted in *The Clock of the Long Now*, p. 152.

Harry Grinnell, who was co-author James Coplien's grandfather, was a life-long postal worker, but many of his life's accomplishments can be found in his avocations. His father was an alcoholic and his mother a long-suffering religious woman. Grandpa Harry dropped out of school after eighth year to take a job in a coal yard to put food on the table after much of the family budget had gone to support his father's habit. Harry would go on to take up a job as a postal worker in 1925 at the age of 19, and married Jim's grandmother the next year. He faced the changes of the Great Depression, of two world wars, and of great economic and social change.

You're probably wondering why an Agile book starts with a story about Grandpa Harry. It's because his avocation as a master craftsman in woodworking together with his common-sense approach to life offer a fitting metaphor for the Agile and Lean styles of development. This is a book about common sense. Of course, one person's common sense is another one's revelation. If you are just learning about Agile and Lean, or are familiar only with their pop versions, you may find new insights here. Even if you know about Agile and Lean and are familiar with architecture, you're likely to learn from this book about how the two ideas can work and play together.

As a postal employee, Grandpa Harry of course worked to assure that the post office met its business objectives. He worked in the days when the U.S. postal service was still nationalized; the competition of UPS and DHL didn't threaten postal business until late in his career. Therefore, the focus of his work wasn't as much on business results and profit as it was on quality and individual customer service. Grandpa Harry was a rural mail carrier who delivered to rural Wisconsin farmers, one mailbox at a time, six days a week, come rain or shine. It wasn't unusual for him to encounter

a half-meter of snow, or snow drifts two meters high on his daily rounds. Flooded creek valleys might isolate a farm, but that could be no obstacle. He delivered mail in his rugged four-wheel drive Willys Jeep that he bought as an Army surplus bargain after World War II. He outfitted it with a snowplow in the winter, often plowing his way to customers' mailboxes.

There are many good parallels between Grandpa Harry's approach to life and the ideals of Lean and Agile today. You need close contact with your customer and have to earn the trust of your customer for Agile to work. It's not about us-and-them as typified by contracts and negotiation; such was not part of Grandpa Harry's job, and it's not the job of a modern software craftsperson in an Agile setting. The focus is on the end user. In Grandpa Harry's case, that end user was the child receiving a birthday card from a relative thousands of miles away, or a soldier in Viet Nam receiving a care package from home after it being entrusted to the United States Postal Service for dispatching to its destination, or the flurry of warm greetings around the Christmas holidays. The business entity in the middle – in Grandpa Harry's case, the U.S. Postal Service, and in our case, our *customers* – tend to become transparent in the light of the *end users*' interests. Customers care about the software CD as a means for profit; end users have a stake in those products' use cases to ensure some measure of day-to-day support of their workflow.

To say this is neither to deny customers a place, nor to infer that our employers' interests should be sacrificed to those of our ultimate clientele. A well-considered system keeps evolving so *everybody* wins. What Grandpa Harry worked for was called the postal *system*: it was really a system, characterized by systems thinking and a concern for the whole. So, yes, the end user was paramount, but the system understood that a good post office working environment and happy postal workers were an important means to the end of user satisfaction. Postal workers were treated fairly in work conditions and pay; exceptions were so unusual that they made the news. In the same sense, the Agile environment is attentive to the needs of the programmer, the analyst, the usability engineer, the manager, and the funders. Tools such as architectural articulation, good requirements management, and lean minimalism improve the quality of life for the production side too. That is important because it supports the business goals. It is imperative because, on a human scale, it is a scandal to sacrifice development staff comfort to end user comfort.

Life in Grandpa Harry's time was maybe simpler than it is today, but many of the concepts of Lean and Agile are simple ideas that hearken back to that era. Just because things are simple doesn't mean they are simplistic. The modern philosopher Thomas Moore asks us to ''live simply, but be complicated'' (Moore 2001, p. 9). He notes that when Thoreau went to Walden Pond, his thoughts became richer and more complicated the

simpler his environment became. To work at this level is to begin to experience the kinds of generative processes we find in nature. Great things can arise from the interactions of a few simple principles. The key, of course, is to find those simple principles.

Grandpa Harry was not much one for convention. He was a doer, but thinking backed his doing. In this book, we'll certainly relate practices and techniques from 15 years of positive experiences together with software partners worldwide. But don't take our word for it. This is as much a book about thinking as about doing, much as the Agile tradition (and the Agile Manifesto itself (Beck et al 2001)) is largely about doing, and the Lean concepts from the Toyota tradition relate more to planning and thinking (Liker 2004, ff. 237). These notions of thinking are among the lost practices of Agile. Agile perhaps lost this focus on thinking and product in its eagerness to shed the process-heavy focus of the methodology-polluted age of the 1980s.

Grandpa Harry's life is also a reminder that we should value timeless domain knowledge. Extreme Programming (XP) started out in part by consciously trying to do exactly the opposite of what conventional wisdom recommended, and in part by limiting itself to small-scale software development. Over time, we have come full circle, and many of the old practices are being restored, even in the halls and canon of Agiledom. System testing is now ''in,'' as is up-front architecture – even in XP (Beck 1999, p. 113, 2005, p. 28). We're starting to recover insights from past generations of system development that perhaps we didn't even appreciate at the time; if we did, we've forgotten. Many of these ''old'' ideas such as architecture and planning, and even some of the newer ideas such as use cases that have fallen into disfavor, deserve a second look. We find many of these ideas re-surfacing under different names anyhow in today's Agile world: architecture reappears as metaphor, and use cases reappear as the collections of user story cards and supplementary constraint and testing cards that go with them (Cohn 2004), or as the requirement structuring we find in story maps (Patton 2009).

