

# **LEAN ARCHITECTURE FOR AGILE SOFTWARE DEVELOPMENT**

---

This manuscript is Copyright ©2009 by James O. Coplien and Gertrud Bjørnvig. All rights reserved. No portion of this manuscript may be copied or reproduced in any form nor by any means, whether electronic, reprographic, or photographic, without prior written consent of the authors and all pertinent legal stakeholders.

For a printable copy of this draft, contact Jim Coplien at [cope@gertrudandcope.com](mailto:cope@gertrudandcope.com)

Draft of August 2, 2009

DRAFT

# **LEAN ARCHITECTURE FOR AGILE SOFTWARE DEVELOPMENT**

---

**James O. Coplien**

**Gertrud Bjørnvig**

**JOHN WILEY AND SONS**

Chichester • New York • Brisbane • Toronto • Singapore

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book and Wiley was aware of a trademark claim, the designations have been pointed in initial caps or all caps.

The authors and publishers have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

Library of Congress Cataloging-in-Publication Data

Coplien, James O. and Gertrud Bjørnvig

Lean Software Architecture and Agile Production / James O. Coplien and Gertrud

Bjørnvig.

p. cm.

Includes bibliographical references and index

ISBN 0-XXX-YYYYYY-1

C++ (Computer program language) I. Title

QA7673.C153C675 2010

05. 13'3-dcl21

08-36336

CIP

Copyright ©2010 James O. Coplien and Gertrud Bjørnvig.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher. Printed in the United Kingdom.

ISBN 0-XXX-YYYYYY-1

Text printed on recycled and acid-free paper

1 2 3 4 5 6 7 8 9 0

First printing, MONTH YEAR

---

# *Table of Contents*

<i>Table of Contents</i>	v
<i>Preface</i>	xiii
<b>CHAPTER 1      <i>Introduction</i></b>	<b>I</b>
1.1    The touchstones: Lean and Agile	1
1.2    Lean Architecture	4
1.3    Agile Production	7
1.4    The Book in a Very Small Nutshell	8
1.5    Lean and Agile: Contrasting and Complementary	10
1.5.1    Architecture in the Value Stream	12
1.5.2    The Lean Secret	13
1.6    Lost Practices	17
1.6.1    Architecture	17
1.6.2    Requirements Dependency Management	18
1.6.3    Foundations for Usability	18
1.6.4    Documentation	18
1.6.5    Common sense, thinking, and caring	21

1.7	<b>What this book is not about</b>	23
1.8	<b>Agile, Lean— oh, yeah, and Scrum and methodologies and such</b>	24
<b>CHAPTER 2</b>	<i>Agile Production in a Nutshell</i>	27
2.1	<b>Engage the stakeholders</b>	28
2.2	<b>Define the problem</b>	30
2.3	<b>Focusing on what the system is: the foundations of form</b>	31
2.4	<b>Focusing on what the system does: the system lifeblood</b>	33
2.5	<b>Design and Code</b>	35
2.6	<b>Countdown: 3, 2, 1...</b>	36
<b>CHAPTER 3</b>	<i>Stakeholder Engagement</i>	37
3.1	<b>The key stakeholders</b>	40
3.1.1	End Users	43
	Psyching out the end users	44
	Don't forget behavior	46
	The End User landscape	47
3.1.2	The Business	48
	A special note for Managers	49
3.1.3	Customers	50
	... as contrasted with end users	50
	"Customers" in the value stream	53
3.1.4	Domain Experts	53
	No ivory tower architects	54
	Experts in both problem and solution domains	55
3.1.5	Developers and Testers	56
3.2	<b>Process elements of stakeholder engagement</b>	58
3.2.1	Getting started	59
3.2.2	Customer Engagement	61
3.2.3	Pipelining	62
3.3	<b>The Network of Stakeholders: Trimming Wasted Time</b>	64
3.3.1	Stovepipe versus Swarm	64
3.3.2	The first thing you build	68
3.3.3	Keep the Team Together	69
<b>CHAPTER 4</b>	<i>Problem Definition</i>	71

<b>4.1</b>	<b>What's Agile about problem definitions?</b>	<b>72</b>
<b>4.2</b>	<b>What's Lean about problem definitions?</b>	<b>73</b>
<b>4.3</b>	<b>Good problem definitions</b>	<b>74</b>
<b>4.4</b>	<b>Problems and solutions</b>	<b>77</b>
<b>4.5</b>	<b>The process around problem definitions</b>	<b>78</b>
4.5.1	Value the hunt over the prize	78
4.5.2	Problem Ownership	79
4.5.3	Creeping featurism	81
<b>4.6</b>	<b>Problem definitions, goals, charters, visions, and objectives</b>	<b>82</b>
<b>4.7</b>	<b>Documentation?</b>	<b>83</b>
<b>CHAPTER 5</b>	<b><i>What the System Is: Lean Architecture</i></b>	<b>85</b>
<b>5.1</b>	<b>Some Surprises about Architecture</b>	<b>87</b>
5.1.1	What's Lean about this?	88
The place of thought		89
Failure-proof constraints or “poka-yoke”		89
The Lean mantras of conservation, consistency and focus		90
5.1.2	What's Agile about architecture?	91
It's all about individuals and interactions		91
Past excesses		92
Dispelling a couple of Agile myths		93
<b>5.2</b>	<b>The first design step: Partitioning</b>	<b>95</b>
5.2.1	The first partition: domain form versus behavioral form	96
5.2.2	The second partitioning: Conway's Law	97
5.2.3	The real complexity of partitioning	100
5.2.4	Dimensions of complexity	101
5.2.5	Domains: a particularly interesting partitioning	102
5.2.6	Back to dimensions of complexity	104
5.2.7	Wrap-up on Conway's Law	108
<b>5.3</b>	<b>The second design step: selecting a design style</b>	<b>109</b>
5.3.1	Contrasting structuring with partitioning	110
5.3.2	The fundamentals of style: commonality and variation	112
5.3.3	Starting with tacit commonality and variation	114
5.3.4	Commonality, variation and scope	117
5.3.5	Making the commonalities and variations explicit	120
Next steps		124

5.3.6	The most common style: object orientation	124
	Just what is object orientation?	125
5.3.7	Other styles within the Von Neumann world	128
5.3.8	Domain-specific languages and application generators	131
	The state of the art in DSLs	132
	DSLs' place in architecture	133
5.3.9	Codified Forms: Pattern Languages	134
5.3.10	Third-party software and other paradigms	136
<b>5.4</b>	<b>The rough framing: Delivering the First Code</b>	<b>139</b>
5.4.1	Abstract Base Classes	140
5.4.2	Pre-conditions and post-conditions	144
5.4.3	Algorithmic scaling: the opposite of static assertions	147
5.4.4	Form versus accessible services	147
5.4.5	Scaffolding	147
5.4.6	Testing the architecture	148
	Usability testing	148
	Architecture testing	149
<b>5.5</b>	<b>Relationships in architecture</b>	<b>149</b>
<b>5.6</b>	<b>Not your grandfather's OO</b>	<b>149</b>
<b>5.7</b>	<b>How much architecture?</b>	<b>150</b>
<b>5.8</b>	<b>Documentation?</b>	<b>151</b>
5.8.1	The Domain Dictionary	151
5.8.2	Architecture Carryover	151
<b>5.9</b>	<b>History and Such</b>	<b>152</b>
<b>CHAPTER 6</b>	<b><i>What the System Does: Capturing the Service Needs</i></b>	<b>153</b>
<b>6.1</b>	<b>Users and Customers</b>	<b>154</b>
<b>6.2</b>	<b>Getting Started: User Stories</b>	<b>156</b>
6.2.1	The importance of the End User Motivation: Goal-driven Use Cases	157
<b>6.3</b>	<b>User Stories to Use Cases</b>	<b>157</b>
6.3.1	The Prelude	157
6.3.2	The Basic Flow	157
6.3.3	The Postlude	158
<b>6.4</b>	<b>Capturing Alternative Flows</b>	<b>158</b>
6.4.1	Packaging "Non-Functional" Requirements in Use Cases	159

<b>6.5</b>	<b>Managing Use Cases Incrementally</b>	<b>159</b>
<b>6.6</b>	<b>Use Cases to Roles</b>	<b>159</b>
6.6.1	Framing it Out	160
6.6.2	Language Support: Interfaces	160
6.6.3	Later: Use Case scenarios to algorithms	160
<b>6.7</b>	<b>Use Cases and Architecture</b>	<b>160</b>
6.7.1	Delimit project scope	160
6.7.2	How about interactions with the architecture?	160
<b>6.8</b>	<b>“It Depends”: When Use Cases are a bad fit</b>	<b>160</b>
6.8.1	Classic OO: Atomic Event Architectures	161
6.8.2	Business Rules	163
<b>6.9</b>	<b>Another Example: Interface to Bank Accounts</b>	<b>163</b>
<b>6.10</b>	<b>Documentation?</b>	<b>167</b>
<b>6.11</b>	<b>History and Such</b>	<b>168</b>
<b>CHAPTER 7</b>	<b><i>Coding It Up: Basic Assembly</i></b>	<b>169</b>
<b>7.1</b>	<b>The Big Picture: Model-View-Controller-User</b>	<b>170</b>
7.1.1	What is a program?	170
7.1.2	What is an Agile program?	172
7.1.3	MVC in more detail	174
7.1.4	MVC-U: Not the end of the story.	175
	A Short History of Computer Science	176
	Atomic Event Architectures	177
	DCI architectures	179
<b>7.2</b>	<b>The form and architecture of atomic event systems</b>	<b>179</b>
7.2.1	Domain Objects	180
7.2.2	Roles, Interfaces and the Model	180
Example		184
7.2.3	Reflection: Use Cases, atomic event architectures, and algorithms	185
7.2.4	A special case: One-to-many mapping of roles to objects	186
<b>7.3</b>	<b>Updating the Domain Logic: Method elaboration, factoring, and re-factoring</b>	<b>187</b>
7.3.1	Creating new classes and filling in existing function placeholders	188
Example		189
7.3.2	Back to the future: this is just good old-fashioned OO	190
7.3.3	Tools [CRC Cards]	190

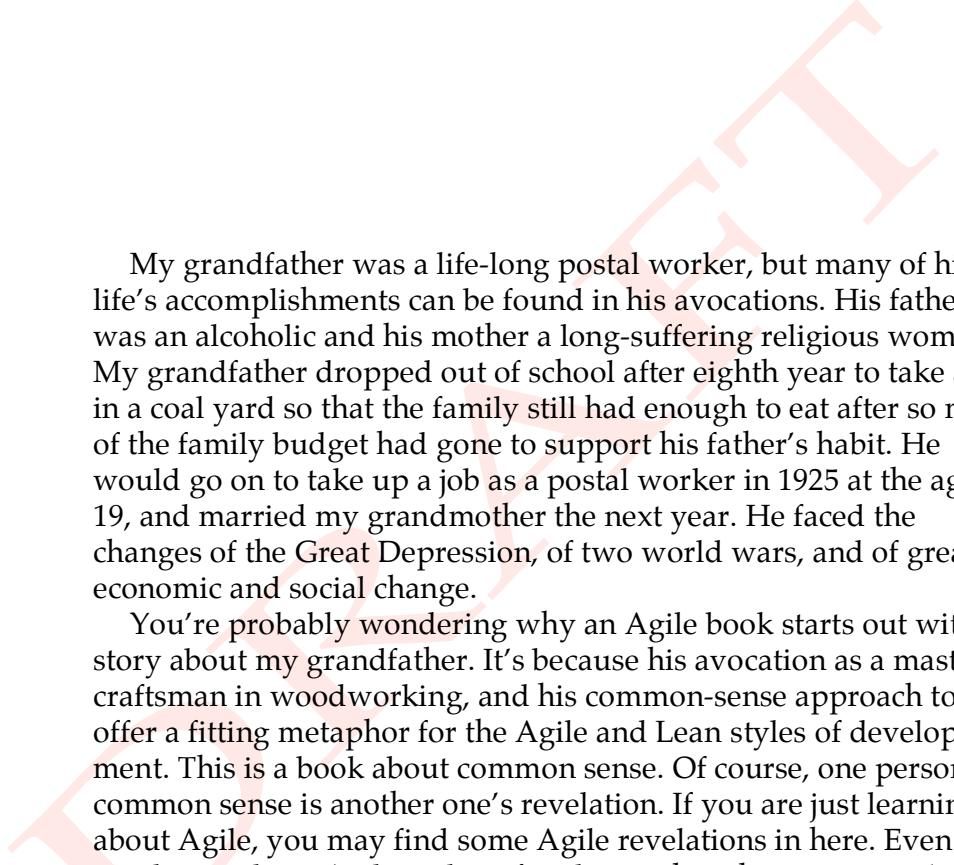
7.3.4	Factoring	190
7.3.5	A Caution about Re-Factoring	191
<b>7.4</b>	<b>Documentation?</b>	<b>192</b>
<b>7.5</b>	<b>Why all these artifacts?</b>	<b>192</b>
<b>CHAPTER 8</b>	<b><i>Coding it Up: the DCI Architecture</i></b>	<b>193</b>
<b>8.1</b>	<b>Sometimes, smart objects just aren't enough</b>	<b>193</b>
<b>8.2</b>	<b>Overview of DCI</b>	<b>194</b>
8.2.1	Some parts of the user mental model we've forgotten	195
8.2.2	Enter method-ful roles	197
8.2.3	Tricks with Traits	200
8.2.4	Context Classes: One per Use Case Scenario	201
<b>8.3</b>	<b>DCI by Example</b>	<b>204</b>
8.3.1	The Inputs to the Design	204
8.3.2	Use Case Scenarios to Algorithms	205
8.3.3	Methodless Roles: The Framework for Identifiers	210
8.3.4	Partitioning the algorithms across Method-ful Roles	211
	Traits as a Building Block	211
	Coding it up	214
8.3.5	The Context framework	216
	Making Contexts Work	221
	Nested Contexts in Method-ful Roles	231
8.3.6	Variants and Tricks on DCI	236
<b>8.4</b>	<b>Updating the Domain Logic</b>	<b>236</b>
8.4.1	Comparing and Contrasting DCI with the Atomic Event Style	237
8.4.2	The DCI approach: special considerations for domain logic	239
<b>8.5</b>	<b>Context Objects in the User Mental Model: Solution to an Age-old problem</b>	<b>241</b>
<b>8.6</b>	<b>Why all these artifacts?</b>	<b>242</b>
	Why not use classes instead of "method-ful roles"?	243
	Why not put the entire algorithm inside of the class with which it is most closely coupled?	244
	Then why not localize the algorithm to a class and tie it to domain objects as needed?	244
	Why not put the algorithm into a procedure, and combine the procedural paradigm with the object paradigm in a single program?	244
	So, what do DCI and lean architecture give me?	245
	And remember...	246

<b>8.7</b>	<b>Beyond C++: DCI in Other Languages</b>	<b>246</b>
8.7.1	Scala	246
8.7.2	Python	247
8.7.3	C#	248
8.7.4	... and even Java	248
8.7.5	The Original Stars and Circles Example in Smalltalk	249
	Deriving the Method-ful Roles	249
<b>8.8</b>	<b>Documentation?</b>	<b>251</b>
<b>8.9</b>	<b>History and Such</b>	<b>254</b>
8.9.1	DCI and Aspect-Oriented Programming	254
8.9.2	Context in the original DCI	254
8.9.3	Other approaches	255
<b>APPENDIX A</b>	<i>Scala Implementation of the DCI Account Example</i>	<b>257</b>
<b>APPENDIX B</b>	<i>Account example in Python</i>	<b>261</b>
<b>APPENDIX C</b>	<i>Account example in C#</i>	<b>265</b>
<b>APPENDIX D</b>	<i>Account example in Ruby</i>	<b>271</b>
<b>APPENDIX E</b>	<i>Java Demonstration of DCI</i>	<b>275</b>
<b>APPENDIX F</b>	<i>Qi4J</i>	<b>285</b>
<b>APPENDIX G</b>	<i>DCI, Aspects, and Multi-Methods</i>	<b>289</b>
<i>Bibliography</i>		<b>291</b>
<i>Leftovers</i>		<b>299</b>

DRAFT

---

# Preface



My 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. My grandfather dropped out of school after eighth year to take a job in a coal yard so that the family still had enough to eat after so much of the family budget had gone to support his father's habit. He would go on to take up a job as a postal worker in 1925 at the age of 19, and married my 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 out with a story about my grandfather. It's because his avocation as a master craftsman in woodworking, and 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, you may find some Agile revelations in here. Even if you know about Agile and are familiar with architecture, you're likely to learn a lot from this book about how they can work and play together.

As a postal employee, my grandfather 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 goal wasn't as much on business results and

profit as it was on quality and individual customer service. My grandfather 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, nor flooded creek valleys that would cut off an area. 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 metaphors between my grandfather'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 contacts and negotiation; such was not part of my grandfather's job, and it's not the job of a modern software engineer in an Agile setting. The focus falls on the end user. In my grandfather'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 being entrusted to my grandfather for dispatching to its destination, or the flurry of warm greetings around the Christmas holidays. The business entity in the middle—in my grandfather'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 Use Cases as a means for profit; end users have a stake in them to ensure some measure of day-to-day comfort in supporting their workflow.

That isn't 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 my grandfather 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.

Because things are simple doesn't mean they are simplistic. The modern philosopher Thomas Moore asks us to "live simply, but be complicated" [Moo2001, 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 same 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.

My grandfather 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 that we have found broadly to work over 15 years of close work 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 [Bec+2001]) is largely about doing, and the Lean concepts from the Toyota tradition relate more to planning and thinking [Lik2004, ff. 237]. These notions of thinking are among the lost practices of Agile—lost in the eager Agile focus on product that was too easily lost in the methodology-polluted age of the 1980s.

My grandfather's life is also a reminder that we should value timeless domain knowledge. XP started out in part by consciously trying to do exactly the opposite of what conventional wisdom recommended, and in part out of ignorance of the practices suitable to large-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 ([Beck1999], p. 113; [Beck2005], p. 28). We're starting to recover the Lost Arks and Golden Chalices of past ages—relics whose value we didn't appreciate at the time. Without

pretense of a miracle (admittedly a risk given this choice of metaphors) we look a little bit to the past to rediscover the treasures there. More often than not, we find most of the Emperors' new clothes are cut from old cloth.

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 have also drawn on our own experience in software development going back to our first hobby programs in the 1960s, and lives in the software profession going back to the early 1970s. We have built on Jim's more recent book, *Organizational Patterns of Agile Software Development* [CopHar2004], 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 Scrum meetings [Sut2003, Sut2007] and of much of the structural component of XP [Fra+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 programming language has rarely been found to matter in any broadly applicable way. As we know from Agile and Lean thinking, most things that matter involve people and values, and come to terms 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 Jim 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 my grandfather 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*, [Bra1999] which is inspired in no small part by software greats including ??? and ??? The manuscript is sprinkled with several small outtakes for the book. 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 very brief interludes that should 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.

Thanks to the many reviewers who scoured the manuscript and helped us to polish it: Dave Byers, Urvashi Kaul, Steen Lehmann, Simon Michael, Roy Ben Hayun, Lena Nikolaev... Many special thanks to Lars Fogtmann Sønderskov for a thorough review of an early version of the manuscript. His deep expertise in Lean thinking challenged our own thinking and pushed us to review and re-think some of the emphases in the book. Brett Schuchert, who was one of my most treasured reviewers for *Advanced C++* 20 years ago, again treated us to a tough scouring of the manuscript. Thanks, Brett! Christian Horsdal Gammelgaard did the pioneering work on DCI in .Net/C#. 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, Tomi Tuominen, and Ari Tikka all contributed mightily.

We'd like to appreciate many who have gone before us and who have had an influence on the way we look at the world and how we keep learning about it. Phillip Fuhrer gave me useful insights on problem definition. I had some useful mail conversations with Larry Constantine, and it was a pleasure to again interact with him and gain some 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. Trygve Reenskaug, Jeff Sutherland, Alistair Cockburn, Jerry Weinberg,... And, of course, my grandfather.

Some assorted notes go here for now  
Preface material

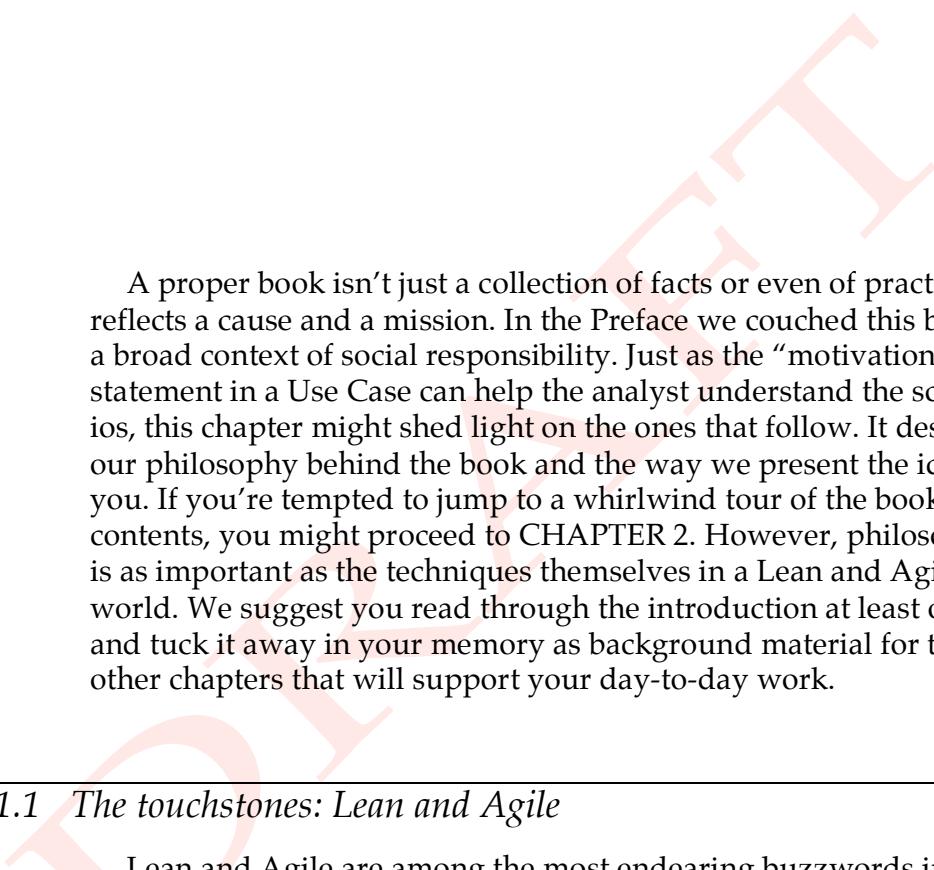
Lean instead of Agile. Why? Because it's the right word.  
What's new here? Not much, except the Trygve stuff.  
Holistic: requirements, architecture, and organization

Other stuff:

A picture per every three pages as a rule of thumb  
Every chapter should end in a summary and a history  
Include org patterns  
Focus on direct manipulation metaphor, original OO  
No XP-bashing.  
Running example.



# *Introduction*



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" statement in a Use Case can help the analyst understand the 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 [Wo+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 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. 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 honor the practice of continuous flow. Each of these 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 re-work 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.

Architecture is *hard*. The need for so many different talents working together is one of the main reasons architecture is hard: dealing with the basic form of a system's essential complexity. Architecture is also difficult because it requires balance: no silver bullet alone targets the answer. In this same vein, neither Lean nor Agile alone make it easy. But both clarify the target through thought and feedback. That clarity is a touchstone of effectiveness, in what it can do to reduce waste, inconsistency, and irregular development.

Another key Lean principle is to focus on long-term results [Lik2004, 71—84]. Lean architecture is about being 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 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 [Gla2006] found that 70% of the software they build is never used [Standish]. If your customer can't read UML, where do comprehensive UML diagrams fit in the value stream?

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-adopt 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*

A simplistic view of Agile is that it is an approach to software development that removes the heavyweight techniques that accrued during the 1980s. Software architecture kind of lost its way during the 1980s as methodologies and CASE tools put a stranglehold on software development cultures. Agile software development, which is rooted partly in Lean principles, strives to recover from that era. The Agile Manifesto [Be+2001] defines the principles that underlie the Agile vision, and the Toyota Way [Lik2004] defines the Lean vision. This book defines a vision of what architecture looks like in an organization that has embraced these two sets of ideals. How does that vision differ from the heavyweight architectural practices that dominated object-oriented development in the 1980s? We summarize the differences in Figure 1.

- 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 taking its form, we let form follow proven experience instead of being driven by supposedly scientific engineering rationales. Those will come soon enough.

<i>Lean Architecture</i>	<i>Classic Architecture</i>
Defers engineering	Includes engineering
Gives the craftsman “wiggle room” for change	Tries to limit large changes as “dangerous” (fear change?)
Defers implementation (manages API and relationships)	Includes much implementation (platforms, libraries)
Code-focused (lightweight documentation)	Documentation-focused: heavy and early
People	Tools and notations
Tease out the changing parts	Difficult-to-change monolith
Do what’s important now	Do what’s important in the long term
End user mental model	Technical coupling and cohesion

**Figure 1: What is Lean Architecture?**

- This in turn implies that the everyday software artisan has some license to tailor form according to experience. Neither Agile nor lean gives wholesale license to ravage the system form, but both honor the value of adaptation. Classic architecture tends to be fearful that big change might happen blocks off paths that could lead to dangerous change. In our combined Lean/Agile approach, the domain architecture captures form that is important to success but which needn’t be part of the minute-to-minute focus on end user value. This is another argument for a true architecture of form rather than a governance of structure.
- Classic software architecture is sometimes overly eager to embrace implementation early on in the interest of code reuse or standards. Lean architecture also adopts the perspective that standards are valuable, but again: at the level of form, protocols, and APIs, rather than the implementation of those forms, protocols, and APIs.

- Some classic approaches to software architecture too often depended on, or at least produce, volumes of documentation at high cost. Agile believes in communication, and we will talk about documentation as a sometimes-viable communication mechanism. However, we will document only the stuff that really matters, and we'll communicate some of our decisions in code. That kills two birds with one stone.
- 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 the mental models they carry with them.
- Classic architectures too often focus on building something that could be done once and tucked away. Agile understands that nothing lasts forever, and so we'll be focusing explicitly on surfacing the parts that change.
- Classic architecture sometimes gets so caught up in long-term planning that tomorrow's needs are pushed aside. Worse than the plan is the prescription to follow the plan. The largest problems aren't in planning, but in following the plan blindly. Agile focuses on what's important now—whether that be hitting the target for next week's delivery or doing long-term planning. 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.

As we describe it in this book, Lean architecture provides a firm foundation for the ongoing business of a software enterprise: providing 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. [Bec+2001]

Just as with the “all hands on deck” approach of Lean, Agile development also embraces close person-to-person contact, particularly with the customer. Unlike the tendencies of Lean, and certainly the tendencies of classic architecture, our vision of Agile production embraces 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 customer whims. And we love customer whims—because that’s how we provide satisfaction and make money.

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 may also need other design representations that close the feedback loop to the end-user,

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.

The fulfillment of this dream has long eluded the object-oriented programming community, but the recent work on the DCI architecture, featured in CHAPTER 8, brings this dream much closer to reality than we have ever realized. If we can directly capture key end-user mental models in the code, it radically increases the chances the code will work. 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 wants it to do. 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 basic 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 capture these models in code as early as possible. Coupling and cohesion [StMeCo1974] follow these principles as a secondary effect.
- To explore both form and function requires up-front engagement of all stakeholders' insights. Deferring interactions with stakeholders, or deferring decisions beyond the last responsible moment, slows things down, raises cost, and increases frustration. A team acts like a team from the start.

- We can express system form as abstract base classes, which are a concrete expression of a domain model. The team can build such a model in about one Scrum sprint: a couple of weeks to a month. Design-by-contract 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 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. By making dependencies between requirements explicit, Use Cases avoid dependency management and communication problems that are common in complex Agile projects.
- 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 classes. 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 stupid, 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.

<i>Lean</i>	<i>Agile</i>
Thinking and doing	Doing
Feed-forward and feedback	Feedback
High throughput	Low latency
Planning	Reaction
Focus on Process	Focus on People
Complicated systems	Complex systems
Embrace standards	Inspect and adapt
Rework in manufacturing adds value, in manufacturing is waste	Minimize up-front work of any kind and use rework of code to get quality
Bring decisions forward	Defer decisions

---

**Figure 2: Contrast between Lean and Agile**

[TODO: Lars suggests a figure here that is an overview of the book's landskab]

---

## 1.5 *Lean and Agile: Contrasting and Complementary*

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

One unsung strength of Agile is that it seems to be more about the long-term sustenance of a project than just its beginnings. We get the feeling from the waterfall model that we start from scratch every time: requirements are an input to a waterfall project, but information about existing software doesn't fit the model well. Most architecture notations find a home for the initial system vision, but notational artifacts too often fall into obscurity over time. If we constantly refresh the architecture in cyclic development, and if we express the architecture in living code, we'll avoid exactly the kinds of problems that Agile was created to address. Yes, we'll talk about ar-

---

chitectural beginnings, but the right way to view software development is to treat everything beyond the first iteration as 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. Scrum co-founder Jeff Sutherland refers to Lean and Agile as separate and complementary developments that both arose from observations about complex adaptive systems [Sut2008a]. Indeed, in some places Lean principles and Agile principles tug in different directions. The Toyota Way is based explicitly on standardization [Lik2004, 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 [Lik2004, chapter 19]; most Agile practice is based on rapid *decisions*.

Some of the disconnect between Agile and Lean comes not from their foundations but from common misunderstanding and practice in the real world. 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. [Lik2004, 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. [Øst2008]

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 [Bal2000] points out that a little rework in design and thought adds value by reducing interval and cost, while rework during production is waste (Section 1.5.1). System-level-factoring entails a bit of both, but viewing 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. [SnBo2007] Agile is the art of the possible, and of expecting the unexpected.

This book is about having a Lean architecture that makes Agile development possible. 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 reduce work and rework. As a stitch in time saves nine, so up-front thinking can empower decision makers to implement decisions lightening-fast with confidence and authority. Agile architecture should be rooted in the thought 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 Architecture in the Value Stream

[TODO: Is this the right place for this section?]

The main goal of Lean is to create a value stream, every one of whose activities adds value to the end user. Architecture too often is viewed as an awkward cost rather than as something that provides end-user value. Where does architecture fit in the value stream?

---

Again, an interplay of Lean and Agile provide powerful ways of adding end-user value:

- *Time*: A good architecture shortens the time between understanding user needs and delivering a solution. It does this by eliminating rework and by providing a de-facto standard framework based on the end user domain model.
- *Function*: Unlike traditional architecture which deals almost exclusively with domain structure, the Agile production side of this book embraces the deep form of function and gives it first class standing in the architecture. This focus improves the likelihood that we will deliver what the customer expects.
- *Cost*: Lean results in cost savings that can be passed on to the user. These cost savings come from reducing the distance between the end user world model and the code structure and from reduced waste, including just-in-time feature production.

To make this work requires a carefully coordinated and synchronized team, with the entire team focused on end user value and on continuously improving that value. That's the Lean Secret.

### 1.5.2 The Lean Secret

The bottom line of Lean comes down to this rule of thumb:

Everybody, All at Once, Early On

Using other words, we also call this "all hands on deck." Why is this a "secret"? Because it seems that, within the realm of Agile development it is either wholly unknown or embraced only in part, lacking any perspective that these three considerations work together to support the most central purposes of Lean.

The roots of Lean go back to just-in-time delivery concepts developed by Kiichiro Toyoda in Toyota's predecessor company in the early twentieth century. The system became defunct after the war,

and was revamped by an engineer in charge of final car assembly in Toyota at the time, Taiichi Ohno. His key tenets of Lean are based on reduced waste, on consistency and order, and on maintaining a smooth process flow without bunching or bumps. [Lik2004, Chapter 2] Honda operated the same way [TakNon1986, p.6]:

Like a rugby team, the core project members at Honda stay intact from beginning to end and are responsible for combining all of the phases.

The first step is to avoid waste: of taking care to make the best use of materials and to not produce anything unless it adds value to the customer. The second step is to reduce inconsistencies by making sure that everything fits together and that everyone is on the same page. The third step is to smooth out flow in the process, reducing internal inventory and wait states in the process.

The Agile world embodies Lean principles in different measures. Too many Agile rules of thumb have roots in a naive cause-and-effect approach to Lean principles. To reduce waste, reduce what you build; to reduce what you build, remove the thing furthest from the customer: the internal documentation. To reduce inconsistencies, defer any decision that could establish an inconsistency, and introduce tight feedback loops to keep any single thread of progress from wandering too far from the shared path. To smooth out flow, keep the work units small.

While some of these practices (such as small work units) are squarely in the center of Lean practice, others are overly brute-force attempts that can benefit from exactly the kind of systems thinking that Lean offers. Ballard [Bal2000] has developed a model of Lean production based on a few simple concepts:

- Rework in production creates waste. However, design without rework is wasted time. If design rework is a consequence of a newly learned requirement, it saves even more expensive re-work that would arise in production. Rework in design creates value.

- You can increase the benefits of such rework in design, and avoid rework in production, by engaging all stakeholders together (a cross-functional team) as early as possible.
- Make decisions at the responsible moment—that moment after which the foregone decision has a bearing on how subsequent decisions are made, or would have been made. Note that “[k]nowledge of the lead times required for realizing design alternatives is necessary in order to determine last responsible moments.” Therefore, line up dependencies early to optimize the decision-making process: front-load the process with design planning. There is rarely any difference between the first responsible moment and the last responsible moment: it’s just *the responsible moment*. Maybe rework is caused by work done before the responsible moment.<sup>1</sup>

In addition to these principles, Ballard agrees that batch sizes should be small, and that production should use feedback loops carrying even incomplete information.

This combination of teamwork, iteration, and up-front planning is part of Lean’s secret. There are aspects of this secret that are too often lost in the Agile world. Agile often fails to value up-front design or celebrate the rework that happens there. Too many Agilists confuse the sin of following a plan with planning itself. Scrum features a bit of up-front design sequencing at the beginning of each sprint, when the team comes together to tentatively lay out the tasks for the sprint. However, few Agile practices bring mature domain design knowledge forward in a way that reduces the cost and effort of later design decisions, or that simply makes them unnecessary.

Much of the hard part of design is the partitioning of a system into modules and collections of related APIs: that is exactly the problem that up-front architecture addresses. While popular Agile culture holds out hope that such partitioning and groupings can easily be re-factored into shape, few contemporary re-factoring approaches rise to the occasion. Most re-factoring approaches are designed to

---

<sup>1</sup> Thanks to Lars Fogtmann Sønderskov.

work within one of these partitioned subsystems or within a class hierarchy that reflects such a grouping of APIs: they don't deal with the more complex problem of moving an API from one neighborhood to another. That's hard work.

Architecture doesn't eliminate this hard work but can greatly reduce the need for it. When the world shifts and technology, law or business cause wrinkles in a domain, you either have to start over (which isn't always a bad idea) or do the hard work of restructuring the software. And it *is* hard work. *Agile* doesn't mean *easy*, and it offers no promises: only a set of values that focused, dedicated team members can employ as they face a new situation every day. Architecture is hard work, too, and requires foresight (obviously), courage, and commitment—commitment that is too much for timid developers. But a Lean architecture can have sweet payoff in the long term.

That's the technical side. There is also an organizational side to Lean and Agile, which usually dominates the industry buzz around the buzzwords. Lean strives to eliminate waste, which in the Toyota Way is called *mura* in Japanese. There are many kinds of waste, including late discovery of mistakes, overproduction, and waiting. If one person has to wait, and is idle until a supplier delivers needed information or materials, it is a waste of an opportunity to move things quickly along the value stream.

These ideas are hardly new. Everybody, all at once, early on is a time-honored technique. One early (1970s), broadly practiced instance of this idea is Joint Application Design, or JAD [Dav1999]. JAD was a bit heavyweight, since it involved all the stakeholders from the top management and the clients, to the seminar secretary, for weeks of meetings. While it was a bit heavyweight and probably didn't admit enough about emergent requirements (its goal was a specification), its concept of broad stakeholder engagement is worth embracing. Letting everybody be heard, even during design, is a lost practice of great software design. But there are even more lost practices worth reviving in a vibrant, Agile context.

## 1.6 Lost Practices

Both of us had 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, Agile, Lean and object communities started interacting more, 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.

As Agile rolled out into the industry—often through the channels of consultants and trainers who were the product of the massive Scrum certification program—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 practice of XP and Scrum. These include system and software architecture, requirements dependency management, foundations for usability, documentation, and others.

### 1.6.1 Architecture

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 [TODO: Jeff cite]. We build for change.

We know that ignoring architecture in the long term increases long-term cost. Traditional architecture is 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 Requirements Dependency Management

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.

### 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 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

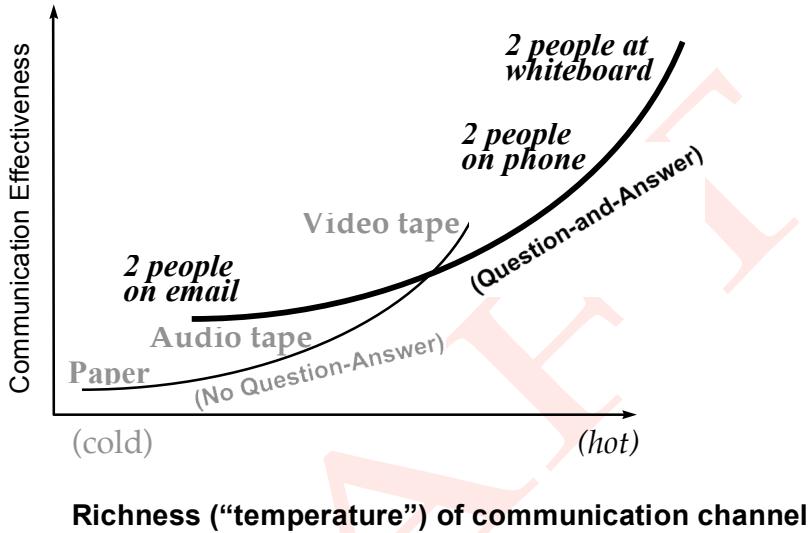
Documentation gets a bad rap. Methodologists too often miss the point that documentation has two important functions: to *communicate* perspectives and decisions, and to *remember* perspectives and

---

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 [Coc2007, pp. 23 - 24]. Much of the Agile mindset misses this dichotomy and casts aspersions on any kind of documentation. However, 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.

However, Agile has a weakness that begs for supplementary support from outside the principles of the Agile Manifesto. 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 also 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 3).