The domain knowledge in this book goes beyond standing on our tiptoes to standing on the shoulders of giants. We have let our minds be sharpened by people who have earned broad respect in the industry – and double that amount of respect from us – from Larry Constantine and David Parnas to Jeff Sutherland and Alistair Cockburn. We also draw on our own experience in software development going back to our first hobby programs in the 1960s, and our software careers going back to the early 1970s (Coplien) and 1980s (Bjørnvig). We draw lightly on Coplien's more recent book together with Neil Harrison, *Organizational Patterns of Agile Software Development* (Coplien and Harrison 2004), which stands on ten years of careful research into software development organizations worldwide. Its findings stand as

the foundations of the Agile discipline, having been the inspiration for stand-up meetings in the popular Scrum product management framework (Sutherland 2003, 2007), and of much of the structural component of XP (Fraser et al 2003). Whereas the previous book focused on the organizational with an eye to the technical, this one focuses on the technical with an eye to the organizational. Nerds: enjoy!

As long as we have you thinking, we want you thinking about issues of lasting significance to your work, your enterprise, and the world we as software craftsmen and craftswomen serve. If we offer a technique, it's because we think it's important enough that you'd notice the difference in the outcome of projects that use it and those that don't. We won't recommend exactly what incantation of words you should use in a user story. We won't bore you with whether to draw class diagrams bottom-up or top-down nor, in fact, whether to draw diagrams at all. We won't try to indoctrinate you with programming language arguments – since the choice of programming language has rarely been found to matter in any broadly significant way. As we know from Agile and Lean thinking, people and values matter most, and bring us to ideals such as *caring*. The byline on the book's cover, *Software as if people mattered,* is a free re-translation of the title of Larry Constantine's keynote that Coplien invited him to give at OOPSLA in 1996. People are ultimately the focus of all software, and it's time that we show enough evidence to convict us of honoring that focus. We will dare use the phrase ''common sense,'' as uncommon as its practice is. We try to emphasize things that matter – concrete things, nonetheless.

There is a subtext to this book for which Grandpa Harry is a symbol: valuing timelessness. In our software journey the past 40 years we have noticed an ever-deepening erosion of concern for the long game in software. This book is about returning to the long game. However, this may be a sobering concern as much for society in general as it is for our relatively myopic view of software. To help drive home this perspective we've taken inspiration from the extended broadside *The Clock of the Long Now* (Brand 1999), which is inspired in no small part by software greats including Mitchell Kapoor and Daniel Hillis. The manuscript is sprinkled with small outtakes from the book, such as this one:

> What we can do is convert the design of software from brittle to resilient, from heedlessly headlong to responsible, and from time corrupted to time embracing. (Brand 1999, p. 86)

These outtakes are short departures from the book's (hopefully practical) focus on architecture and design that raise the principles to levels of social relevance. They are brief interludes to inspire discussions around dinner and reflection during a walk in the woods. We offer them neither to

preach at you nor to frighten you, but to help contextualize the humble software-focused theses of this book in a bigger picture.

We've worked with quite a few great men and women to develop and refine the ideas in this book. It has been an honor sparring with Trygve Reenskaug about his DCI (Data, Context and Interaction) architecture, learning much from him and occasionally scoring an insight. We have also traded many notes with Richard Öberg, whose Qi4j ideas echo many aspects of DCI, and it has been fun as we've built on each other's work.

We've also built on the work of many people who started coding up DCI examples after a presentation at JaOO in 2008: Serge Beaumont at Xebia (Python), Ceasario Ramos (who thoroughly explored the Java space), Jesper Rugård Jensen (ditto), Lars Vonk (in Groovy), David Byers (also in Python), Anders Narwath (JavaScript), Unmesh Joshi (AspectJ), Bill Venners (Scala, of course), and Christian Horsdal Gammelgaard of Mjølner (C#/.Net). Many examples in this book build on Steen Lehmann's exploration of DCI in Ruby. We, and the entire computing community, should be ever grateful to all of these folks.

We appreciate all the good folks who've devoted some of their hours to reading and reflecting on our early manuscripts. Trygve, again, offered many useful suggestions and his ideas on the manuscript itself have helped us clarify and sharpen the exposition of DCI. It goes without saying that the many hours we spent with Trygve discussing DCI, even apart from any focus on this book, were memorable times. Trygve stands almost as a silent co-author of this book, and we are ever indebted to him and to his wife Bjørg for many hours of stimulating discussion. Thanks, Trygve!

We are also indebted to Rebecca Wirfs-Brock for good discussions about use cases, for clarifying the historical context behind them, for confirming many of our hunches, and for straightening out others.

We owe special thanks to Lars Fogtmann Sønderskov for a detailed review of an early version of the manuscript. His considerable experience in Lean challenged our own thinking and pushed us to review and re-think some topics in the book. Brett Schuchert, who was a treasured reviewer for *Advanced C++* 20 years ago, again treated us to a tough scouring of the manuscript. Thanks, Brett! Thanks also to our other official reviewer, the renowned software architect Philippe Kruchten, who helped us make some valuable connections to other broadly related work. Atzmon Hen-tov not only found many small mistakes but also helped us frame the big picture, and his comments clearly brought years of hard-won insights from his long journey as a software architect. Thanks to the many other reviewers who scoured the manuscript and helped us to polish it: Roy Ben Hayun, Dennis L DeBruler, Dave Byers, Viktor Grgic, Neil Harrison, Bojan Jovičić, Urvashi Kaul, Steen Lehmann, Dennis Mancl, Simon Michael, Sandra Raffle Carrico, Jeppe Kilberg Møller, Rune Funch Søltoft, Mikko

Suonio, and Lena Nikolaev. Many ideas came up in discussions at the Agile Architecture course in Käpylä, Finland, in October 2008: Aleksi Ahtiainen, Aki Kolehmainen, Heimo Laukkanen, Mika Leivo, Ari Tikka, and Tomi Tuominen all contributed mightily. Thanks, too, to James Noble, Peter Bunus, and John McGregor for their evaluations of the book proposal in its formative days and for their encouragement and feedback.