**Figure 3: Forms of Communication Documentation**

*From Alistair Cockburn, "Agile Software Development: The Cooperative Game," 2nd ed Addison Wesley 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 looks how we used heavyweight notations in the 1980s

---

and replaces it with well-considered code. There in fact isn't much new or Agile in this: such was also the spirit of literate programming. Lean architecture also accords a place for documentation not so much for communication as for team memory. In some settings, such as geographically distributed development, written documentation may play a more crucial role than in a geographically collocated team. Numerous advices to this end can be found from other Agile advocates [CITE FOWLER].

In general, “the code is the design” is a good rule of thumb. But it is neither a law nor a principle. Much of the same crowd that we today found advocating Agile principles came to Agile through the pattern discipline. Patterns were created out of an understanding that code sometimes does not stand alone. Even in the widely accepted Gang of Four book, we find that “it's clear that code won't reveal everything about how a system will work.” [Ga+2005, p.23] We go to the code for *what* and *how*, but only the authors or their documentation tell us *why*. As David Byers quips, the *why* of software is an important memory that deserves to be preserved.

### 1.6.5 Common sense, thinking, and caring

Finally, this book is about simple things: code, common sense, thinking, and caring. Code is the ever-present artifact at the core of the Agile development team. Properly done, it is an effigy of the end user conceptual model that constantly reminds the programmer of end user needs and dreams, even when the customer isn't around. In the end, it all comes down to code, and that's because code is the vehicle that brings quality of life to end users.

Common sense hides deeply within ourselves. Thinking and caring are equally simple concepts, though they require the effort of human will to carry out. Will takes courage and discipline, and that makes simple things look hard. That in turns implies that simple things aren't simplistic. As we said in the Preface, we try to find the fewest simple things that together can solve complex problems.

As the intelligent designer in the middle, we sometimes must wrestle with the entire spectrum of complexity. But we should all the

while strive to deliver a product that is pure. It's like good cooking: a good cook combines a few pure, high quality ingredients into a dish with rich and complex flavor. The chef combines the dishes with carefully chosen wines in a menu du jour with tastes and ingredients that are consistent, that complement each other. That's a lot better than trying to balance dozens of ingredients to achieve something even palatable, or of throwing together ingredients that are just good enough. Such food is enough for survival, but we can reach beyond surviving to thriving.

Like a cook, a programmer applies lean, critical thinking. Keep the set of ingredients small. Plan so you at least know what ingredients you'll need for the meals you envision. That doesn't necessarily mean shopping for all the ingredients far in advance; in fact, you end up delivering stale ingredients if you do that. Software is the same way, and Lean thinking in particular focuses on designing both the meal and the process to avoid waste. It places us in a proactive posture, a posture from which we can better be reactive when the need arises. Much of this book is about techniques that help you manage the overall form—the culinary menus, if you will—so you can create software that offers the services that your end users expect from you. It's about lining things up at just the right time to eliminate waste, to reduce fallow inventory, and to sustain the system perspectives that keep your work consistent.

Last, we do all of this with an attitude of caring, caring about the human being at the other end. Most of you will be thinking “customer” after reading that provocation. Yes, we care about our customers and accord them their proper place. We may think about end users even more. Agile depends on trust. True trust is reciprocal, and we expect the same respect and sense of satisfaction on the part of developers as on the part of end users and customers. Lest we forget, we care even about those nasty old managers. We think of a team that extends beyond the Scrum team with a community of trust that includes all of these folks.

That trust in hand, we'll be able to put powerful tools in place. The Lean Secret is the foundation: everybody, all at once, early on. Having such a proverbial round table lightens the load on heavy-

---

weight written communication and formal decision processes. That's where productivity and team velocity come from. That's how we reduce misunderstandings that lead to what are commonly called "requirements failures." That's how we embrace change when it happens. Many of these tools have been lost in the Agile rush to *do*. We want to restore more of a Lean perspective of *think and do*.

---

## 1.7 *What this book is not about*

This is not a design method. Agile software development shouldn't get caught in the trap of offering recipes. We as authors can't presume upon your way of working. We would find it strange if the method your company is using made it difficult to adopt any of the ideas of this book; it's those ideas we consider important, not the processes in which they are embedded.

While we pay attention to the current industry mindshare in certain fad areas, it is a matter of discussing how the fundamentals fit the fads rather than deriving our practices with the fads. For example, we believe that documentation is usually important, though the amount of documentation suitable to a project depends on its size, longevity, and distribution. This brings us around to the current business imperatives behind multi-site development, which tend to require more support from written media than in a geographically collocated project. We address the documentation problem by shifting from high-overhead artifacts such as UML to zero-overhead documentation such as APIs that become part of the deliverable, or through enough low-overhead artifacts to fit needs for supplemental team memory and communication.

The book also talks a lot about the need to view software as a way to deliver a service, and the fact that it is a product is only a means to that end. The word "service" appears frequently in the book. It is by coincidence only that the same word figures prominently in SOA, and we're making no conscious attempt to make this a SOA-friendly book, and we don't claim to represent the SOA perspective on what

constitutes a service. If you're a SOA person, what we can say is: if the shoe fits, wear it. We have no problem with happy coincidences.

We don't talk about some of the thorny architectural issues in this book such as concurrency, distribution, and security. We know they're important, but we feel there are no universal answers that we can recommend with confidence. The spectra of solutions in these areas are the topics of whole books and libraries relevant to each. Most of the advice you'll find there won't contradict anything that we say here.

The same is true for performance. We avoid the performance issue in part because of Knuth's Law: Premature optimization is the root of all evil. Most performance issues are best addressed by applying Pareto's law of economics to software: 80% of the stuff happens in 20% of the places. Find the hot spots and tune. The other reason we don't go into the art of real-time performance is partly because so much of the art is unteachable, and partly because it depends a great deal on specific domain knowledge. Volumes about on performance-tuning databases, and there are decades of real-time systems knowledge waiting to be mined. That's another book. The book by Noble and Weir [NoWe2000].

Though we are concerned with the programmer's role in producing usable, habitable, humane software, the book doesn't focus explicitly on interaction design or screen design. There are plenty of good resources on that [what ones to choose here?] We instead focus on the architectural issues that support good end-user conceptualization: these are crucial issues of software and system design.

---

### 1.8 *Agile, Lean—oh, yeah, and Scrum and methodologies and such*

If any buzzwords loom even larger than Agile on the Agile landscape itself, it is Scrum and XP. We figured that we'd lose credibility with you if we didn't say something wise about them. This section is our contribution to that need.

This book is about a lean approach to architecture, and about using that approach to support the Agile principles. Our inspirations for lean come through many paths, including Scrum, but all of them trace back to basics of the Lean philosophies that emerged in Japanese industry over the past century [Lik2004]: just-in-time, people and teamwork, continuous improvement, reduction of waste, and continuous built-in quality. We drive deeper than the techno-pop culture use of the term lean that focuses on the technique alone, but we show the path to the kind of human engagement that could, and should, excite and drive your team.

When we said that this book would build on a Lean approach to architecture to support Agile principles, most of you would have thought that by Agile we meant “fast” or maybe “flexible.” Agile is a buzzword that has taken on a life of its own, but even speed and flexibility reflect a bit of its core meaning. When you see the term “Agile,” in this book, take it that we mean the word in the sense of the Agile Manifesto [Bec+2001].

Scrum is an Agile framework for the management side of development. Its focus is to optimize return on investment by always producing the most important things first, by reducing rework through short cycles and improved human communication between stakeholders, by the elimination of wait states through self-organization, and by a “balance of power” in development roles that supports the developers with the business information they need to get their job done while working to remove impediments that block their progress.

This is not a Scrum book, and you probably don’t need Scrum to make sense of the material in this book nor to apply all or part of this book to your project. Because the techniques in this book derive from the Agile values, and because Scrum practices have the same foundations, the two complement each other well.

In theory, Scrum is agnostic with respect to the kind of business that uses it, and pretends to know nothing about software development. However, most interest in Scrum today comes from software development organizations. This book captures key practices such as

software architecture and requirements-driven testing that are crucial to the success of software Scrum projects. [CoSu2009]

Agile approaches, including Scrum, are based on three basic principles:

1. Trust
2. Communication
3. Self-organization

Each of these values has its place throughout requirements acquisition and architecture. While these values tend to touch concerns we commonly associate with management, and architecture touches concerns we commonly associate with programmers, there are huge gray areas in between. These areas include customer engagement and problem definition. We take up those issues, respectively, in CHAPTER 3 and CHAPTER 4. Those in hand, we'll be ready to move toward code that captures both what the system is and what the system does. But first, we'll give you a whirlwind tour of the book in CHAPTER 2.

# *Agile Production in a Nutshell*

This is the big-picture chapter, the get-started-quick chapter. This is for those readers who make it only through the first pages of most books they pick up in spite of best intentions to struggle through to the end. We got you this far. Hold on for ten pages as we describe the basic five activities of Agile Architecture.

These activities are neither waterfall phases nor steps; however, each one provides a focus for what a team member is doing at any given time. The ensuing chapters are:

- CHAPTER 3: Stakeholder Engagement
- CHAPTER 4: Problem Definition
- CHAPTER 5: System Architecture
- CHAPTER 6: Capturing the Service Needs
- Coding it up: CHAPTER 7: Basic assembly, and CHAPTER 8: The DCI Architecture

Here's the skinny on what's to come.

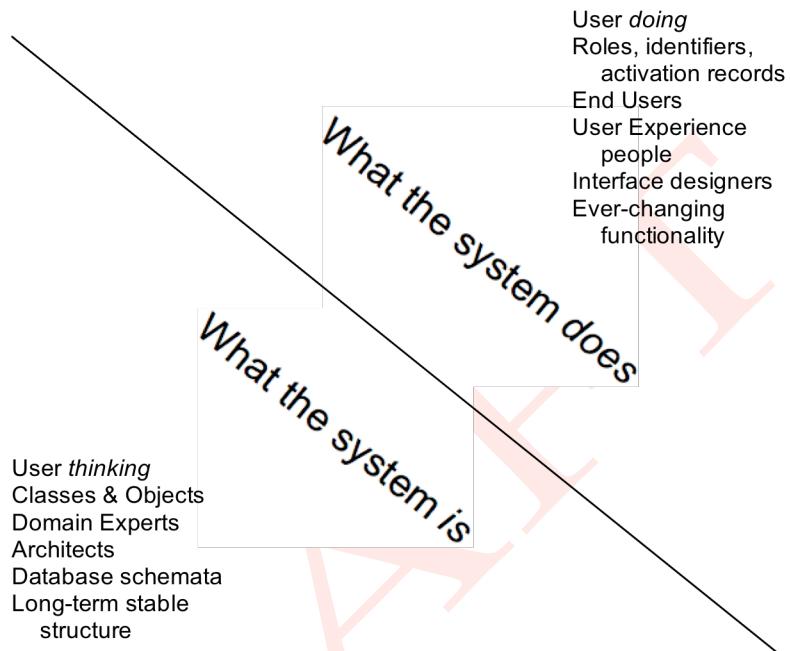
## 2.1 Engage the stakeholders

What is a “stakeholder”? Team members, system engineers, architects, and testers all have a stake in creating sound system form.

There may be many more: use your imagination. People appreciate being invited to the party early. And remember the Lean Secret: *everybody, all at once, early on.*

Identify the people and systems that care that this system even exists, what it does, or how it does it. Remember that you are building a system. A system (Weinberg again) is a collection of parts, none of which has meaningful existence apart from the others. When you are building a system, you need a system view—and in the system view, everything matters.

In the following chapters, we’ll often divide the system roughly in two. As shown in Figure 4, one part is what the system *is*; the other part is what the system *does*. (We will use variations of this figure several times throughout the book to illustrate the lean architecture principles). The what-the-system-*is* part relates to what is commonly called the architecture or platform: the part that reflects the stable structures of the business over time. Our key stakeholders for that part of the system are domain experts, system architects, and the wise gray-haired people of the business. The what-the-system-*does* part relates to the end user’s view of the services provided by the system: the tasks the system carries out for users and the way those tasks are structured. The end user, user experience folk, interface designers, and requirements folks are the key stakeholders in this part of the system. Of course, this system dichotomy isn’t black and white, and even if it were, most people have insights that can inform both of these areas.



**Figure 4: What the system is, and what the system does**

While we must acknowledge emergence in design and system development, a little planning can avoid much waste. My grandfather used to quote George Bernard Shaw in saying that good fences make good neighbors—and my grandfather knew that Shaw was cynical in that observation, much preferring that there be no fences at all.

By all means, don't fence off your customer: Customer engagement is one of the strongest of Agile principles. But it doesn't mean dragging the customer or end user into your workplace and interrogating them, or keeping them on watch to clarify important perceptions on a moment's notice. It means getting to know them. As a rural postal carrier, my grandfather got to know his clients over time by being attentive to the upkeep of their farms and by noticing the

kind of mail they received. He would recall Yogi Berra's quip: You can observe a lot just by watching. That's how it should be with customers: to understand their world-model and to reflect it in our design. In fact, this goes to the original foundations of object-oriented programming: the goal of Simula was to reflect the end user world model in the system design. That key principle has been lost in decades of methodology and programming language obfuscation, and we aim to help you restore it here.

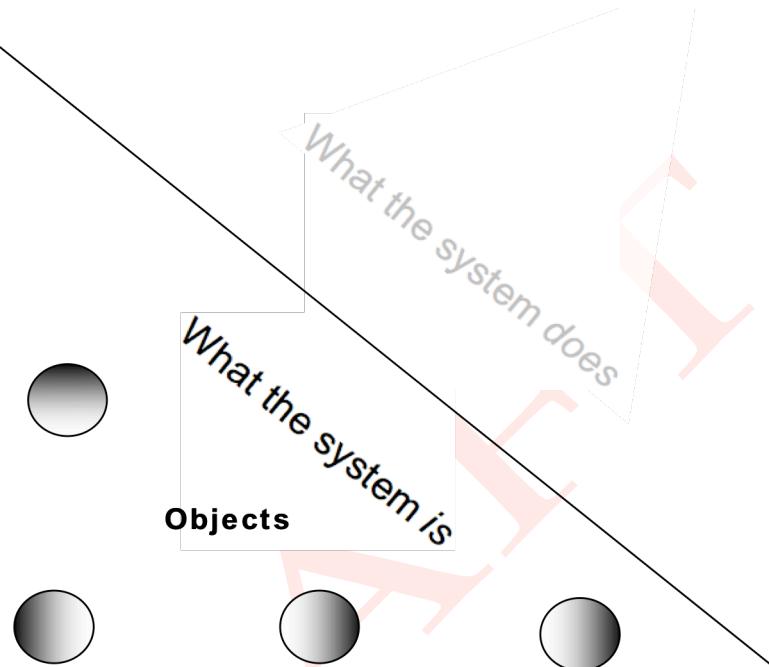
---

## 2.2 Define the problem

My grandfather used to say that if you didn't know where you were going, any road would get you there. Get the group together to write a short, crisp problem definition. I like Jerry Weinberg's definition of *problem*: the difference between the current state and a desired state. Write down this compass heading in a sentence or two. If the group collectively owns a notion of what problem they are solving, then they can own the notion of *done* when you're done. So we start with a one- or two-sentence problem definition.

Do this early in the project. A problem definition is a better compass than the first release's feature set, or the map of a domain which is yet unexplored. Don't wait until you've engaged every last stakeholder: great projects start with a visionary and vision rather than a market analysis. The important market analysis will come soon enough. Again, remember: *everybody, all at once, early on*.

Without a problem definition, it's hard to know when you're done. Sure, you can tick off some list of tasks that you may actually have completed and which were initially designed to bridge the gap between the current and desired state, but that doesn't mean that you're done. Emergent requirements cause the landscape to shift during development, and even the best-planned path may not lead you to the best-conceived destination.



**Figure 5: Focusing on what the system is: its form**

For the same reasons, be sure to check both your destination and your current compass heading in mid-journey. Revisit your problem definition once in a while to make sure it's current.

### 2.3 Focusing on what the system is: the foundations of form

Every system has two designs: the design of its functionality: what it *does*, and the design of its form: what it *is*. At the beginning of a project you need to focus on both. This double-edged focus applies not only to good beginnings but is at the heart of long-term product

health: the form, to establish a firm foundation for change, and the functionality to support end-user services.

My grandfather was an ardent woodsman, and we'd sometimes find ourselves miles from nowhere in the wilderness. He used to say: Trust the terrain, not the map. The terrain is the part that some methodologies call architecture. We can call it lean architecture. It is what the system *is*—as contrasted with what the system *does*, which we'll talk about later.

We'll introduce an architectural strategy that lays a foundation of abstract base classes early in the project, to establish the basic *form* the system will take on over its lifetime. The *structure* of the system will follow the form. It's not that form follows function, but function and form weave together into a real structure that we call member data and methods. And when we do come to structure, we'll focus on the *objects* rather than classes (Figure 5). Classes are more of a nerd thing; objects relate to the end user and to the business. The initial base classes we'll put in place are placeholders for objects of many different classes—the fact that they are abstract classes distances us from the class specifics. Yes, we'll come to classes soon enough, and there's some cool stuff there. But objects should dominate the design thought process.

The end user mental model is one sound foundation for architecture, and domain knowledge helps us think critically and soberly about user models in light of longstanding experience. Think of domain expertise as a broadening of user expertise into the realms of all stakeholders taken together. Expert developers learn over time what the fundamental building blocks of a given system should be.

Of course, in real development we work on the problem definition, architecture, and use cases in parallel. From a conceptual perspective, architecture and its articulation give us vocabulary and solid foundations for Use Case implementation later on. So, guess what: *everybody, all at once, early on* rules again. So why advocate an up-front focus on architecture? Is that Agile? Yes: even XP admits the need for architecture [Bec1999, p. 113]. We do architecture:

1. To capture stakeholders' perspectives that affect design;

- 
2. To embrace change, reduce cost of solving problems;
  3. To create a shared vision across the team and the stakeholders;
  4. To smooth the decision-making process.

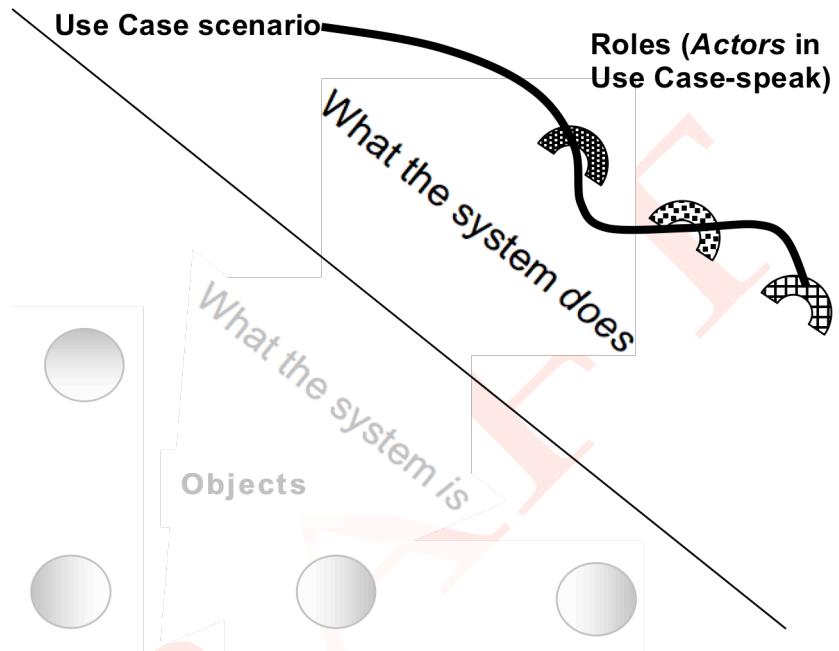
Software architecture disciplines from the 1980s delivered a truck-load of artifacts that preceded the first line of code—artifacts never seen by the end user. That isn’t lean. Lean architecture very carefully slices the system to express the essence of system form in source code. While the interface is the program, the code is the design—and architecture helps us see it from a perspective that’s often invisible in the source code. We deliver a thin shell of declarative, compilable code: domain class interfaces as source code contracts, boilerplate, a domain dictionary and a bit of documentation.