A big thanks to Paul Mitchell Design, Ltd., for a great job working with us on the book cover design. Claire Spinks took on the unenviable job of copy editing and helped us polish up the manuscript. And, of course, many thanks to Birgit Gruber, our editor, and to Ellie Scott, who oversaw much of the editorial hand-holding during the book's formative years.

Thanks to Magnus Palmgård of Tobo, Sweden for providing a lovely venue for several months of thoughtful reflection and writing.

We appreciate the pioneers who have gone before us and who have influenced the way we look at the world and how we keep learning about it. Phillip Fuhrer lent useful insights on problem definition. We had thoughtful E-mail conversations with Larry Constantine, and it was a pleasure to again interact with him and gain insight on coupling and cohesion from a historical context. Some of his timeless ideas on coupling, cohesion, and even Conway's Law (which he named) are coming back into vogue. Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, Andrew Black, Roel Wuyts and others laid the foundations for traits. Trygve Reenskaug, Jeff Sutherland, Alistair Cockburn, Jerry Weinberg, and hundreds of others have all led us here. So, of course, has Grandpa Harry.

CHAPTER 1

Introduction

We are changing the Earth more rapidly than we are understanding it.
– Peter Vitousek et al. quoted in *The Clock of the Long Now*, p. 9.

A proper book isn't just a collection of facts or even of practices: it reflects a cause and a mission. In the preface we couched this book in a broad context of social responsibility. Just as the motivation section (goal in context, summary, or whatever else you call it) in a use case helps the analyst understand requirements scenarios, this chapter might shed light on the ones that follow. It describes our philosophy behind the book and the way we present the ideas to you. If you're tempted to jump to a whirlwind tour of the book's contents, you might proceed to Chapter 2. However, philosophy is as important as the techniques themselves in a Lean and Agile world. We suggest you read through the introduction at least once, and tuck it away in your memory as background material for the other chapters that will support your day-to-day work.

1.1 The Touchstones: Lean and Agile

Lean and Agile are among the most endearing buzzwords in software today, capturing the imagination of management and nerds alike. Popular management books of the 1990s (Womack et al 1991) coined the term *Lean* for the management culture popularized by the Japanese auto industry, and which can be traced back to Toyota where it is called The Toyota Way. In vernacular English, *minimal* is an obvious synonym for *Lean*, but to link lean to minimalism alone is misleading.

Lean's primary focus is the enterprise value stream. Lean grabs the consumer world and pulls it through the value stream to the beginnings of development, so that every subsequent activity adds value. Waste in production reduces value; constant improvement increases value. In Western cultures managers often interpret Lean in terms of its production practices: just-in-time, end-to-end continuous flow, and reduction of inventory. But its real heart is The Lean Secret: an ''all hands on deck'' mentality that permeates every employee, every manager, every supplier, and every partner. Whereas the Agile manifesto emphasizes customers, Lean emphasizes stakeholders – with everybody in sight being a stakeholder.

Lean architecture and Agile feature development aren't about working harder. They're not about working ''smarter'' in the academic or traditional computer science senses of the word ''smart.'' They are much more about focus and discipline, supported by common-sense arguments that require no university degree or formal training. This focus and discipline shines through in the roots of Lean management and in many of the Agile values.

We can bring that management and development style to software development. In this book, we bring it to software architecture in particular. Architecture is the big-picture view of the system, keeping in mind that the best big pictures need not be grainy. We don't feel a need to nail down a scientific definition of the term; there are too many credible definitions to pick just one. For what it's worth, the IEEE defines it this way:

> . . . The fundamental organization of a system embodied in its components, their relationships to each other, and to the environment and the principles guiding its design and evolution. (IEEE1471 2007)

Grady Booch gives us this simple definition:

> Architecture represents the significant design decisions that shape a system, where *significant* is measured by cost of change. (Booch 2006)

That isn't too bad. But more generally, we define architecture as the *form* of a system, where the word *form* has a special meaning that we'll explore a bit later. For now, think of it as relating to the first three components of the IEEE definition. No matter how we care to define it, software architecture should support the enterprise value stream even to the extent that the source code itself should reflect the end user mental model of the world. We will deliver code just in time instead of stockpiling software library warehouses ahead of time. We strive towards the practice of continuous flow.

Each of these practices is a keystone of Lean. But at the heart of Lean architecture is the team: the ''all hands on deck'' mentality that everyone is in some small part an architect, and that everyone has a crucial role to play

in good project beginnings. We want the domain experts (sometimes called the architects) present as the architecture takes shape, of course. However, the customer, the developer, the testers, and the managers should also be fully present at those beginnings.

This may sound wasteful and may create a picture of chaotic beginnings. However, one of the great paradoxes of Lean is that such intensity at the beginning of a project, with heavy iteration and rework in design, actually reduces overall life cycle cost and improves product quality. Apply those principles to software, and you have a lightweight up-front architecture. *Lightweight* means that we reduce the waste incurred by rework (from inadequate planning), unused artifacts (such as comprehensive documentation and speculative code), and wait states (as can be caused by the review life cycle of architecture and design documents, or by handoffs between functional teams).

Software folks form a tribe of sorts (Nani 2006) that holds many beliefs, among them that architecture is *hard*. The perception comes in part from architecture's need for diverse talents working together, compounded by the apparently paradoxical need to find the basic form of something that is essentially complex. Even more important, people confuse ''takes a long time'' with ''hard.'' That belief in turn derives from our belief in specialization, which becomes the source of handoffs: the source of the delays that accumulate into long intervals that makes architecture look hard. We tend to gauge our individual uncertainty and limited experience in assessing the difficulty of design, and we come up short, feeling awkward and small rather than collaborative and powerful. Architecture requires a finesse and balance that dodges most silver bullets. Much of that finesse comes with the Lean Secret: the takes-a-long-time part of *hard* becomes softer when you unite specialists together in one room: *everybody, all together, from early on*. We choose to view *that* as hard because, well, that's how it's always been, and perhaps because we believe in individuals first and interactions second.

Neither Lean nor Agile alone make architecture look easy. However, architecture needn't be intrinsically hard. Lean and Agile together illuminate architecture's value. Lean brings careful up-front planning and ''everybody, all together, from early on'' to the table, and Agile teaches or reminds us about feedback. Together they illuminate architecture's value: Lean, for how architecture can reduce waste, inconsistency, and irregular development; and Agile, for how end user engagement and feedback can drive down long-term cost. Putting up a new barn is hard, too. As Grandpa Harry used to say, many hands make light work, and a 19th-century American farm neighborhood could raise a new barn in a couple of days. So can a cross-functional team greatly compress the time, and therefore the apparent difficulty, of creating a solid software architecture.

Another key Lean principle is to focus on long-term results (Liker 2004, pp. 71–84). Lean architecture is about doing what's important *now* that will keep you in the game for the long term. It is nonetheless important to contrast the Lean approach with traditional approaches such as ''investing for the future.'' Traditional software architecture reflects an investment model. It capitalizes on heavyweight artifacts in software inventory and directs cash flow into activities that are difficult to place in the customer value stream. An industry survey of projects with ostensibly high failure rates (as noted in Glass (2006), which posits that the results of the Standish survey may be rooted in characteristically dysfunctional projects) found that 70% of the software they build is never used (Standish Group 1995).

Lean architecture carefully slices the design space to deliver exactly the artifacts that can support downstream development in the long term. It avoids wasteful coding that can better be written just after demand for it appears and just before it generates revenues in the market. From the programmer's perspective, it provides a way to capture crucial design concepts and decisions that must be remembered throughout feature production. These decisions are captured in code that is delivered as part of the product, not as extraneous baggage that becomes irrelevant over time.

With such Lean foundations in place, a project can better support Agile principles and aspire to Agile ideals. If you have all hands on deck, you depend more on people and interactions than on processes and tools. If you have a value stream that drives you without too many intervening tools and processes, you have customer engagement. If we reflect the end user mental model in the code, we are more likely to have working software. And if the code captures the form of the domain in an uncluttered way, we can confidently make the changes that make the code serve end user wants and needs.

This book is about a Lean approach to domain architecture that lays a foundation for Agile software change. The planning values of Lean do not conflict with the inspect-and-adapt principles of Agile: allocated to the proper development activities, each supports the other in the broader framework of development. We'll revisit that contrast in a little while (Section 1.4), but first, let's investigate each of Lean Architecture and Agile Production in more detail.

1.2 Lean Architecture and Agile Feature Development

The Agile Manifesto (Beck et al 2001) defines the principles that underlie the Agile vision, and the Toyota Way (Liker 2004) defines the Lean

vision. This book offers a vision of architecture in an organization that embraces these two sets of ideals. The Lean perspective focuses on how we develop the overall system form by drawing on experience and domain knowledge. The Agile perspective focuses on how that informed form helps us respond to change, and sometimes even to plan for it. How does that vision differ from the classic, heavyweight architectural practices that dominated object-oriented development in the 1980s? We summarize the differences in Table 1-1.

Table 1-1 What is Lean Architecture?

Lean Architecture	Classic Software Architecture
Defers engineering	Includes engineering
Gives the craftsman "wiggle room" for change	Tries to limit large changes as "dangerous" (fear change?)
Defers implementation (delivers lightweight APIs and descriptions of relationships)	Includes much implementation (platforms, libraries) or none at all (documentation only)
Lightweight documentation	Documentation-focused, to describe the implementation or compensate for its absence
People	Tools and notations
Collective planning and cooperation	Specialized planning and control
End user mental model	Technical coupling and cohesion

- Classic software architecture tends to embrace engineering concerns too strongly and too early. Agile architecture is about form, and while a system must obey the same laws that apply to engineering when dealing with form, we let form follow proven experience instead of being driven by supposedly scientific engineering rationales. Those will come soon enough.
- This in turn implies that the everyday developers should use their experience to tailor the system form as new requirements emerge and as they grow in understanding. Neither Agile nor Lean gives coders wholesale license to ravage the system form, but both honor the value of adaptation. Classic architecture tends to be fearful of large changes, so it focuses on incremental changes only to existing artifacts: adding a new derived class is not a transformation of form (architecture), but of structure (implementation). In our combined Lean/Agile approach, we reduce risk by capturing domain architecture, or basic

system form, in a low-overhead way. Furthermore, the architecture encourages *new* forms in those parts of the system that are likely to change the most. Because these forms aren't pre-filled with premature structure, they provide less impedance to change than traditional approaches. This is another argument for a true architecture of the forms of domain knowledge and function rather than an architecture based on structure.