---

## 2.4 Focusing on what the system does: the system lifeblood

The architecture in place, we have a firm foundation where we can stand and respond to change. Most changes are new or revised end-user services. It is these services, where the action is, that are at the heart of system design. To belabor the metaphor with building architecture: Whereas buildings change over decades and centuries [Bra1995], computers enact tasks at human time scales, and this animation is key to their role in life. It’s important to collect and frequently refresh insights about the end user’s connection to these system services.

We model the end user’s *mental model* of these services as the *roles* or *actors* they envision interaction inside the program, and by the interactions between these roles. Such modeling is a foundation of good interface design [Ras2000] and is the original foundation of the object paradigm [Ree1996], [Lau1993]. Use Cases capture roles and their interactions well. They are not only a good tool to capture end user world models (*if used in a lightweight way*) but also serve to organize requirements by their functional grouping, mutual dependency, and priority ordering.



**Figure 6: Focusing on what the system *does***

A sound architecture provides a context in which the user scenarios can unfold. These scenarios have two parts: the pure business logic, and the domain logic that supports the scenario. Most architectural approaches would have us do the “platform” code up front. We establish the form up front, but we wait to fill in the structure of the domain logic until we know more about the feature. And architecture lubricates the value stream: when the feature comes along, we don’t have to create the form from scratch. It’s already there.

So we start with my grandfather’s saw again: you can observe a lot just by watching. We observe our users and become sensitive to their needs, so our software improves their business. User experience folks capture business models and end-user cognitive models;

screen designers build prototypes; but the programmer is continuously engaged, collecting insights to be used during the coding sprint. Testers pay close heed to what the system is supposed to do. The programmer also keeps an eye on all the code that has to work together with the new stuff, so the current base and neighboring software systems also come into analysis.

This is an exploration activity but, in the spirit of Scrum, we always want to focus on delivering something at the end of every work cycle. The artifacts that we deliver include:

1. Screen Designs, so the program not only works, but is usable;
2. Input to the Domain Model, to scope the domain;
3. Use Cases, to organize the timing of responsible decisions.

---

## 2.5 *Design and Code*

My grandfather not only built two houses for himself and worked on countless others, but was also a pretty good cabinetmaker. He was acquainted both with structure in the large and in the small. When it came down to construction, he'd say: Measure twice, cut once. And as life went on, he re-designed parts of the house, built new furniture, changed a bedroom into a library, and made countless other changes.

Because it all comes down to code, design and coding entail a flurry of activities. By this time, many emergent requirements have been flushed out, and implementation can often move swiftly ahead. Developers lead the effort to produce code for the desired end-user services. With respect to process, we don't distinguish between code in the "architecture part" part and the "Use Case part" at this point. We code up what it takes to deliver business value. The stakeholders should be at hand to clarify needs. And this is where the testers turn Use Cases into tests and run them.

The main activities at this point are:

1. Turn the Use Case scenarios into algorithms and craft the roles that will be the home for the code that reflects end-user understanding of system scenarios. This is one of the most exciting parts of the book, as it builds on the recently developed DCI (Data, Context and Interaction) architecture from Trygve Reenskaug. This is code that you can inspect, analyze, and test more easily than under a simple object-oriented approach.
2. Write system tests.
3. Tailor the domain class interfaces to the new algorithms.
4. Code up the algorithms in the role code and the support logic in the domain code, re-factoring along the way.
5. Run the newly written system tests against the new code.

The output is running, tested, and documented code and tests.

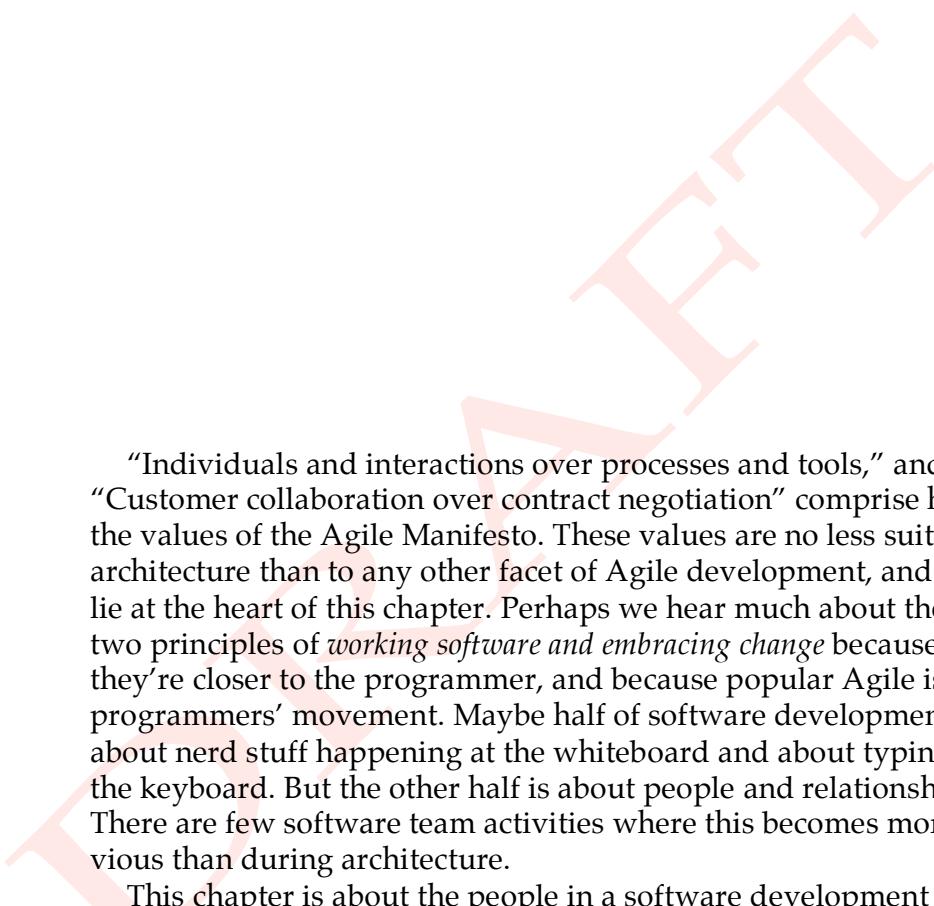
The functional code in the DCI architecture—what the system *does*—has the curious property of being readable with respect to requirement concerns. We have an updated domain architecture—what the system *is*—to support the new functionality.

---

## 2.6 *Countdown: 3, 2, 1...*

There, you have it: everything your mother should have taught you about Lean software architecture and how it supports Agile development. We'll tell you a lot more about our perspectives on these ideas in the coming pages. Don't be a slave to what we tell you but let our perspectives challenge and inspire your thinking. We think that you'll stumble onto new ways of seeing many of these things.

# *Stakeholder Engagement*



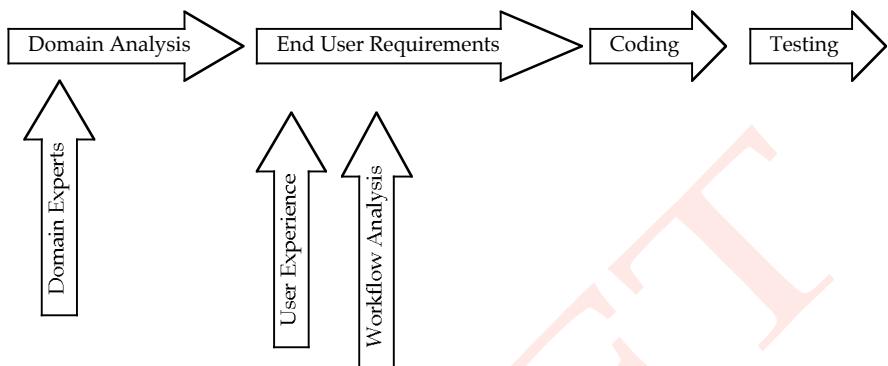
“Individuals and interactions over processes and tools,” and “Customer collaboration over contract negotiation” comprise half of the values of the Agile Manifesto. These values are no less suitable to architecture than to any other facet of Agile development, and they lie at the heart of this chapter. Perhaps we hear much about the other two principles of *working software* and *embracing change* because they’re closer to the programmer, and because popular Agile is a programmers’ movement. Maybe half of software development is about nerd stuff happening at the whiteboard and about typing at the keyboard. But the other half is about people and relationships. There are few software team activities where this becomes more obvious than during architecture.

This chapter is about the people in a software development effort, the roles they play, and a little bit about the processes that guide them. To summarize the chapter, architecture is everybody’s job. We’ll maybe have some surprising things to say about customers and architects, too.

Going back to that saw, “Customer collaboration over contracts and negotiation” brings to mind another human-centered artifact of

development: *requirements*, or whatever term one chooses to ascribe to them. Though the values of the Manifesto don't mention end users, Agile practices tend to focus on the informally communicated needs of users, commonly called User Stories. Now you've picked up a book about architecture—one that even dares to use the word *requirements*—and perhaps you're afraid that all of that is going to go away.

No, don't worry: in fact, we're going to push this perspective even further. We'll venture beyond just the paying customer to the even more-important end user, who is often a different person than the customer. In fact, there are many stakeholders who can derive value from your product. Some of these stake-holding relationships come down to economics: Both customers and end users can benefit from low incremental cost and fast incremental time to market. End-users are stakeholders who want a well-tailored solution to *their* problem as well, including ease of use. Both Lean and Agile can help us out here. Lean teaches us that the value stream is important, and that every activity and every artifact should contribute to end-user value. Agile's "customer collaboration" lets the value stream reach the deepest recesses of the software development process so user needs are explicit and visible to the vendor community.



**Figure 7: The Lean Value Stream**

There are important stakeholders outside your market as well. Your company benefits from a high return on investment (ROI), which comes from low cost and good sales. While good sales come from meeting market stakeholders' needs, low costs relate to how vendors meet their own needs. Designers, coders and testers have more fulfilling jobs if they feel they are contributing to the value stream: reducing time to market by shortening the development interval, and reducing cost by reducing frustrating rework. We can reduce rework by carefully watching and listening to our customers and to the market on one hand, but to history and standards on the other. So both Lean and Agile can help on the vendor side, too.

Lean adds to the value stream by keeping the system consistent, and a good architecture is a de-facto standard that helps system parts fit together. Lean supports the value stream by reducing waste: a good architecture based on historical knowledge can greatly reduce the rework of re-inventing the wheel. Agile keeps the links in the value stream consistent with each other through "individuals and interactions over processes and tools." The feedback in Agile supports Lean's goal of consistency, while Lean's trimmed and fit processes make Agile change more flexible.

Along with eliminating waste and smoothing out flow, overall consistency completes the three fundamental goals of Lean. For a software team, that means consistency between end user needs and what is produced. That implies that we must have consistency between the end user mental models, the end user actions, our Use Cases, the code, its architecture, the tests, the human interface—everything, everyone, everywhere. If everyone is communicating at just the right time, then we can avoid inconsistent actions that lead to waste and irregular production. And it's hard to schedule the "right time" in advance because the future is full of surprises. You could wait until you were done coding and testing to engage the sales people, but maybe they want to develop an advertising campaign so they're ready to sell when you're done. You could wait until you're done coding to make friends with the testers, but engaging them early smoothes later communication and builds the environment of team trust that is crucial to success.

The key to success in both Lean and Agile is the Lean secret: Everybody, all at once, from the beginning. For Lean architecture and Agile production to work, we must engage the right stakeholders and ensure that the process keeps them on the same page, optimizing value to the end user.

---

### 3.1 *The key stakeholders*

We identify five major stakeholder areas:

- the end users,
- the business,
- customers,
- domain experts,
- and developers.

You might be tempted to partition these roles by process phases. However, Agile's emphasis on feedback and Lean's emphasis on consistency suggests another organization: that these roles are more fully present in every development phase than you find in most software organizations.

This doesn't mean that everyone must attend every meeting, and it doesn't mean that you should force your end users or business people to write code. It does mean breaking down the traditional organizational walls that hamper communication between these roles, particularly at those times of crucial design decisions.

A good development organization is organized around roles that have a solid place in the lean value stream. In the Organizational Patterns book (PRODUCER ROLES, [CopHar2004]) we discussed producer roles, supporting roles, and deadbeat roles. You might think that your organization has no deadbeat roles—until you map out your value stream.

We probably miss some of your favorite roles in our discussion here. We discuss the roles that have distinguished contributions to the value stream, and a few of the key roles who help lubricate the value stream. For example, user experience people open doors to the end user role. The roles here should be interpreted broadly: so, for example, “developer” covers designers, coders, and testers; “domain expert” may cover your notion of architect, system engineer, and maybe business analyst or internal consultant; “business” includes sales and marketing and perhaps key management roles; “end user” has the user experience role in their corner; and so forth.

Since this is a book on architecture you might expect more conscious focus on the architect. A true software architect is one who is a domain expert, who knows how to apply the domain expertise to the design of a particular system, and who materially participates in implementation (ARCHITECT CONTROLS PRODUCT and ARCHITECT ALSO IMPLEMENTS in [CopHar2004]). Such an architect can be a valuable asset. We still avoid that term in this book for many reasons. First, having a role named “architect” suggests that the people filling that role are the ones responsible for architecture, when in fact a much broader constituency drives system design. Second, the word

too often raises a vision of someone who is too far removed from the feedback loop of day-to-day coding to be effective. This drives to a key point of Agile software development, and can be traced back to the Software Pattern discipline and to Christopher Alexander's notion of the architecture as "master builder" rather than "artistic genius." The term itself has dubious historic value. As the best-selling architecture critic Witold Rybczynski writes in his book *The Most Beautiful House in the World*,

... But who is an architect?... For centuries, the difference between master masons, journeymen builders, joiners, diletantes, gifted amateurs, and architects has been ill defined. The great Renaissance buildings, for example, were designed by a variety of non-architects. Brunelleschi was trained as a goldsmith; Michelangelo as a sculptor, Leonardo da Vinci as a painter, and Alberti as a lawyer; only Bramante, who was also a painter, had formally studied building. These men are termed architects because, among other things, they created architecture—a tautology that explains nothing. [Ryb1989, p. 9]

Third, most valuable contributions of good architects are captured in the other roles mentioned above: domain experts, the business, and developers.

We don't separate out system engineering as a separate role. It isn't because we find systems engineering boring or useless; it's more that few people know what it is anymore, those that do don't need much advice. For the purposes of this book, systems engineers can be viewed as domain experts (which are too commonly called architects) who can translate architectural idealism into stark reality. Done right, an architect's job looks like great architecture; a coder's work looks like great craftsmanship; and a system engineer's job looks like magic.

Let's explore the roles.

### 3.1.1 End Users

End users anchor the value stream. The buck stops there. Their stake in your system is that it does what they *expect* it to. By “expect” I don’t mean a casual wish, but a *tacit* expectation that goes beyond conscious assumptions to models that lie partly in the user unconscious. Most customers can express their wants; some can justify their needs; and *when using your product* they can all tell you whether it does what they *expect* it to. Most software requirements techniques (and some architecture techniques) start by asking users what they *want* or *need* the system to do, rather than focusing on what they *expect* it to do. Too many projects collect lists of potential features, driven by the business view of what the customer is willing to pay for (in other words, what the customer *wants*). Just having a description of a feature alone doesn’t tell us much about *how* it will be used or *why* he or she will use it (we’ll talk more about the *why* question in Section 6.2.1).

I learned a lesson in user expectations from a client of mine. The client builds noise analysis systems for everything from automobiles to vacuum cleaners. One thing I learned is that their clients for vacuum cleaners insist that a good vacuum cleaner make a substantial amount of noise when on the high setting. For the German market, this is a low roar; for other markets, it is a high whine. The goal isn’t to minimize noise, even though it’s possible to make vacuum cleaners very quiet without reducing their effectiveness. The goal is to meet user *expectations*.

One way to uncover expectations is to start a conversation with User Stories and proceed to goal-driven Use Cases that include scenarios and user motivations for the functionality. User Stories help end users think in a concrete way, and if we feed back our understanding of their expectations, we can gain foresight into how they will react if the system were deployed according to *our* understanding. If we do that job well we can build a system that technically meets end user needs.

Once you’ve gotten the conversation started, make it concrete quickly. One thing I learned from Dani Weinberg many years ago,

from dog training, is that dogs learn well from timely feedback. They have difficulty associating delayed feedback with the associated behavior so they are at best bewildered by your praise or criticism. If you can turn around a feature quickly at low cost to deliver to the end user for a test drive, you can concretely calibrate your interpretation of their expectations. Prototypes can be a good vehicle to elicit the same feedback at lower cost and in shorter time.

One problem with requirements is that they never anticipate all the scenarios that the user will conceive. So we need to go deeper than scenarios and explore the end user's perception of the system *form*. This is called the *end user cognitive model*, and it has everything to do with architecture. And it closely relates to what the user *expects* from a system. We'll explore the mechanics of User Stories and Use Cases more in CHAPTER 6, but here we'll help set the context for those practices.

### ***Psyching out the end users***

Use Cases capture user/system interactions that end users can anticipate. If end users can anticipate every interaction for every possible data value and input, then the specification is complete. It also makes it unnecessary to build the system because we've delineated every possible input and every possible answer, and we could just look up the answer in the spec instead of running the program. Of course, that is a bit absurd. Good software systems have value because they can, to some degree, handle the unanticipated. For example, let's say that my word processor supports tables, text paragraphs, and figures. One Use Case captures the user/system interactions to move a table within the document. I can describe the possible scenarios: moving a table from one page to another; moving the table to a point *within* a paragraph; moving the table just *between* two existing paragraphs; or even moving a table within another table. The possibilities are endless.

Instead of depending on an exhaustive compilation of Use Case scenarios alone, we instead turn to something that matters even more: the end user's cognitive model. Users carry models in their

head of the internals of the program they are using. They trust that the program “knows” about tables, paragraphs, and figures. The end user trusts the programmer to have paid attention to the need for white space between the table and any adjoining paragraphs. These elements of the end user mental model, while quite static, are useful for reasoning about most possible Use Case scenarios.

If the program doesn’t have an internal representation of a text paragraph, of a table and of a figure, then the program must work hard to present the illusion that it does. Otherwise, the program will endlessly surprise the end user. This is true even for scenarios that the end user might not have anticipated while helping the team compile requirements. Another way of talking about user expectations is to note that, unless we are designing video games, the end user rarely finds surprises to be pleasant. Therefore, it is crucial that the system architecture reflect the end user cognitive model. That helps us design the system so it not only meets end users anticipated wants and needs, but so it is also resilient when asked to support an un-anticipated by reasonable scenario.

If we capture this model well, the system and its initial code will handle most reasonable scenarios as they arise. Users will of course invent or stumble onto scenarios that the system doesn’t yet handle quite right—perhaps, for example, creating a table within the table of contents. As programmers we then need to extend the system to support the new Use Case. But the system *form* is unlikely to fundamentally change. Such business domains models remain relatively stable over time.