- Classic software architecture sometimes rushes into implementation to force code reuse to happen or standards to prevail. Lean architecture also adopts the perspective that standards are valuable, but again: at the level of form, protocols, and APIs, rather than their implementation.
- Some classic approaches to software architecture too often depend on, or at least produce, volumes of documentation at high cost. The documentation either describes ''reusable'' platforms in excruciating detail or compensates for the lack of a clarifying implementation. Architects often throw such documentation over the wall into developers' cubicles, where it less often used than not. Agile emphasizes communication, and sometimes written documentation is the right medium. However, we will strive to document only the stuff that really matters, and we'll communicate many decisions in code. That kills two birds with one stone. The rest of the time, it's about getting everybody involved face-to-face.
- Classic architectures too often focus on methods, rules, tools, formalisms, and notations. Use them if you must. But we won't talk much about those in this book. Instead, we'll talk about valuing individuals and their domain expertise, and valuing the end-user experience and their mental models that unfold during analysis.
- Both Lean and classic architecture focus on long-term results, but they differ in how planning is valued. Even worse than heavy planning is a prescription to follow the plan. Lean focuses on what's important now, whenever ''now'' is – whether that is hitting the target for next week's delivery or doing long-term planning. It isn't only to eliminate waste by avoiding what is *never* important (dead code and unread documents), but has a subtler timeliness. Architecture isn't an excuse to defer work; on the contrary, it should be a motivation to embrace implementation as soon as decisions are made. We make decisions and produce artifacts at the most responsible times.

As we describe it in this book, Lean architecture provides a firm foundation for the ongoing business of a software enterprise: providing timely features to end users.

1.3 Agile Production

If your design is lean, it produces an architecture that can help you be more Agile. By Agile, we mean the values held up by the Agile Manifesto:

> We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:
>
> *Individuals and interactions* over processes and tools
>
> *Working software* over comprehensive documentation
>
> *Customer collaboration* over contract negotiation
>
> *Responding to change* over following a plan
>
> That is, while there is value in the items on the right, we value the items on the left more. (Beck et al 2001)

1.3.1 Agile Builds on Lean

Just as with the ''all hands on deck'' approach of Lean, Agile development also embraces close person-to-person contact, particularly with the clients. Unlike the tendencies of Lean, or much of today's software architecture, our vision of Agile production plans for change. Lean architecture provides a context, a vocabulary, and productive constraints that make change easier and perhaps a little bit more failure-proof. It makes explicit a value stream along which stakeholder changes can propagate without being lost. We can respond to market whims. And we love market whims – because that's how we provide satisfaction and keep the enterprise profitable.

Agile production not only builds on a Lean domain architecture, but it stays Lean with its focus on code – working software. The code is the design. No, *really*. The code is the best way to capture the end user mental models in a form suitable to the shaping and problem solving that occur during design. We of course also need other design representations that close the feedback loop to the end user and other stakeholders for whom code is an unsuitable medium, so lightweight documentation may be in order – we'll introduce that topic in Section 1.6.4. We take this concept beyond platitudes, always striving to capture the end-user model of program execution in the code.

Classic architectures focus on what doesn't change, believing that foundations based on domain knowledge reduce the cost of change. Agile understands that nothing lasts forever, and it instead focuses explicitly on what is likely to change. Here we balance the two approaches, giving neither one the upper hand.

Lean also builds on concepts that most people hold to be fundamental to Agile. The Lean notion of value streams starting with end users recalls individual and interactions as well as customer focus. The Lean notion of reduced waste goes hand-in-hand with Agile's view of documentation. It is not about Lean versus Agile and neither about building Lean on top of Agile nor Agile on top of Lean. Each one is a valuable perspective into the kind of systems thinking necessary to repeatedly deliver timely products with quality.

1.3.2 The Scope of Agile Systems

> ***Electronically accelerated market economies have swept the world for good reasons. They are grass-roots driven (by customers and entrepreneurs), swiftly adaptive, and highly rewarding.***
>
> ***The Clock of the Long Now*, p. 25.**

Software architects who were raised in the practices and experience of software architecture of the 1970s and 1980s will find much comfort in the Lean parts of this book, but may find themselves in new territory as they move into the concepts of Agile production. Architecture has long focused on stability while Agile focuses on change. Agile folks can learn from the experience of previous generations of software architecture in how they *plan* for change. As we present a new generation of architectural ideas in this book, we respond to change more directly, teasing out the form even of those parts of software we usually hold to be dynamic. We'll employ use cases to distill the stable backbones of system behavior from dozens or hundreds of variations. We go further to tease out the common rhythms of system behavior into the roles that are the basic concepts we use to describe it and the connections between them.

Grandpa Harry used to say that necessity is the mother of invention, so need and user *expectation* are perhaps the mother and father of change. People expect software to be able to change at lightening speed in modern markets. On the web, in financial services and trading, and in many other market segments, the time constants are on the order of hours or days. The users themselves interact with the software on time scales driven by interactive menus and screens rather than by daily batch runs. Instead of being able to stack the program input on punched cards ahead of time, decisions about the next text input or the next menu selection are made seconds or even milliseconds before the program must respond to them.

Agile software development is well suited to such environments because of its accommodation for change. Agile is less well suited to environments

where feedback is either of little value (such as the development of a protocol based on a fully formal specification and development process) or is difficult to get (such as from software that is so far embedded in other systems that it has no obvious interaction with individuals). Libraries and platforms often fall into this category: how do you create short feedback loops that can steer their design? Sometimes a system is so constrained by its environment that prospects for change are small, and Agile approaches may not help much.