When it comes to testing and to end-user engagement in ongoing feature development, we want the end user’s feedback about how the system behavior matches their expectations. System tests, usability testing, and end-user demos all help. But when we are laying out the product architecture, we want the end user’s cognitive model of the system. Because the architecture reflects that model, the end user is a stakeholder in the architecture. In fact, the original goal of object-orientation was that the code capture the end user mental model (we’ll speak more to this in Section 6.1 and again in Section 7.1); to do that, we need to elicit the model.

It can sometimes be difficult for programmers to separate themselves from their engineering world enough to grasp the end user perspective, and it can be difficult for end users to objectively introspect about what their internal world models really are. This is where user experience people bring value: they are often the key to the stake held by the end user. User experience people are also trained to recommend interfaces and approaches that, while natural to end users and their mental models and behaviors, look crazy to a programmer. (If you don't believe us, just read Raskin's book on interaction design [Ras2000]).

### *Don't forget behavior*

Yes, of course we still collect Use Cases, even if they handle only the most common cases and even if they can't be exhaustively enumerated. Software ultimately is a service, not a product, and Use Case scenarios help us keep that fact in focus. The more Use Cases, the better? Well, but moderation is a key virtue here. Gathering requirements is expensive and hard, and it's easy to go too deep into requirements too early, because end users may behave differently when faced with a delivered system than they envision in the abstract. To gather detailed requirements about ever-changing behaviors is waste, so that's not Lean.

Also, it's important to strike a balance between domain modeling (CHAPTER 5) and behavior modeling (CHAPTER 6). Behavior has form, too (ever hear the phrase "form follows function"?), is part of the end user mental model, and should be captured in the architecture. In the long term, the domain structure tends to be more stable than the behaviors we capture in Use Case scenarios, and its forms are the cornerstones of a good architecture. Object-oriented design techniques traditionally have been good at capturing the domain model but really bad at capturing the behavioral models.

There's another good reason to capture Use Cases: testers are also stakeholders in the system's quality and then need something to exercise the system behavior. In fact, one good way to drive system design is to gather the domain structure and the system behaviors in

---

parallel and to let both drive the design. This is called behavior-driven development (BDD [TODO: BDD Cite]). Note that this means that the end user, the developer, the tester and the user experience specialist should be engaged together from very early in the project.

Focusing more on form than on function will help drive you in a direction that supports what users *expect*. Do that well, and you'll easily be able to provide what they say that they *want* and *need*.

### *The End User landscape*

Many systems have multiple end users and potentially multiple value streams. Think of a banking system. Simple exercises in object-oriented design courses often present account classes as the typical design building blocks and account-holders as the typical users. Yet bank tellers are also end users of a bank computer system. So are actuaries: the folks who look at the details of financial transactions inside the bank. So are the loan officers and the investment staff inside the bank. Does the concept of "account" suit them all?

Identifying such end user communities is crucial to a sound architecture. Different user communities have different domain models. Derivatives and commodities are potential domain entities to the investor, but not to the teller nor to the loan offer. Mortgages are domain entities to the loan folks but not the investment people. Yet somehow banks seem to run with a single common underlying domain model. What is it? In most complex financial system, the basic building blocks are financial transactions: those become the "data model." From the user interface, different end users have the illusion that the system comprises accounts and loans and pork bellies. We'll find that accounts really fall into a middle ground called Contexts: stateless collections of related behavior that behave like domain objects but which are richer in functionality and closer to the end user than many domain objects are. The Agile world has started to recognize the need for such user community differentiation, and that realization is showing up in tools like Concept Maps [TODO: Cite Jeff Patton].

Much of the rest of this book is about learning to identify these layers and to create an architecture that will allow the most common changes over time to be encapsulated or handled locally. Object-oriented architecture isn't just a matter of "underline the nouns in the requirements document" any more. Before a project can build a resilient architecture, its people must know enough about the end user communities and their many mental models that they can form the system around long-term stable concepts of the business.

### 3.1.2 The Business

While end users ultimately wants the software to provide some service for them, the business has a stake in providing that service, as well as a stake in the well-being of its employees. That usually means paying the employees a wage or salary and/or bonuses, and that usually implies making money from the software. Good software businesses usually have a stake in growing their customer base and giving good return to investors—which means that the enterprise will have a diversity of end users or customers to support in the market. We usually think of business stake-holding as lying with sales and marketing, or with the Product Owner in Scrum.

If the business can serve more customers, it both grows its stake in its customer base and likely increases revenues. Good revenues are one way to be able to pay employees well; another is to reduce costs. Architecture is a means to hold down long-term development costs.

The business itself provides key inputs to the architectural effort that can hold down cost. One of the most important business decisions is the project scope: the business owns this decision. Nonetheless, it must be an informed decision. Scoping has to balance the expectations of all customers and users against financial objectives (should the scope be the union of the entire market's expectations? Or should the business focus on the 20% of the market where 80% of the revenues lie?) The product scope can't take development into an area that depends on technology that won't yet be mature in the product's lifetime (are we building electric cars? Business forecast

---

software that uses artificial intelligence?) Such insight comes from domain experts and developers.

Just who is “the business”? The board of directors, executive management, business management, marketing, and sales are all key roles in this broad view of the business. Again, you probably don’t need all of these people at all meetings. But if you are discussing scope, or the what-the-system-does part of the architecture, invite selected representatives of these areas to the table.

The business may also hold down costs by making a buy-versus-build decision. Such decisions of course have broad and lasting influence on the architecture. Such decisions should be informed by customers and end users, who desire selected standards, and by domain experts who can advise the business on the feasibility of integrating third-party software. And don’t forget the developers, who actually have to do the work of integrating that software, and the testers, who have the burden of testing a system that contains a potential “black box.”

### *A special note for Managers*

It’s perhaps noteworthy that though Scrum is a risk-reduction framework, or a framework to optimize ROI, it has no role named manager. The ScrumMaster has all many of the characteristics of a good servant-leadership style manager, and the Product Owner has the business savvy and the command-style attributes of a good Product Manager while lacking all of that role’s control-style attributes.

Remember that two keystones of Agile are self-organization and feedback. For your team to be successful, you should use the influence and power of your position to help make that happen. One of the most important roles of line management is to remove impediments that frustrate the team, or that slow the team’s progress. A line manager’s attitude can make or break the esprit de corps of a team. That is worth more than any methodology or tool.

A good way to think about managers in an Agile context is as members of a team whose product is the organization. As such,

managers aren't preoccupied with the production process for the enterprise product; instead, they influence that process by putting the right organization in place. Good organizations support effective communication through group autonomy, collocation, and group functions.

While developers should set their horizons on end users who care about the product, managers can focus on customers. This helps free the development team from "pure business" issues where they have less experience, insight, or responsibility than managers. Because most software these days is sold as a commodity, customers are more concerned with revenue streams and delivery dates than the actual product itself. That means that they may be more interested in the process than the product. Managers are a good entry point for these concerns—much better than the development team.

All that said, don't forget: everybody, all together, from the beginning.

### 3.1.3 Customers

*Customer* is an almost emotive word. To not sign up to be customer-driven, or to strive for customer satisfaction, is to be a heathen and to be "not a team player." If we look beyond the mythical associations of the terms we find that it's useful to separate the Customer role from the End User role. Simply put: End Users are more about products and services, while Customers are more about process.

#### *... as contrasted with end users*

Customers and end users are interesting links in the value stream. Agile uses the word "customer" and we find the word featured both in the Agile manifesto and in much of the original Scrum vocabulary. Yet the Agile Manifesto says nothing about the end user, and we don't find either role formally in today's Scrum framework!

Customer and end users are very different stakeholders. The end users' stake is relatively simple by comparison. They seek service from the software you are developing; that is the value that you

---

supply for them. The end-user value to the development organization is that they are usually the source of revenues that ultimately feed your team members. It's a good deal for them if your software in fact supports services that increase the end user's quality of life.

The customer is in general a middleman. In general terms, customers are not consumers of the service that your software provides; they treat your software as a product that passes through their systems the same way that gold in a Japanese martini passes through the digestive system of its consumer. It may come out the other end in different packaging, but it is still the same product. When engaging such customers, consider their stake in opportunistically developing products for which there are yet no end users. While all stakeholders want to reduce risk, customers in this position are particularly averse to risk. They are much more interested in delivery times and in your development costs (because those become their costs) than they are in functionality. *Therefore, the customer has a larger stake in your development process than in the service that your software provides.* You may be engaging customers more in the area of process improvement than in development enactment. It is important to accord such activities a place in your enterprise using retrospectives—and here, retrospective means a serious activity that encompasses business scope and issues of trust. What passes for a retrospective in the two-hour "check-ups" at the end-of-sprint is inadequate. See [Ker2001] for more on retrospectives.

Customers have at least one other key stake-holding relationship to the architecture, and that relates to the market segments that they serve. In general, when it comes to customers, the more the merrier: Customers are a path to markets and therefore to revenues. However, different customers often represent different constituencies and bring the power of negotiation or market leverage to the negotiating table. If customers want to distinguish themselves from their competition, they will want their own configuration of your product. So one size does not fit all. Such configurations may extend beyond simple algorithms to variations on the system form: its architecture. A good architecture can be a tool that helps the Business cater to the

needs of individual customers and market segments by supporting plug-and-play substitution of system modules.

Of course many combinations of customer and end-user are possible. They are sometimes one and the same. Sometimes you have the luxury of working directly with end-users, achieving the ultimate Agile goal of short feedback loops that avoid requirements misunderstandings.

It is common that a Scrum team delivers to another software team developing code in which your code is embedded. These projects are challenging to run in an Agile way. It is rare that such a team has or even can have meaningful discussions with end users. If your software has repercussions on end users (and what software doesn't?), then your own customer is likely to introduce delay that makes it difficult to receive timely customer feedback before starting your next sprint. Testing, and, in general, most notions of "done," become difficult. *In these situations it is much better to extend the scope of "done" to include such customers and to effectively enlarge the scope of development to engage the party as a development partner rather than as a customer.*

Sometimes you have customers who yet have no end users because they are striving to develop a market, trying to develop a service that they hope will sell. In an Agile context, be wary of the possibility that your customer will look to you as a vendor to be the source of the requirements!

Other times you yourself are both the developer and the end user, such as might occur when developing tools that support the development team. That's great! We encourage you to continue the discipline of separating your customer role from your developer role. Dave Byers relates:

Because in the developer role you're trying to get away with doing as little as possible, but in the customer/user role you want as much done as possible. Separate the two and it's possible to find a decent balance. Don't separate them and chances are you'll drift too far to one or the other. [Bye2008a]

---

Indeed, the possibilities are endless. Use common sense, guided but not constrained by Agile and lean principles.

### *“Customers” in the value stream*

Sometimes our “customers” are just politically or organizationally separate entities on the production side of the value stream. If you build framework software and sell it to a company that embeds it in their product, then there are no real end-users of your product in your customer. Your end users are on the other side of your customer—and that can cause a break in the value stream.

If you are in this situation, look carefully at the interfaces between the organizations and look for waste, delay, inconsistency, or boom-and-bust production cycles. If you find such problems, then Lean has some answers for you. (If you don’t find such problems, then that’s great! Like my grandfather said: If it ain’t broke don’t fix it.)

Lean’s main answer to these problems is to integrate both parties more fully into a single value stream. Remove obstacles to feedback between the teams. Leverage standards as a supplement to communication, and as a way of reducing changes in dependencies.

Remember that the same principle applies if you are taking software from another vendor and embedding it in your project. Try to close the gap. Toyota did this when they found they needed a new battery for their hybrid car. They didn’t have the expertise to build one in-house and couldn’t find one from any supplier. They solved the problem by partnering with Matsushita to jointly design a battery uniquely suited to the design of the Prius. [Lik2004, pp. 208-209]

#### **3.1.4 Domain Experts**

Domain experts are usually the grey-haired folks in the organization who know stuff. Domain experts are often the most direct and most explicit source of insight and advice on how to structure a new system. Most new systems in a domain look—from the perspective of form—very much like previous systems in the same domain.

It's important to understand that everyone in an organization is probably an expert on something; otherwise, they wouldn't be there. Software development is rarely a matter of having enough muscle to get the job done, but rather of having the right skill sets there. It's about having diversity of skill sets, not just that one can overtake the market using Java muscle.

On the other hand, the folks commonly *called* domain experts have a special role in architecture. Over the years they have integrated the perspectives of multiple end user communities and other stakeholders into the forms that underlie the best systems.

Such knowledge is a priceless asset. Consider the alternative. With no knowledge of the best long-term structure of the system, designers would have to start with first principles—end user domain models at best, but more likely Use Cases—and try to derive the objects from those. It becomes more difficult if the team must deal with Use Cases from several different kinds of end users (e.g., both savings account holders and actuaries for a bank), and becomes even more difficult if the scope covers multiple clients or customers. The knowledge of the form suitable to such a complex landscape assimilates over years or decades, not over sprints or months. If you have your domain experts handy, they can relate the forms that they have already integrated, and in any case can point out areas that have been particularly challenging in the past.

Domain expert engagement is to architecture as end user engagement is to feature development. You should find that end users and domain experts are your most treasured contacts in a lean and Agile project. It is difficult to establish good working relationships with both of these roles (with end users because of organizational boundaries and with domain experts because of their scarcity), but make the extra effort. It's worth it.

### ***No ivory tower architects***

Domain experts often bear the title of Architect. In the Organizational Patterns book [CopHar2004] we find patterns such as ARCHITECT CONTROLS PRODUCT and ARCHITECT ALSO IMPLEMENTS,

---

that use the name “architect” exactly in the sense of this common title. But both patterns suggest the architectural principles and domain expertise embodied in the role.

Today, we prefer the term “domain expert” more and more and the term “architect” less and less. The reason? In practice, “architect” isn’t a very distinguishing title. Interaction designers and coders have as much or more influence on the overall form of the system—its architecture—as titled architects do. In an Agile framework we value everybody’s contribution to the architecture, and to have a titled “architect” can actually disempower other stakeholders with deep insights. Differentiating the role of “domain expert” along the lines of expertise and experience, instead of along the lines of contribution to product foundations, better captures the stake-holding relationships.

### *Experts in both problem and solution domains*

Don’t forget solution domain experts! It’s easy to get too caught up in value stream thinking that insists on tracing all business decisions to the end user as stakeholder. The business is also a stakeholder, as are the developers, and they want to use the best tools and design techniques possible. In the most general and theoretical sense, this eventually contributes to the value stream in a way that will benefit the end user, but it’s easier to think about it in terms of the more direct benefits to the team: benefits that help them do their job better.

Innovation in the solution domain goes hand-in-hand with long-term experience from solution domain experts. A good object-oriented expert can tell you not only where OO will give you benefits, but can also tell you where it won’t give you benefits. (In fact, a good rule of thumb is to trust someone as an expert only if they are good at telling a balanced story. An advocate is not always an expert, and an expert is not always an advocate.) So you want good dialog on the team between the innovators (which can be any role on the team) and the solution domain experts.

Keep your architecture team balanced so that both problem domain experts and solution domain experts have an equal say. One problem with stovepipe development is that needed dialog between these two perspectives turns into a war, because the earlier one in the process over-constrains the other. It is often a problem for a business to be driven too much by technological innovation, but it is even more problematic to be driven by the MBAs. To read a depressing case study about how this imbalance can go wrong, read Richard Gabriel's post-mortem of Lisp innovator Lucid [Gab1998].

### 3.1.5 Developers and Testers

Developers are where the rubber meets the road. Their main job in architecture is often to rein in the grand visions of the business and architects with grounded domain expertise. As in Scrum, the developers should own the development estimates—after all, they're the ones who will do the actual work of implementing. As we mentioned before it's even better if the architects also implement—or, turning it the other way in terms of this book's terminology, if at least some of the developers also have deep domain expertise.

As such, developers are the main oracle of technical feasibility. They are the primary solution domain experts. They should be active experts. For the tough questions, opinion isn't well informed enough to make a long-term business decision or architecture decision. It's important to gather empirical insights. Developers can help by building prototypes that compare and contrast architectural alternatives that are up for discussion. At PatientKeeper in Massachusetts, the Scrum Product Owners might spend months building prototypes and workflow models to refine their understanding of the design space. Developers are taxed to support them with prototyping tools and in building the actual prototypes.

Remember that developers are the primary channels of interaction between teams. If you believe Conway's Law, that says that the team structure mirrors the architecture, then you can probably believe that the interaction between parts of your architecture will be only as effective as the interactions between the team members representing

---

those parts of the architecture. As you frame out the form of your system, make sure that the stakeholders for the parts—at the level of the coders—negotiate the interfaces through which they will interact. Much of this negotiation will of course involve domain experts.

Developers and testers should be friends. While every serious system needs some acceptance or system testers who write double-blind tests, you should have ongoing testing support during development. You even need that at the system level. For such testing, the testers need to know requirements at least as well as the developers, so they’re likely to be invited to a lot of the same meetings as developers.

Even though developers and testers should be friends, at least some of them should play a game of “hide and seek.” The developer and tester can have a friendly meeting with the business people to agree on the requirements, and then they go their separate ways. The tester codes up tests for the new feature while the developer implements the feature in the current architecture. The developers works hard to implement the feature exactly as they come to understand it, going back to the business if necessary. The testers work hard to test the feature as they understand it, also asking for clarification when they need it. After one or two days of work, they come together to see if their perspectives meet up. My grandfather used to say that two heads are better than one. Start by having two sharp thinkers develop their interpretation independently; this doubles the opportunity to discover latent misunderstandings. That’s the “hiding” part of hide-and-seek. Working independently avoids groupthink, and avoids one personality overpowering the other with arguments that things must be thus-and-so [Jan1971]. Then, having “found” each other, reason together (again, with the business if necessary) to clarify mismatches. Not only does this approach increase the chance of uncovering requirements problems, but it is a weak form of pipelining, or parallelism, that shortens feedback cycles.

As for testers, there’s an old saw that *architecture defines your test points*. Testers have a stake in the architecture that it be testable. Hardware designers have become good at something called DFT, or “design for testability.” Software people haven’t come quite that far,

but some testers have more insight and instinct in this area than others. Use testers' insight to make key APIs available to support your test program.

Last but certainly not least are usability testers. Usability testing comes rather early in development, after Use Cases have been firmed up and before coding begins. You can do user experience testing with mock-ups, prototypes, or with simple hard-copy mock-ups of the anticipated screen designs. Usability testing can validate whether the team has captured the end user mental models properly: a crucial test of the architecture.

---

### 3.2 *Process elements of stakeholder engagement*

Your longstanding development process is in place, and you're wondering what an Agile process should look like. Traditional development processes often organize into one (or both) of two patterns: one that exhaustively covers the stages in sequence, and one that exhaustively covers the tasks in roles. A total ordering of tasks can over-constrain self-organization, and a task organization alone can become arbitrary if it's not grounded in knowledge or resources that support the value chain. A generic framework like RUP that tries to delineate all the roles in general has difficulty mapping onto domain-specific roles and processes, and it's difficult to map these roles onto the value stream.

In this book, we discuss only the most basic notions of software process as they relate to Lean architecture and Agile production. This is a book about architecture and implementation. You might ask: Where does architecture start and end? To answer that, we need to answer: What is architecture? Architecture isn't a sub-process, but a product of a process—a process called design. Design is the act of solving a problem. Viewed broadly, and a bit tongue-in-cheek, we might say that there are only three processes in software development: analysis (understanding the need); design (solving the problem); and delivery (which may include engineering, installation,

---

shipping and deployment). This means that the time scope of architecture is broad.