Lean likewise shines in some areas better than others. It's overkill for simple products. While Lean can deal with *complicated* products, it needs innovation from Agile to deal with *complex* products where we take *complicated* and *complex* in Snowden's (Snowden 2009) terms. Complicated systems can rely on fact-based management and can handle known unknowns, but only with expert diagnosis. Complex systems have unknown unknowns, and there is no predictable path from the current state to a better state (though such paths can be rationalized in retrospect). There are no right answers, but patterns emerge over time. Most of the organizational patterns cited in this book relate to complex problems. Even in dealing with complex systems, Agile can draw on Lean techniques to establish the boundary conditions necessary for progress.

The good news is that most systems have both a Lean component and an Agile component. For example, embedded or deeply layered system software can benefit from domain experience and the kind of thorough analysis characteristic of Lean, while other software components that interact with people can benefit from Agile.

Below the realm of Lean and Agile lie *simple* systems, which are largely knowable and predictable, so we can succeed even if our efforts fall short of both Lean and Agile. On the other end are *chaotic* system problems such as dealing with a mass system outage. There, even patterns are difficult to find. It is important to act quickly and to just find something that works rather than seeking the right answer. Chaotic systems are outside the scope of our work here.

1.3.3 Agile and DCI

If we can directly capture key end-user mental models in the code, it radically increases the chances the code will work. The fulfillment of this dream has long eluded the object-oriented programming community, but the recent work on the Data, Context and Interaction (DCI) architecture, featured in Chapter 9, brings this dream much closer to reality than we have ever realized. And by ''work'' we don't mean that it passes tests or

that the green bar comes up: we mean that it does what the user *expects* it to do.[1] The key is the architectural link between the end user mental model and the code itself.

1.4 The Book in a Very Small Nutshell

We'll provide a bit meatier overview in Chapter 2, but here is the one-page (and a bit more) summary of the technical goodies in the book, for you nerds reading the introduction:

- System architecture should reflect the end users' mental model of their world. This model has two parts. The first part relates to the user's thought process when viewing the screen, and to what the system *is*: its *form*. The second part relates to what end users *do* – interacting with the system – and how the system should respond to user input. This is the system *functionality*. We work with users to elicit and develop these models and to capture them in code as early as possible. Coupling and cohesion (Stevens, Myers, and Constantine 1974) follow from these as a secondary effect.
- To explore both form and function requires up-front engagement of all stakeholders, and early exploration of their insights. Deferring interactions with stakeholders, or deferring decisions beyond the responsible moment slows progress, raises cost, and increases frustration. A team acts like a team from the start.
- Programming languages help us to concretely express form in the code. For example, abstract base classes can concretely express domain models. Development teams can build such models in about one Scrum Sprint: a couple of weeks to a month. Design-by-contract, used well, gets us closer to running code even faster. Going beyond this expression of *form* with too much *structure* (such as class implementation) is not Lean, slows things down, and leads to rework.
- We can express complex system functionality in use cases. Lightweight, incrementally constructed use cases help the project to quickly capture and iterate models of interaction between the end user (actor) and the system, and to structure the relationships between scenarios.

[1] What users really *expect* has been destroyed by the legacy of the past 40 years of software deployment. It's really hard to find out what they actually *need*, and what they *want* too often reflects short-term end-user thinking. Our goal is to avoid the rule of least surprise: we don't want end users to feel unproductive, or to feel that the system implementers didn't understand their needs, or to feel that system implementers feel that they are stupid. Much of this discussion is beyond the scope of this book, though we will touch on it from time to time.

By making requirement dependencies explicit, use cases avoid dependency management and communication problems that are common in complex Agile projects. Simpler documents like User Narratives are still good enough to capture simple functional requirements.

- We can translate use case scenarios into algorithms, just in time, as new scenarios enter the business process. We encode these algorithms directly as *role methods*. We will introduce *roles* (implemented as role classes or *traits*) as a new formalism that captures the behavioral essence of a system in the same way that classes capture the essence of domain structure. Algorithms that come from use cases are more or less directly readable from the role methods. Their form follows function. This has profound implications for code comprehension, testability, and formal analysis. At the same time, we create or update classes in the domain model to support the new functionality. These classes stay fairly dumb, with the end-user scenario information separated into the role classes.
- We use a recent adaptation of traits to glue together role classes with the domain classes. When a use case scenario is enacted at run time, the system maps the use case actors into the objects that will support the scenario (through the appropriate role interface), and the scenario runs.

Got your attention? It gets even better. Read on.

1.5 Lean and Agile: Contrasting and Complementary

You should now have a basic idea of where we're heading. Let's more carefully consider Agile and Lean, and their relationships to each other and to the topic of software design.

One unsung strength of Agile is that it is more focused on the ongoing sustenance of a project than just its beginnings. The waterfall stereotype is patterned around greenfield development. It doesn't easily accommodate the constraints of any embedded base to which the new software must fit, nor does it explicitly provide for future changes in requirements, nor does it project what happens after the first delivery. But Agile sometimes doesn't focus enough on the beginnings, on the long deliberation that supports long-term profitability, or on enabling standards. Both Lean and Agile are eager to remove defects as they arise. Too many stereotypes of Lean and Agile ignore both the synergies and potential conflicts between Lean and Agile. Let's explore this overlap a bit.

Architects use notations to capture their vision of an ideal system at the beginning of the life cycle, but these documents and visions quickly become out-of-date and become increasingly irrelevant over time. If we constantly refresh the architecture in cyclic development, and if we express the architecture in living code, then we'll be working with an Agile spirit. Yes, we'll talk about architectural beginnings, but the right way to view software development is that everything after the first successful compilation is maintenance.

Lean is often cited as a foundation of Agile, or as a cousin of Agile, or today as a foundation of some Agile technique and tomorrow not. There is much confusion and curiosity about such questions in software today. Scrum inventor Jeff Sutherland refers to Lean and Scrum as separate and complementary developments that both arose from observations about complex adaptive systems (Sutherland 2008). Indeed, in some places Lean principles and Agile principles tug in different directions. The Toyota Way is based explicitly on standardization (Liker 2004, chapter 12); Scrum says always to inspect and adapt. The Toyota Way is based on long deliberation and thought, with rapid deployment only *after* a decision has been reached (Liker 2004, chapter 19); most Agile practice is based on rapid *decisions* (Table 1-2).

Table 1-2 Contrast between Lean and Agile.

Lean	Agile
Thinking and doing	Doing
Inspect-plan-do	Do-inspect-adapt
Feed-forward and feedback (design for change and respond to change)	Feedback (react to change)
High throughput	Low latency
Planning and responding	Reacting
Focus on Process	Focus on People
Teams (working as a unit)	Individuals (and interactions)
Complicated systems	Complex systems
Embrace standards	Inspect and adapt
Rework in design adds value, in making is waste	Minimize up-front work of any kind and rework code to get quality
Bring decisions forward (Decision Structure Matrices)	Defer decisions (to the last responsible moment)

Some of the disconnect between Agile and Lean comes not from their foundations but from common misunderstanding and from everyday pragmatics. Many people believe that Scrum insists that there be no specialists on the team; however, Lean treasures both seeing the whole as well as specialization:

> [W]hen Toyota selects one person out of hundreds of job applicants after searching for many months, it is sending a message – the capabilities and characteristics of individuals matter. The years spent carefully grooming each individual to develop depth of technical knowledge, a broad range of skills, and a second-nature understanding of Toyota's philosophy speaks to the importance of the individual in Toyota's system. (Liker 2004, p. 186)

Scrum insists on cross-functional team, but itself says nothing about specialization. The specialization myth arises in part from the XP legacy that discourages specialization and code ownership, and in part from the Scrum practice that no one use their specialization as an excuse to avoid other kind of work during a Sprint (Østergaard 2008).

If we were to look at Lean and Agile through a coarse lens, we'd discover that Agile is about *doing* and that Lean is about *thinking* (about continuous process improvement) *and* doing. A little bit of thought can avoid a lot of doing, and in particular *re*-doing. Ballard (2000) points out that a little rework and thought in design adds value by reducing product turn-around time and cost, while rework during making is waste (Section 3.1.2). System-level-factoring entails a bit of both, but regarding architecture only as an emergent view of the system substantially slows the decision process. Software isn't soft, and architectures aren't very malleable once developers start filling in the general *form* with the *structure* of running code. Lean architecture moves beyond structure to form. Good form is Lean, and that helps the system be Agile.

Lean is about complicated things; Agile is about complexity. Lean principles support predictable, repeatable processes, such as automobile manufacturing. Software is hardly predictable, and is almost always a creative – one might say artistic – endeavor (Snowden and Boone 2007). Agile is the art of the possible, and of expecting the unexpected.

This book tells how to craft a Lean architecture that goes hand-in-glove with Agile development. Think of Lean techniques, or a Lean architecture, as a filter that prevents problems from finding a way into your development stream. Keeping those problems out avoids rework.

Lean principles lie at the heart of architectures behind Agile projects. Agile is about embracing change, and it's hard to reshape a system if there is too much clutter. Standards can reduce decision time and can reduce

work and rework. Grandpa Harry used to say that a stitch in time saves nine; so up-front thinking can empower decision makers to implement decisions lightening-fast with confidence and authority. Lean architecture should be rooted in the thought processes of good domain analysis, in the specialization of deeply knowledgeable domain experts, and once in a while on de facto, community, or international standards.

1.5.1 The Lean Secret

The human side of Lean comes down to this rule of thumb:

Everybody, All together, Early On

Using other words, we also call this ''all hands on deck.'' Why is this a ''secret''? Because it seems that teams that call themselves Agile either don't know it or embrace it only in part. Too often, the ''lazy'' side of Lean shines through (avoiding excess work) while teams set aside elements of social discipline and process. Keeping the ''everybody'' part secret lets us get by with talking to the customer, which has some stature associated with it, while diminishing focus on other stakeholders like maintenance, investors, sales, and the business. Keeping the ''early on'' part a secret makes it possible to defer decisions – and to decide to defer a decision is itself a decision with consequences. Yet all three of these elements are crucial to the human foundations of Lean. We'll explore the Lean Secret in more depth in Chapter 3.

1.6 Lost Practices

We speak . . . about the events of decades now, not centuries. One advantage of that, perhaps, is that the acceleration of history now makes us all historians.

***The Clock of the Long Now*, p. 16.**

As we distilled our experience into the beginnings of this book, both of us started to feel a bit uncomfortable and even a little guilty about being old folks in an industry we had always seen fueled by the energy of the young, the new, and the restless. As people from the patterns, Lean and object communities started interacting more with the new Agile community, however, we found that we were in good company. Agile might be the first major software movement that has come about as a broad-based mature set of disciplines.

Nonetheless, as Agile rolled out into the industry the ties back to experience were often lost. That Scrum strived to remain agnostic with respect to

software didn't help, so crucial software practices necessary to Scrum's success were too easily forgotten. In this book we go back to the fundamental notions that are often lost in modern interpretation or in the practice of XP or Scrum. These include system and software architecture, requirements dependency management, foundations for usability, documentation, and others.