A pithy but adjustable problem statement makes a great project compass. To do a good job of analysis on an extensive, complex market takes time. Getting everyone in the same room shortens feedback loops; if you can't get them in the same room, minimize the number of communication hops between them (see the pattern **RESPONSIBILITIES ENGAGE** in [CopHar2004]). We care more about how roles connect to the value stream than to their place in the process (e.g., to have marketing people feed analysts who feed architects who feed designers who feed developers who feed testers) or to how their responsibilities fit together into a comprehensive set of tasks (as in RUP). You can find the stakeholders: the end users, the business, customers, domain experts, and developers in just about every enterprise that builds something for someone else. Some traditional software development processes translate well to an Agile and Lean world, but others merit special attention. So, with a light touch this section offers some rules of thumb on the process.

### 3.2.1 Getting started

The vision and problem statement come very early in development—often even before you have customers. Once the vision is in place, use your marketing people to extract knowledge from the market, and your domain experts to extract knowledge from the business world and technology sector. This knowledge can prepare both the business and the development community to shape their understanding of the domain and to start to understand the forms of the architecture.

If this is a new project, consider the organizational patterns that tie together the structures of your domain, your market, and the geographic distribution of your development team. (Here, “geographic distribution” includes separations as small as one building away or more than 50 meters distant. Don’t underestimate the power of space!) In particular, CONWAY’S LAW [CopHar2004] and the re-

lated patterns ORGANIZATION FOLLOWS LOCATION and ORGANIZATION FOLLOWS MARKET are major considerations in organizing your teams.

We have heard several conference talks on software architecture that start with a claim such as, “To get started on architecture, first get your requirements in hand.” It is true that we need to understand end user requirements if we are to deliver value to them; we dedicate CHAPTER 6, CHAPTER 7, and CHAPTER 8 to that topic. But the foundations of system structure lie elsewhere. The other side of the same modeling coin is domain expertise. Think of domain expertise as a broadening of the end user mental model into the realms of all stakeholders taken together. Expert developers learn over time what the fundamental building blocks of a given system should be. This is also the place where customer—as opposed to end user—concerns weigh most heavily. Some of those reflect the end user perspective, but they sometimes must honor the concerns of other stakeholders whose views are radically different from those of the end user. For example, checking Account Holders think that banking software comprises a ledger that mirrors the entries in their own checkbooks. However, an auditor chooses to view that account as a process over an audit trail in a transaction log. Both of these are real; which of these wins out as the systems foundation is a function of many Agile concerns, particularly ease of use, and frequency and type of changes to system functionality.

Those models balance enough generality to accommodate a wide variety of business scenarios with enough concreteness to provide a shared vocabulary for all stakeholders. Such care for the end user perspective on architecture, combined with its concreteness, take us from the problem definition one step closer to a delivered system. Of course, in real development we work on the problem definition, architecture, and use cases in parallel; however, from a conceptual perspective, architecture and its articulation provides a vocabulary and foundation for the later concerns with what the system does. So, guess what: *everybody, all at once, early on rules again*.

Also, if this is a new project, start small. Great projects grow from small projects that work. Have your developers work with analysts to explore the domain by building prototypes. Once you have a vi-

---

sion of where you're headed, put together one or two teams to frame out the architecture and to demonstrate rudimentary functionality. Here, "team" means five to seven people. Aim for an early success and for a firm foundation that will support the product throughout its lifetime. The primary consideration should be on supporting change in the long term, being particularly attentive to feedback loops; if you can change, you can improve the value stream. The second consideration is to build your process around the value stream. Strong domain knowledge, and its articulation in an architectural framework, is one of the best things you can do to support change and to draw attention to the value stream.

### 3.2.2 Customer Engagement

Your thoughts will soon turn to delivering features. We'll emphasize a previous point again: focus on user expectations rather than just wants or *your* perception of their needs. User experience people are experts in extracting (and anticipating) user expectations.

It is usually important to study end users and even customers in their native habitat. Don't bring them into your office, but go to theirs. It doesn't matter whether your office has all the touches of the best interior decorator in town, or whether it's just a nerd's paradise—it just can't replace the client's home base as an environment to learn domain knowledge and the context in which requirements arise.

To say this goes against the popular practice of on-site customer. Recent studies have found that on-site customers can in fact compound the requirements process by creating problems of trust [MaBiNo2004], [Mar2004]. On top of that is the more obvious problem of missing key contextual cues that arise in the environment. Our colleague Diana Velasco tells of a site visit where the client was describing the process they used but failed to mention the sticky notes posted around the border of the computer screen, and also wasn't conscious of the importance of the "crib sheet" notebook that everyone kept as a guide to navigating the screen command struc-

tures. These are crucial components of the developer world and are crucial to system architecture.

Beyer and Holtzblatt's book Contextual Design [BeyHol1998] offers a wealth of techniques for exploring and capturing end-user mental models. Be selective in the tools you adopt from this and other sources. Use these tools on customer site visits to garner insight both for architecture and Use Cases.

As described in Section 3.1.1, you want to actively elicit feedback from end users using short development cycles or by using prototypes and models during analysis. A good feedback cycle has the appearance of causing problems. It will cause emergent and latent requirements to surface. That means rework: the value of prototypes is that they push this rework back into analysis, where it has more value. And most important, good end user engagement *changes end user expectations*. It is only by participating in a feedback loop that's grounded in reality that customers get the opportunity they need to reflect on what they're asking for. If your customer changes their expectations in the process, you've both learned something. Embracing change doesn't just mean reacting to it: it means providing the catalysts that accelerate it.

### 3.2.3 Pipelining

Mary Poppendieck likes to tell the story of how the Empire State Building was built in a year. Consistent with Lean philosophy, the construction effort took advantage of just-in-time material delivery. That doesn't happen by accident but requires up-front planning. With a little up-front planning you can make room for parallelism and "pipeline" your process. And remember that part of planning is giving yourself room to change the plan.

The "materials" in software are often requirements. While development is coding up and testing the current release, the demand on the business people is lower and they have more time for analysis. The business can serve up a new set of requirements that are ready when the team reaches the end of its development interval (**Figure 8**).



**Figure 8: Pipelining with Look-Ahead**

All stakeholders are more or less involved in all phases of development. But as Release 1 moves into implementation, the team members whose talents lie in analysis become less involved with Release 1 and start looking ahead to Release 2. They will still be there to support Release 1, particularly by clarifying analysis issues. And the coders will be there to support analysis activities with development cost estimates, feasibility advice, and perhaps even prototype development. This is “swarm style development.” The entire team works as a community. Each role individually is more or less oblivious to whether a feature is in market research, formulation, development, test or deployment. Any team member may be called on to pull the feature along the value chain at any point in the process. Sure, the business people write most of the requirement documentation and, sure, the coder will be the one carrying the ball during implementation. But it doesn’t mean that the analyst can’t help clarify requirements during implementation. It doesn’t mean that the coder shouldn’t provide development estimates to the business people early on so they can get a feeling for the business value of a feature. You should think of having a network of stakeholders that swarm across the development process. We discuss this more in Section 3.3.

### *An Example: Scrum*

In Scrum, the Product Owner represents the business to optimize ROI. The team, which has responsibility for development, meets with the Product Owner at the beginning of every development interval (called a *Sprint*) to update estimates for all features which will be delivered in the short term. The Team also meets with the Product Owner weekly (called the “Wednesday afternoon meeting”) to update estimates on newly arrived requirements. During design and coding, the Product Owner is present to clarify requirements, though the Product Owner dedicates most of this period to preparing requirements for the next Sprint.

This is just some filler to force a page break. Remove me.

---

## *3.3 The Network of Stakeholders: Trimming Wasted Time*

Now that we have covered the roles, we come to the heart of the matter. There is nothing particularly lean or Agile about the roles themselves. What is important is how they work together. Here we come back to the Lean Secret: everybody, all at once, from the beginning.

### **3.3.1 Stovepipe versus Swarm**

Old-style software development is patterned after the industrial assembly-line models of the Henry Ford era. In a simple, old-fashioned assembly line, workers interact directly only with the people in the adjacent station on the line. Worse yet, they may not even interact with the people, but may focus totally on the artifact and on their task of reshaping it or attaching something to it that adds value to the product. In manufacturing one can push this independence all the way back into the design process, because even de-

---

signers can count on the laws of physics holding for the parts they design and how they will fit together and hold up in deployment. The development process is divided up into stovepipes: independent spheres of influence lined up side-by-side to create a product piecemeal.

Software has no equivalent to the laws of physics. Alistair Cockburn likens software construction to group poetry writing. It requires many different talents, ranging from knowledge of the business domain to good programming skills to keen insights into ergonomics and interaction design. What's worse, these skill sets can't easily be separated into process steps that can be done one at a time. And even worse, most of these skill sets drive some aspect of the basic system *form*: its *architecture*. If you follow that chain of dependencies, we arrive to the conclusion that we need everybody, all at once, from the beginning.

	End User	The Business	Customers	Domain Experts	Developers and Testers
End User		Feature Priorities, Scope	Purchase convenience	Product / feature Feasibility	Quality and proper functionality
The Business	Feasibility	Create Standards	Process Requirements	Feasibility	Source of revenue
Customers	A market	Products and Services	Create Standards	Compliance with Standards	Source of revenue
Domain Experts	Range of Product Variation	Workplace well-being	Need for Standards	Domain synergies and conflicts	Constraints on technology
Developers and Testers	Requirement Clarification	Workplace well-being	Advice on delivery process	Guidance, APIs, "poka yoke"	Clarification of how existing code works

Read down the columns to see what the roles contribute to the value stream; rows indicate the roles to whom the value is provided.

**Figure 9: Stakeholder Relationships**

Look at Figure 9, which summarizes stakeholder relationships discussed earlier in this chapter. If we had drawn such a diagram for a manufacturing assembly line, we could each role might have a direct dependency only on the one immediately preceding it in the process. But in software, there are essential, ongoing dependencies that form an almost fully connected network of dependencies between roles.

Many software organizations handle these dependencies in an ad-hoc way, which is more or less one at a time. If the architect is sitting at his or her desk writing the Big Architecture Document, and if he or she needs the insight of the interaction designer before proceeding, too often the information request must go "through channels." Such interactions usually draw many non-producer roles into the process, and that puts the architect into a wait state. If the architect is waiting, so are the GUI designers, the coders, the customers, and the end users. In the very best case, the answer will come back to the ar-

---

chitect in days or weeks and is still a useful piece of information that hasn't been invalidated by changes in the market, standards, technology, or development team staffing. More typically, the response raises as many questions as it provides answers (knowing the interaction designer's recommendation, do we need to ask the coder if we can implement it?). Unfortunately, the real scenario is that the architect makes an assumption or guesses simply because it's too wasteful of time to clarify the details. And that means that when the Big Architecture Document is unrolled to the interaction designer and coder, there will be much wailing and gnashing of teeth—and another big cycle of rework and waste.

So by trying to do the right thing in an assembly-line organization, an architect will cause delay. By failing to do the right thing but instead taking all the decisions upon himself or herself, the architect incurs even more delay. This is why architecture development takes months or years in linearly organized complex projects. (These are called "NASA-type phased program planning (PPP) systems" in [TokNon1986].) *It isn't that architecture is so much work; it's that everybody spends so much time waiting while Emails sit languish in-boxes, while architects write architecture documents, or unread memos sit awaiting review.*

A good team that develops relationships between the roles—relationships that correspond to the dependencies between stakeholders—can trim the architecture effort from months down to days or weeks.

Organize more like an insect swarm than as stovepipes. We're writing this chapter from the middle of the Swedish Northwoods. Yesterday we took a walk in the forest and passed several anthills. The largest one was more than a meter high and more than two meters in diameter, and every millimeter of its surface was alive with scurrying ants. We couldn't find a single project manager among them, nor a single process description. And we didn't see anyone in an architectural wait state.

If you're using Scrum, try to fit your architecture exercise into a single sprint. Who is the team? It's the cross-functional Scrum team.

If you're in a multi-team Scrum project, you'll need input from multiple teams.

Your team members should be collocated so they can respond to questions in seconds rather than hours, days, or weeks. What does "team" mean here? It comprises at least those roles described in this chapter. Too often, Agile initiatives limit Agile principles and practices to developers, perhaps with a token on-site customer thrown in. Scrutinizing roles and their take-holding relationships more carefully shows that things are more complicated than that.

### 3.3.2 The first thing you build

Brad Appleton is an old colleague of mine from the Pattern Community and is a long-time respected person of influence at Motorola. His E-mail byline has consistently said for years: "The first thing you build is trust."

Jerry Weinberg tells a story of a company where a highly placed, powerful manager issued an urgent request for new computing equipment. The requirements were a little bit sketchy, but he did insist that "the cabinets had to be blue." His subordinates, and purchasing, and others, scurried around trying to decode this supposedly mysterious message. "Blue! Does he mean to buy from 'big Blue' (IBM)?" "Does he mean that he wants it the same color as the other equipment? But some of it isn't blue!" Someone finally got the nerve to ask him (whether it was actually before or after the equipment arrived, I don't remember) and he said, "No, blue is my wife's favorite color, and she thought that the new computers should be blue."

One exercise in Lean is called "ask five times." If someone makes an unjustified claim, ask them about it. More often than not you'll get another unjustified claim. Within about five exchanges you'll come to the core of it. Why didn't anyone ask the executive why he wanted blue? It was perhaps out of fear of being an idiot for not knowing the answer. Or it was perhaps it was out of fear of potentially embarrassing the boss in public. There wasn't enough trust in the organization to clarify the requirements.

---

Jerry talks about egoless development teams—a commonly misunderstood phrase that simply means that you put your personal stake in perspective so you can defer to the team's stake as a whole. We've all heard the saw: "There are no stupid questions here," but we are not always good at following it. We should be. An Agile team is a team of trust that can ask such questions openly.

### 3.3.3 Keep the Team Together

As stakeholders, team members have expectations, too. In addition to their value stream expectations, they come to expect certain abilities, reactions, and ways of working from each other. Knowledge of such expectations, like most expectations, is tacit knowledge and takes time to develop.

To support the team in the ever-ongoing learning of how to become a team and become a better team, keep the team together over time. If you re-assemble teams for every new product or on a periodic business cycle, you force each team into the well-known cycle of forming, storming, and norming, before reaching a performing stage. That's waste. Get rid of it.

Teamwork works on the scale of milliseconds. Just watch a football team. Or, better, yet, watch a software team engaged in a design meeting. It's exactly this kind of feedback that can reduce architecture efforts from months to days by displacing formal communication channels and forums. If a team is not co-located, you lose these feedback loops. To sustain team effectiveness, keep the team together in space. A multi-site team can work but will have difficulty sustaining the same pace as a collocated team, everything else being equal. Martin Fowler writes [TODO: Cite] that multi-site development requires more written documentation and, in general, more formal communication styles.

There are many variations of Conway's Law that provide guidance for organizing teams. The primary organizing principle is that the team structure should reflect the architecture. However, even that is difficult, because architectures themselves have cross-cutting concerns. In the DCI architecture (CHAPTER 8), the structure of

roles and their interactions cuts across the structure of the domain objects. And beyond this simple part of Conway's Law, you also want the organizational structure to align with your market structure. You also want it to align with the physical distribution of people. You also want it to align with the structure of the business. Figuring out exactly how to structure a team means balancing the tradeoffs that emphasize different ones of these organizations.

For completeness, one organizational structure that we know does not work is to isolate all the architects in their own team.

No matter how you organize it's important to keep the boundaries between the teams thin. Any work on the architecture must cut across organizational boundaries.

It's hard. You're Agile. We trust you. You'll figure it out.

# *Problem Definition*

Architecture is one product of an activity called design, and there is no design without a problem. A problem definition is an explicit, written statement of a *problem*: the gap between a current state and a desired state.

Before we lay out the route to our destination, we have to know where we're going. More often than not, when I ask software developers what problem their product solves, the discussion goes something like this:

*What problem are you solving?*

"We're trying to become more object-oriented."

*No, that's a solution to some problem, not a problem. What problem are you solving?*

"Oh, we're using object orientation so we get better reuse."

*No: reuse is itself a solution to some problem. What problem are you solving?*

"Well, the last project was too costly and we're trying to reduce our costs."

*How many alternatives did you consider?*

"Well, none. Everyone else is using objects, so we decided to take a low-risk path."

If you recognize your organization in this reasoning, you're hardly alone.

My grandfather always had a sense of purpose when he set out to build something. Whether it was serious (a house for his family), generous (a small railway station for my model railroad set), or whimsical (carving a small wooden puzzle or toy) it was always *purposeful*. Your projects should probably be purposeful, too. A good problem definition can help point the way.

Many of the principles of problem definition apply in many other microcosms of design. Problems are closely related to goals in Use Cases and, in general, are closely linked to requirements. Keep this in mind when reading CHAPTER 6 in particular. However, also remember that problem definitions aren't a club that you can use to beat other project members into submission, and don't go looking for problem definitions under every rock. Most problems are a matter of everyday conversation and feedback. Related concepts such as goals and objectives have their own specific needs (Section 4.6).

---

## 4.1 What's Agile about problem definitions?

Agile is about working software. Software provides a service that solves some problem, and it works only if it solves that problem.

A good problem definition can be a catalyst for self-organization. The Agile notion of "self-organization" means neither "no organization" nor total lack of structure. Systems in nature that self-organize are called autopoietic systems. They usually organize around some simple law or set of laws or ideas, and always include a notion of reflection or at least of self-reference. Nothing outside of a cell organizes a cell; its structures take the molecules and energy in its environment to build and sustain the overall cell organization which in turn gives rise to these structures. [Autopo2009] Problem definitions can provide the catalyst, or seed crystal, that can cause a team to organize and figuratively crystallize its thoughts into a consistent whole.

---

## 4.2 What's Lean about problem definitions?

The deepest foundations of Lean feature a continuous process of innovation that increases value to the end user, and decreases or eliminates everything else. A good problem definition brings focus to the entire team—an outward focus that is broader than the local problems of their cubicle or work group and that ultimately supersedes any local focus.

To a casual observer, problem definitions look like waste. It is time spent away from the keyboard, and our Western upbringing tells us that we are avoiding work or being unproductive when “just talking.” But Lean is full of paradoxes like this ([Lik2004], pp. 8—9). Sometimes the best thing you can do is to idle your equipment and stop making parts; sometimes it is better to avoid computers and IT and to resort to manual processes, just because people are a more flexible resource.

Perhaps the most obvious tie from problem definitions to Lean is their foundation for consistency. Lean asks us to reduce tensions and inconsistencies in a system. A problem statement at least articulates a consistent objective. Too often projects suffer from the simple problem that its members are not all solving the same problem. A well-written problem statement offers a consistent vision of direction. As such, it can be a powerful team tool or management tool.

Problem definition starts at the beginning of a project and may evolve over time. You should view it as an up-front investment—not investment in a reusable artifact, but an investment in your people and your customers. Lean is based on “a culture of stopping or slowing down to get quality right the first time to enhance productivity in the long run ([Lik2004], p. 38). Reworking ideas early in the process helps you avoid the more costly reworking of code later in the process. This fits with Boehm’s software engineering findings that a bug discovered in a requirements review costs you 100 times less to fix than one discovered in the field [TODO: CITE]. Problem definition is one of your first chances to get it right. It takes time, but time taken here is potentially a lot of time saved later. As my grandfather used to say: A stitch in time saves nine.

We won't make any pretense that you'll always get it right the first time. Lean is also about continuous process improvement, about making turning every coder into an architectural innovator. It's about challenging the expectations of your end user and customer, lifting them to new levels of awareness about their own wants and needs. That means that you'll be chasing a moving target. But we almost always do, anyhow, and good process improvement coupled with Agile principles can help the target settle down more quickly. And as for moving targets—well, we embrace change. We'll discuss this more below.

The Lean literature is full of techniques that can support problem definition, such as asking "Why?" five times every time you encounter a problem; the goal is to drive to the root cause. [TODO: CITES?] Lean is designed for complicated systems, whose problems often can be isolated to a "root cause." Software systems are not only complicated: they are *complex*. Though you can track a system effect back to set of causes you can't always chart a path from cause to effect. We compensate for this with the Agile principle of frequent feedback. In any case, a good problem definition removes one large degree of uncertainty from development. It won't remove your team from the ship in the storm, but it will at least help ensure that they are on a chartered ship.

---

### 4.3 Good problem definitions

A good problem definition has these characteristics:

1. It is written down and shared.
2. It is a difference between the current state and some desired state of the organization or business.
3. Its achievement is measurable, usually at some mutually understood point in time.
4. It is short: one or two sentences in clear, simple, natural language.

- 
- 5. It is internally consistent: that is, it does not set up an over-constrained problem.

Though problem definitions are short, and can be developed in a short period of time, their importance far outweighs their size. We'll take a little time here exploring good problem definitions and some of the ways that problem definition can get off-track.

Here are some examples of good problem definitions:

To be able to sell a picture transfer application that will work with 90% of the phones on the market.

This is a pretty good problem definition. It is measurable. It defines the problem as "being able to sell," which is a business proposition, rather than "designing and building." The latter is a more constrained *solution*; as stated, it points more directly at the *problem*.

Here is another one:

All television signal transmissions will be converted to FCC standard digital format by 1 January, 2009.

The result is measurable (provided that the FCC standard is well-defined), and we are even told when we should apply the measurement to evaluate success. If we ask *Why?* we might be told that the law requires us to meet this conversion timetable. We might have written another problem statement that viewed the impending law itself as a problem and might have worked with our lobbyist to delay the enforcement of the law, at least for us. But *that* would be a different problem, and it would show up as a different problem definition.

Let's look at some bad examples. Consider this one:

We sill solve the problem of needing to become object-oriented.

We actually heard this one a lot from our clients in the 1980s. Why is it not a good problem definition? Because it's not at all apparent that there is even a problem: a difference between the current state and a desired state. This is a *solution*, not a *problem*! We can get a hint that this is not a good problem definition by asking *Why?* five times. In fact, if we do that, one of the answers may point the direction to a good problem definition.

Here is another one:

To be the best we can be at delivering on time.

What's the problem with this one? The result isn't measurable (a shortcoming of the preceding one as well). The problem will never be solved, almost by design.

Here is yet a third one:

We need to increase productivity 40% while cutting costs 40%.

This is more like a requirement list than a problem statement. First, it is really two problems, not one. Second, it may be a solution in disguise. Ask *why* we need to hit these targets. Third, it may set up an over-constrained problem if productivity and cost are linked.

Sometimes asking *Why?* five times can lead you from a poor problem definition to a good one. Consider this one as a starting point:

Our quality sucks.

Well, O.K., there must be something more we can say. We ask *Why?* The answer comes back "Because users report a lot of bugs." We could even make that into a problem statement:

Users report more bugs per release than our competition does.

*Why?* we ask, anticipating that the discussion will take a turn into testing. "Because users complain that they go all the way through our ordering and payment screens before they find that an item is

---

out-of-stock.” Aha. Maybe we are missing some scenarios in our Use Cases and we need to go back and ask for more User Stories from the end users. Maybe our problem statement ends up being:

The system does not properly handle end-user orders for out-of-stock items.

That’s a problem definition that the team can get its teeth into.

No matter what your title or organizational level, and no matter why you are creating a problem definition, it always pays to ask *Why?* five times. Get beyond the symptoms to the problem, and take ownership of the problem by articulating it yourself.

---

#### 4.4 Problems and solutions

If you think that *problem* is to *solution* like *cause* is to *effect*, you’re probably in for some surprises during design. The relationship between problem and solution is rich and complex. Broad experience with methods has shown that, in fact, you can’t start with a problem definition and methodically elaborate it into a solution.<sup>2</sup>

There are two simple facts to remember when administering your problems and solutions. The first one is that the mapping from problems to solutions is many-to-many. Consider a house that has a room with poor light. Furthermore, there are rooms in the house with poor summer ventilation. A single solution—a window in the dark room—might be enough to solve both problems. My grandfather always loved it when he could kill two birds with one stone. However, you should also be aware that four or five seemingly unrelated solutions might be required to solve what you perceive as a single problem. Some problems, like world hunger or world peace, seem to defy any mapping at all. The problem that you face in soft-

---

<sup>2</sup> The reference [Cro1984] is a good source on this topic.

ware design has intrinsic complexity in the same family of problems as world peace and world hunger: they're called wicked problems. The comparison is a perhaps a bit dramatic, but it is a formally apt comparison. There are no proven escapes from this difficulty except constant attentiveness and adjustment. It's just enough to keep one humble. But such adjustment is also what Agile is all about.

---

## 4.5 *The process around problem definitions*

Your initial problem definitions might take a bit of work.

Much of Agile—and Scrum in particular—is based on a fact of life called emergent requirements. My grandfather used to say that the best laid plans of mice and men often go astray. You can't master plan a project and expect to follow the plan. Agile folks know that. However, most Agile folks think that means only that we discover new problems along the way. It isn't just that we discover new problems: the act of design actually creates new problems. More to the point, the act of design sometimes changes the very nature of the problem we have set out to solve. We must revisit the problem definition to refresh it now and then.

### 4.5.1 **Value the hunt over the prize**

The underlying Agile value here is people and communication. There is an old French saying: "The hunt is more valuable than the prize." It is much more important that the team is in dialogue, discussing the problem identity, than that the final problem definition is perfect. Any given definition is potentially ephemeral, anyhow, and the value comes from dialogue. Problem definition expert Phil Fuhrer says it well [Fuh2008]:

Just the effort of trying to crystallize the client's needs into a statement that speaks to the designers is worthwhile even if the output is not finalized. It is about communication.

---

A good or great problem definition is harder to characterize. I would say that a good problem definition is one that leads to a successful project. It must fit project's product and process. Most of the problem with problem defining is that it is hard to aligning the defining effort with the project's process culture. Project managers often like to manage clear-cut deliverables and tend to rush the chartering, scoping, and other get-started tasks.

Measuring how well it captures the critical success factors of the client or how well it focuses the design effort is itself a problem. The four considerations (function, form, economy, and time) are helpful but I have seen problem definition efforts get hung up on them.

Having said that I would say that a great problem definition opens up possibilities and identifies and addresses overly constrained problems.

It is still crucial to drive toward a single, simple, closed-form problem definition; otherwise, you end up with analysis paralysis. Phil mentions another key concern: over-constrained problems. It's easy to define a problem that is impossible to solve, such as is often the case with space/time tradeoffs, cost/schedule tradeoffs, build/buy tradeoffs—in fact, just about all design decisions are tradeoffs and therefore open up the opportunity for conflict between desiderata. Great design is finding just the right solution that lets you have your cake and eat it too, but competent design is realizing when the business just won't allow such magic, owning up to that realization, and making hard decisions based on the consequences of that realization.

#### 4.5.2 Problem Ownership

Jerry Weinberg is articulate on the topic of problem ownership. It is a simple concept but is commonly misconstrued. The question

should arise: *Who owns the problem?* The answer should always respect the people with the power to solve the problem.

It's common in life that one person formulates a problem and hands it over to someone else to solve. You ask your secretary to get rid of a salesman. Your boss asks you to cut your budget by 15%. The customer asks you to fix a bug. While these situations will persist in the real world, it's better if the person who owns (or who will own) the problem writes the problem definition. Otherwise, problem statements become a way for one person to wield power over another, and that constrains the self-organization (and feedback!) that make Agile work.

As a stopgap measure, anyone can receive a "request to fix something" that originated somewhere along the lines of power in the organization, re-write it as a good problem definition, and feed it back to the requestor. Such feedback ensures that you are solving the right problem, that you together understand what the solution criteria are (they are measurable), and that the problem doesn't have a built-in trap that will lead to failure. That is a way to use problem definitions reactively.

In a true Agile organization, the team strives to expand the scope of problem ownership so that the problem definition opens up possibilities rather than focusing on how to allocate blame. This means that those responsible for solving the problem have a part in defining the problem. Problem statements shouldn't be a way for one person to wield power over another, but should help channel the energy of the organization in a consistent direction. A problem definition has more power if used proactively than if used reactively. It won't always work out that way, but keep striving to expand the scope of problem ownership. Keep it simple, fast, and light.

### *An Example of Problem Ownership: Scrum*

In Scrum, the Product Owner owns the problem of sustaining the ROI vision and meeting ROI targets. The Team supports the Product Owner in solving this problem by delivering product in the order specified by the Product Owner. The Team owns the problem of converting Product Backlog Items (PBIs, or requirements) into product, and the Product Owner supports the team with enabling specifications and ongoing clarification of requirements. The ScrumMaster owns the problem of improving the culture and the process, and supports the Team by working impediments that prevent them from solving their problems. The ScrumMaster owns a list of problems called the *impediment list*, one of the main artifacts supporting process improvement in Scrum. Taking away someone else's problem ownership (e.g. by taking over their problem) is disempowering and de-motivating.

#### **4.5.3 Creeping featurism**

Though problem definitions evolve, it is important to avoid slippery slopes and creeping featurism. There is always a tradeoff between being able to take on new problems and to be able to keep your commitments for solving the ones already on the table. All kinds of red flags should go up when a latent requirement, emergent requirement, or other surprise substitutes something new for something you are working on. Scrum nicely solves this by giving the team the option of tackling the newly formulated problem or rejecting it, on the basis of keeping a time-boxed commitment to what the customer wants. If the business can't live with the team's decision, then the new problem becomes a business crisis suitable for discussion at the business level [TODO: CITE]. Lean views such crises as a positive thing that draws the team onward and upward. It is usually important to convene a ceremony of the pertinent size and scope

when the problem definition shifts; see the organizational patterns TAKE NO SMALL SLIPS and RECOMMITMENT MEETING in [CopHar2004].

---

#### 4.6 *Problem definitions, goals, charters, visions, and objectives*

Agile is full of casual terminology for this issue of what-direction-are-we-going. We urge teams to distinguish between the following concepts, and we use the following words to describe them.

An *objective* is a waypoint that we must achieve to succeed. To not achieve an agreed objective is call for reflection and process improvement. The collective contents of a Sprint backlog form an objective for the Sprint. A problem definition is most often a form of objective. We can compare objectives to other important concepts that draw us forward.

A *vision* is a broad, inspiring portrait of a world that our system can help to create. It might relate to broad convenience that results from adopting our product; it might relate to increased profits for the company; it might relate to growing market share.

A *goal* is the desired endpoint in the best of all possible worlds. In Use Cases, the goal is what the main success scenario achieves. If we have a Use Case called “Phone Call Origination,” the goal is to talk to the party we are calling. Sometimes we don’t achieve that goal—because our friend is busy, or because the system is overloaded, or because we forget the phone number in the middle of dialing. Yet all these scenarios are part of the “Phone Call Origination” Use Case and each works *toward* the same goal. My grandfather always had the goal of completing his mail delivery route by 3:30 in the afternoon. Sometimes he made that goal, and sometimes he didn’t. That he sometimes didn’t make that goal doesn’t mean that he was a failure.

A *Sprint goal* in Scrum is usually more closely tied to “Done” than in the broader use of the term *goal*, so a Sprint goal is really an objective most of the time. Therefore, a Sprint goal can conveniently be described as a problem definition.

A *charter* is a document that describes the ongoing ways of working for a group. A charter usually comes from a chartering organization to a chartered organization, though the chartered organization can have a say in the charter's content. Charters very easily challenge Agile foundations. I recently looked at a charter document template created by a well-known facilitator; the boilerplate itself was [TODO X] pages long. That's not lean. Furthermore, the us-and-them notion of charter-er and charter-ee breaks down the notion of *team* that is crucial to effective communication in Agile approaches such as Scrum.

---

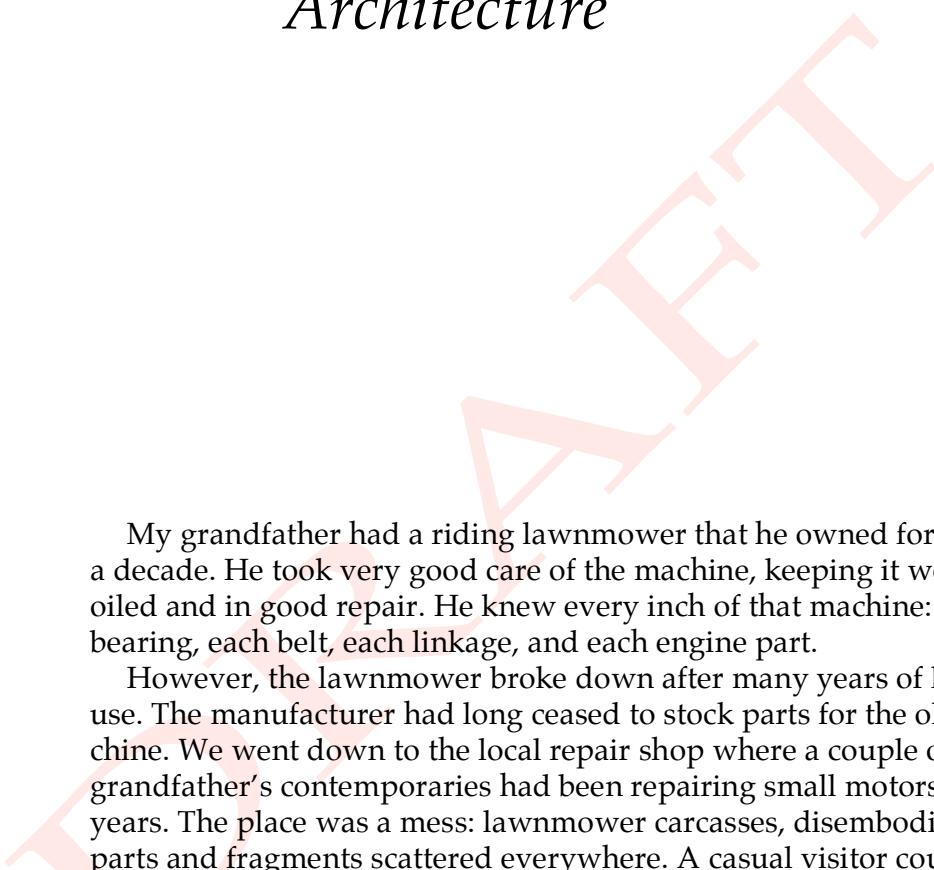
## 4.7 Documentation?

It's a good idea to circulate your problem definition broadly in written form. Producing a tangible artifact provides focus for the team and can bring together what is initially just a group of people and provide the seed for them to become a team. But remember that the main value isn't in the document and that it's not lean to produce something unnecessarily large. One or two sentences is ideal, and a page is almost starting to get too long.

There is something strangely powerful about documenting problem definitions on paper. Don't bury them as a field on some methodological form deep in some database. We'd discourage you even from using E-mail as the primary distribution mechanism. Try printing your problem definitions on small pieces of paper (small is beautiful, and emphasizes the non-methodological tone of a good problem definition) and hand them out to all the stakeholders. Have all the stakeholders autograph a copy on a major project. Make it fun, not a death march. Use problem statements to open up possibilities.

DRAFT

## *What the System Is: Lean Architecture*



My grandfather had a riding lawnmower that he owned for about a decade. He took very good care of the machine, keeping it well oiled and in good repair. He knew every inch of that machine: each bearing, each belt, each linkage, and each engine part.

However, the lawnmower broke down after many years of loving use. The manufacturer had long ceased to stock parts for the old machine. We went down to the local repair shop where a couple of my grandfather's contemporaries had been repairing small motors for years. The place was a mess: lawnmower carcasses, disembodied parts and fragments scattered everywhere. A casual visitor could see neither rhyme nor reason to the arrangement of parts in that shop; indeed, other than in XXX's memory, there probably was none.

But, yes, they had the part, and XXX worked his way gradually toward the treasure, moving obstacles large and small along the way. He came back to my grandfather, part in hand, and sent us on our merry way after we paid a token fee for the almost-antique piece of gadgetry.

The simplest way to maintain software is to create it from scratch every time. A program is only a delivery vehicle: the real end deliverable is a service, not a product. The service changes over time and the fact that we have a product—the code—is a liability. Code is not Lean.

Like it or not, code becomes an investment, and it's unreasonable to recreate significant works of software from scratch every time we need a change. There must be some order to the artifacts we create so we can find our way around in them. If we are a team of one, working on a simple, single application, then we can treat our software like my grandfather treated his lawnmower. We can claim to know it all. If we work on large complex systems, we need some order to be able to find what we need when we need it. Furthermore, the order should stay more or less the same over time: if it changes few days, then the order doesn't really help us.

In this chapter we will help you create a Lean domain architecture for your system. Such an architecture can support Agile software development much better than a traditional, heavy-weight architecture can. The inputs to this architecture include well-informed experience of domain experts as well as end-user mental models. The process starts with simple partitioning and then proceeds to selecting a design style (paradigm) and coding it up. We'll take a short interlude into some fine points of object-oriented design that will provide a foundation for the *what-the-system-does* work coming up in CHAPTER 6 through CHAPTER 8. The output is code and documentation. We will create class interfaces, annotated with pre-conditions and post-conditions, supported by a domain dictionary and a short domain document for each business area.

Before we get into the real work of architecture, we're going to belabor you with important foundations and ideas that will make the work easier. This is an Agile book, and we'd rather give you a fishing pole than a fish. Sections 5.2 and 5.3 are fishing-pole stuff, and the remainder of the chapter talks more about technique.

---

## 5.1 Some Surprises about Architecture

*Architecture* is another one of those terms that everyone uses but which has a broad spectrum of meanings. The term came into software through Fred Brooks while at IBM, who one day asked Jerry Weinberg whether he thought that what architects did was a suitable metaphor for what we do in software, and Jerry agreed. Even in the field of urban design and building architecture, the title of “architect” has taken on a disproportionate sense of power and mastery ([Ryb1989, p. 9]; see Section 3.1). In this book we heed time-honored principles of architecture that may be a little bit different than you find in your culture or organization, and we want to avoid misunderstandings.

- *Architecture is about form, not structure.* In this chapter we strive toward an architecture which, though concretely expressed in code, communicates form without the clutter of structure. If we capture the form (including its associations and attributes) without expanding into full structure, we stay Lean. That means that we can scale better than in an architecture that goes into full-blown detail of structure.
- *Architecture is more about compression than abstraction.* Abstraction is “the process of considering something independently of its associations, attributes, or concrete accompaniments” [Dic2007]. In architecture we want to consider system entities *with* their associations and attributes! But we want to keep the architectural expression compact. We keep the architectural expression small by appealing to standards and the domain knowledge shared by the team. In the same sense that poetry is not abstract, but compressed, so is architecture: every word means something more than its common dictionary definition.
- *Most architecture is not about solving user problems.* That’s mainly in the what-the-system-does. The hard part of architecture is to express the forms of the business, and it is these forms in which user problems arise. Lean architecture, by going beyond cou-

pling and cohesion to the end user world model, brings architecture back into the value stream.