1.6.1 Architecture

> ***Electronically accelerated market economies have swept the world for good reasons. They are grass-roots driven (by customers and entrepreneurs), swiftly adaptive, and highly rewarding. But among the things they reward, as McKenna points out, is short-sightedness.***
>
> ***The Clock of the Long Now*, p. 25.**

A project must be strong to embrace change. Architecture not only helps give a project the firmness necessary to stand up to change, but also supports the crucial Agile value of communication. Jeff Sutherland has said that he never has, and never would, run a software Scrum without software architecture (Coplien and Sutherland 2009). We build for change.

We know that ignoring architecture in the long term increases long-term cost. Traditional architecture is heavily front-loaded and increases cost in the short term, but more importantly, pushes out the schedule. This is often the case because the architecture invests too much in the actual structure of implementation instead of sticking with form. A structure-free up-front architecture, constructed as pure form, can be built in days or weeks, and can lay the foundation for a system lifetime of savings. Part of the speedup comes from the elimination of wait states that comes from all-hands-on-deck, and part comes from favoring lightweight form over massive structure.

1.6.2 Handling Dependencies between Requirements

To make software work, the development team must know what other software and features lay the foundation for the work at hand. Few Agile approaches speak about the subtleties of customer engagement and end-user engagement. Without these insights, software developers are starved for the guidance they need while advising product management about product rollout. Such failures lead to customer surprises, especially when rapidly iterating new functionality into the customer stream.

Stakeholder engagement (Chapter 3) is a key consideration in requirements management. While both Scrum and XP encourage tight coupling to the customer, the word ''end user'' doesn't appear often enough, and

the practices overlook far too many details of these business relationships. That's where the subtle details of requirements show up – in the dependencies between them.

1.6.3 Foundations for Usability

The Agile Manifesto speaks about working software, but nothing about usable software. The origins of Agile can be traced back to object orientation, which originally concerned itself with capturing the end-user model in the code. Trygve Reenskaug's Model-View-Controller (MVC) architecture makes this concern clear and provides us a framework to achieve usability goals. In this book we build heavily on Trygve's work, both in the classic way that MVC brings end user mental models together with the system models, and on his DCI work, which helps users enact system functionality.

1.6.4 Documentation

Suppose we wanted to improve the quality of decisions that have long-term consequences. What would make decision makers feel accountable to posterity as well as to their present constituents? What would shift the terms of debate from the immediate consequences of the delayed consequences, where the real impact is? It might help to have the debate put on the record in a way that invites serious review.

***The Clock of the Long Now*, p. 98.**

Documentation gets a bad rap. Methodologists too often miss the point that documentation has two important functions: to *communicate* perspectives and decisions, and to *remember* perspectives and decisions. Alistair Cockburn draws a similar dichotomy between documentation that serves as a *reminder* for people who were there when the documented discussions took place, and as a *tutorial* for those who weren't (Cockburn 2007, pp. 23–24). Much of the Agile mindset misses this dichotomy and casts aspersions on any kind of documentation. Nonetheless, the Agile manifesto contrasts the waste of documentation with the production of working code: where code can communicate or remember decisions, redundant documentation may be a waste.

The Agile manifesto fails to explicitly communicate key foundations that lie beneath its own well-known principles and values. It is change that guides the Agile process; nowhere does the Manifesto mention learning or experience. It tends to cast human interaction in the framework of code development, as contrasted with processes and tools, rather than in the framework of community-building or professional growth. Documentation has a role there.

We should distinguish the act of writing a document from the long-term maintenance of a document. A whiteboard diagram, a CRC card, and a

diagram on the back of a napkin are all design documents, but they are documents that we rarely archive or return to over time. Such documentation is crucial to Agile development: Alistair Cockburn characterizes two people creating an artifact on a whiteboard as the most effective form of common engineering communication (Figure 1-1).

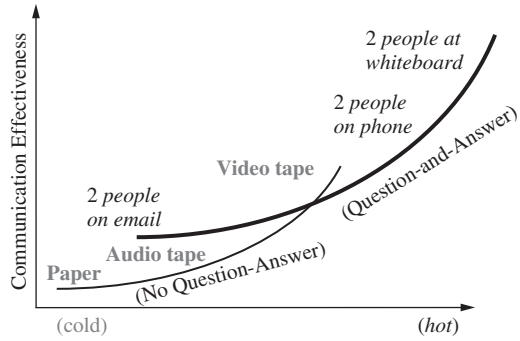

Figure 1-1 Forms of communication documentation. From Cockburn (2007, p. 125).

It is exactly this kind of communication, supplemented with the artifact that brings people together, that supports the kind of dynamics we want on an Agile team. From this perspective, documentation is fundamental to any Agile approach. There is nothing in the Manifesto that contradicts this: it cautions only against our striving for *comprehensive* documentation, and against a value system that places the documentation that serves the team ahead of the artifacts that support end-user services.

In the 1980s, too many serious software development projects were characterized by heavyweight write-only documentation. Lean architecture replaces the heavyweight notations of the 1980s with lightweight but expressive code. There in fact isn't much new or Agile in this: such was also the spirit of literate programming. Lean architecture has a place for lightweight documentation both for communication and for team memory. Experience repeatedly shows that documentation is more crucial in a geographically distributed development than when the team is collocated, and even Agile champions such as Martin Fowler agree (Fowler 2006).

Code Does Not Stand Alone

In general, ''the code is the design'' is a good rule of thumb. But it is neither a law nor a proven principle. Much of the crowd that advocates Agile today first advocated such ideas as members of the pattern discipline. Patterns were created out of an understanding that code sometimes does not stand