- *Architecture has both a static and dynamic component.* In this chapter we'll focus on that part of architecture that changes little over time, that form that comes from the structure of the domain. It is like the form of a great ballroom that over its lifetime will witness many balls and many dancers. We'll return to the dancers in CHAPTER 6, but for now we'll focus on building the environment suitable to whatever dance your system is bound to perform.
- *Architecture is everybody's job.* Too many enterprises leave the important task of architecture to the architect. In an Agile world based on stakeholder engagement and feedback, we invite everyone to the party. Doing so reduces waste, reduces the intervals that come with those review meetings that are scheduled so far in advance, and develops buy-in for the system design across the enterprise. This isn't to say that architecture can be done by just anybody; we'll insist on some of the attributes reminiscent of the architect role from your old process. But we also face the stark reality that it's becoming increasingly difficult to find just that single right person who can master-plan the system. Instead, we recognize that together we know more than any one of us.

### 5.1.1 What's Lean about this?

We started writing this book to relate useful architectural practices for an Agile project, as we felt that nature had left a vacuum to be filled. The more we put our thoughts to words, the more that we discovered that good Agile architectural practice might best be expressed in terms of Lean principles. In retrospect, that shouldn't have been a surprise. Lean in fact has little or nothing to do with automobiles and everything to do with product, with value, and with the people in the design, production and maintenance processes. Much of what passes for Agile in Scrum these days in fact

---

comes directly from Lean (in particular, from the paper by Takeuchi and Nonaka [TakNon1986]).

### *The place of thought*

Lean is based on “a culture of stopping or slowing down to get quality right the first time to enhance productivity in the long run” ([Lik2004], p. 38). Architecture comes from early and ongoing deliberation about the overall form of a system, and it comes out of patience that can put aside pressure for the most immediate results to give time to long-term productivity. Analogous to the quote from Liker, the goal is long-term productivity. Doing architecture now lays a foundation for more productive work later.

Getting “quality right the first time” is a goal to shoot for, not an objective which defines success and failure. To set an absolute definition of quality and to gauge success by whether we met that foreordained number or mandate is to pretend more control over the future than is humanly possible. We embrace the changes that arise from latent requirements, emergent requirements, and just stupid surprises, because that’s the real world. However, every moment we strive to do the best we can given the information at hand, knowing that the future will bring us more information.

The Lean notion of quality makes more sense when we think of the value stream, and of the notion of “pull” versus “push.” The end user “pulls” the product through the value stream from its raw materials to the delivered product.

All people things of the next section still apply as satisfying the Lean principle of “pull.”

### *Failure-proof constraints or “poka-yoke”*

My grandfather sometimes applied his woodworking skills to cabinet making. If he had a lot of duplicate cabinet doors or drawers to build, he would sometimes build a “jig” to guide the cutting or assembly of the parts. His cabinets were still hand-made craftsmanship, but he made tools to help him in the tedious repetitive tasks of

assembly. It was a good way to avoid the kind of stupid mistakes one can make when undertaking tedious, repetitive tasks. What he was doing was a simple form of the Lean concept of “poka yoke”—the idea of using a guide or jig that makes it almost impossible to put together an assembly incorrectly. It means “fail-proof” in Japanese. (They originally used the Japanese for “idiot-proof” but political correctness won out.)

Software architecture is a perfect reflection of the Lean concept of “poka yoke”. It guides the engineer (in the case of software, the feature programmer) to write code that “fits” with the rest of the system. “Poka yoke” is not a punishing constraint but a guide and a help.

“Poka yoke” is a good fit for Agile development. As team members work together during an iteration, it is possible for one developer to check in even a rough implementation of some feature while others work on other features, even within the same integration. Architectural firewalls protect the overall system form—a form that reflects longstanding experience and systems thinking. The structure stays more stable, which means less rework.

### *The Lean mantras of conservation, consistency and focus*

Architecture embodies several more lean principles. As mentioned above, it reduces rework. It provides an overall consistent system view: to reduce inconsistency is a central theme in Lean. It helps keep the team focused on the feature and its value during feature development by removing most of the need to worry about system form, and that keeps development flowing in the heat of change. But most directly, software architecture reflects an investment economy. Lean believes in short-term loss for long-term gain. Here “investment” is a better term than “loss.”

### 5.1.2 What's Agile about architecture?

#### *It's all about individuals and interactions*

Agile is all about “individuals and interactions over processes and tools.” So, in fact, is architecture! We ask people in our seminars, “Why do we do architecture?” The answer usually relates to coupling and cohesion [StMeCo1974], [YoCo1975] and other classic measures of good architecture. Why do we strive for those? Does our software run better if its modules have better cohesion and if they are better decoupled? In fact, one can argue that a good architecture slows the code down by adding layers of APIs and levels of indirection.

No, we value architecture for the sake of what it portends for individuals and interactions. My grandfather used to quote the old saw that “birds of a feather flock together,” so individuals group according to their domain expertise, or at least according to their domain responsibilities. Each of these little groups should have its own software artifact that it can manage and evolve with minimal interference from the outside—that means minimal interference from other little groups who each have their own artifacts. If we divide up the artifacts according to the domain expertise we find in these groups of individuals, we provide each team more autonomy in the long term. That allows each team to be more responsive (“embracing change over following a plan”). This is called Conway’s Law [Con1986]. We support Conway’s Law by delivering just enough architecture to shape the organizational structure.

It’s not just about the individuals on the team and their interactions. Concepts of architecture extend all the way to the end user. Jef Raskin tells us: The interface is the program [Ras2000]. More precisely, the concepts in the end user’s head extend all the way into the code. If these structures are the same, it closes the feedback loop of understanding between end user and programmer. This isomorphism is the key to Doug Englebart’s notion of the “direct manipulation metaphor:” that end users directly manipulate the software objects in the program, objects that should reflect the end user mental

model. No amount of interface sugarcoating can hide the deep structures of code. Just try using your favorite word processor to place a picture exactly where you want it in the middle of the paragraph. The underlying program structure wins out in spite of the best efforts of interface designers and user intuition. The organizational structure reflects the architecture; the architecture reflects the form of the domain; and the domain has its roots in the mental models of end users and other stakeholders. Architecture is the explicit artifact that aligns these views.

### *Past excesses*

Software architecture has a history of excesses that in part spurred a reaction called Agile. Architecture is famous for producing reams of documentation that no one reads. The CASE tools of the 1980s were particularly notorious for their ability to produce documentation even faster than human beings could—again, documentation that was often write-only.

But Agile is about “working software over comprehensive documentation.” We will strive for an architecture delivered as APIs and code rather than duplicating the information in documents. While the interface is the program, the code is the design. One of the heaviest costs of software development is the so-called “discovery cost:” knowing where to find the code for a particular business area, or trying to find the source of a fault. Comprehensive documentation of the system organization is one way to do it. But if the code is well-organized, we can let the code speak for itself instead. Code has formal properties that elude most documentation (for example, type conformance of interfaces) that make it even more valuable as a design document. Yes, there will be some documentation, too, but we’ll keep it Lean. After all, the Agile Manifesto doesn’t say to “eliminate documentation,” and Lean just admonishes us to make sure that the documentation feeds the value stream.

In the past, architecture not only produced an impressive mountain of artifacts but also took an inordinate amount of time. We have a client who takes 6 months to take a new requirement into produc-

tion, and much of that time is architecture work. If you look closely, much of the time is spent in writing and reviewing documents. One team member lamented that “many of the things we write down simply because they are true.” That is waste. With everybody, all together, all at once, we eliminate these delays. It isn’t unreasonable to compress these six months down to one or two weeks if communication between team members is great.

### *Dispelling a couple of Agile myths*

The Agile world is full of practices that are reactions to these past excesses. Sometimes, these are over-reactions that go too far. Here we briefly look at two common failure modes in Agile projects.

The first belief is that you can always re-factor your way to a better architecture. While this is true in small degree, particularly for very small projects, it becomes increasingly difficult with the scale of the system and size of the organization. Time hardens the interfaces and the entire system slowly hardens. It is like stirring a batch of cement: eventually, it sets, and you can’t stir it any more.<sup>3</sup>

Re-factoring shows up as a practice in its own right, done for the sake of clean code [Mar2009] and as a key practice of Test-Driven Development (TDD) [Bec2002]. In its original form, TDD was a design technique for programmers based on unit-test-first. It grew out of distaste for big up-front architecture, and proposed to displace such practices with incremental architecture evolution. It claimed that it would improve the coupling and cohesion metrics. However, empirical studies don’t bear this out. A 2008 IEEE Software article, while reporting the local benefits of re-factored code, found that their research results didn’t support TDD’s coupling and cohesion claims [JanSal2008]. Research by Sinialto and Abrahamsson concludes not only that there is no architectural benefit, but that TDD may cause the architecture to deteriorate [SinAbr2007a], [SinAbr2007b]. This link from re-factoring to architecture arises from a belief that form follows function. That may be true, but it is true only

<sup>3</sup> Thanks to Ian Graham for this delightful image.

over the aggregation of hundreds or thousands of functions over the system lifetime. This link also presumes that re-factoring's scope is broad enough to straighten out system-level relationships. However, re-factoring is almost always a local activity, whereas architecture is a global property. There is a serious mismatch between the two.

A second common belief in Agile is that we should do things at the last responsible moment [TODO: COHN]. It's a bit of a problematic formulation because one never knows exactly when it has slipped past. It might be better said that we shouldn't make decisions at an irresponsible early moment.

What makes a decision irresponsible? We get in trouble when we don't have enough insight to support the decision. Postponing decisions increases the time for learning to take place and for requirements to emerge, and that is the argument for deferral. But a deferred decision entails more than the passage of time: we aren't just sitting in the sun waiting for things to happen, but are doing work and committing to structure in the code. Leaving time go by as we create structure in the mean time, without conscious attentiveness to form, leads to undisciplined form. We may learn something in the process of creating it, but we have also left a trail of work that must be unwound and re-done. Therefore, we must balance between an approach where we unearth and act on fundamental knowledge early, and one where we allow knowledge to emerge from compounded local, short-sighted actions.

The fundamental form of a business often repeats itself in system after system. This constancy means that we know much of it at the beginning of every project. To defer the decision of what essential domain form to lay as the system's foundation is irresponsible because it has a high chance of creating waste. Therefore, we embrace domain knowledge—stuff that we truly know about the business—at the beginning of product construction.

## 5.2 *The first design step: Partitioning*

My grandfather always said that you put your pants on one leg at a time. As human beings, we are psychologically wired with a feature called a locus of attention that is closely tied to the notion of consciousness: we have exactly one of these and, at some level, we can focus on only one thing at a time. Our first inclination as human beings to deal with complexity is to divide and conquer.

There's an old saw that a complex system has many architectures. There's a lot of truth to that. If you think of the old-fashioned top-down approach to system design, it worked well for simple systems. Top-down design viewed a system as providing some function, where we can use the term "function" almost in the mathematical sense. A program was to take an input, transform it, and produce an output. Maybe it did this many times, once for each of thousands of punch cards provided to it as input in what was called batch processing. The function could be broken down into sub-functions, and those functions into smaller functions, and so forth.

Another way of thinking about the complexity is to think of it in terms of *classification* rather than partitioning. Classification is a technique we use to partition items into distinct sets. There are of course many forms of classification. Think of classifying the items in the room you are sitting in right now: by color? By size? By use? By age? There are many different classification schemes you could use. The same is true with software. The problem is that though one classification scheme may seem best, many more are at least partly right.

Top-down design breaks down for complex systems because a complex system has many "tops."<sup>4</sup> People don't use today's interactive, Agile systems in batch mode: their keystrokes and mouse clicks come in seemingly random order. End users juggle between "tops" in a session with the program. Picasso was said to do his oil paintings in an analogous way, jumping from one area of the picture to another rather than dwelling first on one part and then on another. (When doing our early work in the 1980s with the multi-window

---

<sup>4</sup> Thanks to Dennis DeBruler for this insight.

work station called the **blit**<sup>5</sup> in Bell Laboratories, we found that one of the main uses of multiple windows was to maintain simultaneous views of these multiple “tops.”) A good program presents many tops clearly, and a good architecture expresses them elegantly.

Let’s investigate some of the important “tops” of software, and further investigate ways to use them to partition a system.

### 5.2.1 The first partition: domain form versus behavioral form

Agile software development looks at software the same way, and the Agile Manifesto contrasts embracing change with following a plan. A timeless goal of software engineering has been to separate the code that changes frequently from the code that is stable. For most software, that dividing line is the same as the division between what the system *is* (that is relatively stable), and what the system *does* (that changes as customer expectations change and grow). In some sense, those are two different “tops” of the system. Most software architectures exhibit this split in one form or another. For example, client/server systems often load up the client with most of the “does” code and the server with the “is” code. Our first step towards an Agile architecture is therefore:

*Technique 1: Focus on the essence of the system form (what the system is) without being unduly influenced by the functionality that the system provides (what the system does).*

There is no explicit design activity where we separate the what-the-system-does part from the what-the-system-is part. Rather, there are two parallel, cooperating development streams that focus on these respective areas. The what-the-system-is component often precedes the what-the-system-does component, because you often have a development team in place before you have enough behavioral requirements to enable design and development. Of course, it happens

---

<sup>5</sup> “Blit” does *not* stand for Bell Labs Intelligent Terminal.

the other way around, too: a customer arrives on your doorstep and asks for some new software service, and you already have a start on your what-the-system-does requirements.

This partitioning owes its roots to a lot of history, including the basic way that computers work (the Von Neumann computational model), the way we are raised in the Western world (to believe in a dichotomy between form and function), and all the programming languages and tools that precipitate from those two foundations.

### 5.2.2 The second partitioning: Conway's Law

Given this architectural context we can now actually get to work. As mentioned earlier, we have two design processes going on: one for what-the-system-is, and another for what-the-system-does. We'll focus more on the latter in CHAPTER 6; here, we'll focus on what-the-system-is.

This second partitioning is both one of the least formal and most important steps of system architecture. Stated broadly and simply, it is:

*Technique 2: Focus on the form of what the system is, partition it so that each part can be managed as autonomously as possible.*

This recommendation is a consequence of something called Conway's Law [Con1986]. It seems to be a law of software systems that the form of the product looks a lot like the form of the organization that built it. It's not clear which is cause or which is effect in general, but the organization structure usually precedes design, so the organizational structure usually drives the architecture. This isn't a good thing or a bad thing; it just *is*, kind of like the law of gravity. And this technique doesn't exclude looking at what the system *does*; it's just that we'll come back to that later.

These partitions are sometimes called *subsystems*. This partitioning doesn't look very important to the nerds because of its low-tech and intuitive nature. In fact, it is a largely administrative partitioning and

has only incidental ties to the business structure (although the administrative and business structures often align for reasons of history or convenience, which is a good thing when you can get it). Such partitioning is nonetheless crucial to the success of the enterprise in accordance with Lean and Agile principles. It minimizes how much information must pass back and forth between locations. It supports more effective interaction between team members, because they share mutual interests, and can talk across a table instead of across the Internet. (Yeah, it's fun to talk across the Internet but, believe it or not, face-to-face communication is usually more effective.)

Autonomy goes hand-in-hand with an important principle of the Agile Manifesto: embracing change. We want to organize our software so that common changes in the market and elsewhere in the "real world" can each be dealt with as a local change, inside one of the subsystems. Subsystems are a gross form of modularity. So when we talk about autonomy in the long term, think of it in terms of change.

*Technique 3: The dominant consideration in supporting team autonomy is how locally common changes are handled.*

Notice that we don't tell you to partition using objects, or modules, or rules, or database relations, or any specific methodology. The partitioning criteria should follow history, standards and convention, experience and common sense. How do you know if it's right? Think of the teams that will work on the software, and do the partitioning in a way that allows each team to work as independently as possible.

Remember, too, that architecture is mainly about people, and an Agile perspective helps bring that fact into focus. Think a bit about how your customers and even end users expect your system to be organized from a business perspective. Use their vocabulary whenever it makes sense and organize in a way that supports shared communication and understanding. For large systems, this initial ac-

---

tivity of partitioning probably won't go very deep into the end user's cognitive model of their business and workflow; that is a more prominent concern of structuring, which we will address in Section 5.3.

A time-honored software engineering measure of success is that each subsystem is as cohesive as possible, and is as de-coupled as possible from the other subsystems. Coupling and cohesion are defined in terms of the amount of work it takes to change something. [StMeCo1974] Tight coupling between two subsystems isn't a serious problem if neither subsystem changes much. The challenges arise from change, so you should always be thinking dynamically when choosing a partitioning.

Of course we'll get to code (very soon) and we'll have to decide on an implementation technique (objects, or modules, or rules, or database relations, etc.) An Agile approach honors these four rules of thumb:

1. We use design paradigms (object-oriented, modular, rule-based, databases) to best support the autonomy of teams in the long term and to reflect the end-user mental model of the system.
2. A complex system might use several paradigms.
3. The paradigm we use is ideally subordinate to, and supports, the partitioning based on autonomy of teams in the long term and the end-user mental model.
4. The object paradigm was consciously designed to meet these goals and it will usually drive the dominant partitioning.

The key notion here is #3: Strive to let the human issues drive the partitioning with an eye to the technological issues such as coupling and cohesion, depth of inheritance hierarchies, the Laws of Demeter [Lie1996], and so forth. It is sometimes too easy for nerds to get caught up in their educational degrees, their religious zeal for a design technique, or in a misplaced trust in the formalism of a type system or "formally" designed language.

*Technique 4: Let the human considerations drive the partitioning, with software engineering concerns secondary.*

Sometimes you are given an over-constrained problem (see CHAPTER 4) like “come up with software for a high-quality, automatic exposure camera that is compact, lightweight, easy to use, 30% cheaper than existing single-lens reflex cameras and which is written in Java.” The Java stipulation limits your partitioning options, and that will certainly influence your system partitioning. While not an ideal situation, it’s a common one. In Agile software development we inspect and adapt. Don’t look too hard for a fixed set of rules and guidelines; rather, follow experience and the deeper principles of design, balancing things as best as you can to optimize the value stream.

That’s partitioning in a nutshell. Let’s dig deeper into principles related to partitioning—principles that can help you reason about particularly complex systems and special cases.

### 5.2.3 The real complexity of partitioning

If you are building a simple product, it is easy to partition the system based on the rule of thumb of long-term team autonomy. Again, simple means “not complex,” and complexity is proportional to the number of distinct, meaningful “tops” of the system. We can talk about these tops from a purely technological perspective, but that’s not interesting, and it deals with only part of what is a much larger complex system. The organizational structure is also part of that system. If the two major reasons for architecture are to support the organizational structure and the end-user mental model, then we should pay careful attention to those.

Let’s say that a company called ElectroCard produces printed circuit-card layouts for clients’ electronic circuits. These circuits typically require three to twenty circuit cards to implement. ElectroCard software is to be an Agile application that has a GUI through which

an ElectroCard engineer can guide the circuit card construction. The founder of the corporation has hit upon a fantastic opportunity. He has found a group in Hangzhou, China that knows how to group logic gates into packages available in commercial integrated circuits (chips), and how best to put chips together on a card to minimize the number of connections between cards. (This is called *chip placement* or just *placement* in the industry.) And he has found a group in New Jersey that has expertise in algorithms that automatically route the connections between chips on a board. (This is called *routing* in the industry.) Both groups have practical knowledge in human-assisted algorithms using a GUI. The question is: what are the major organizational components of the architecture?

If the founder wants to optimize group autonomy in the long term, then there should be one architectural component for placement and another for routing. Each would have its own GUI, its own notion of card and its own notion of chip. You might object and say that it's obvious that there should be a common GUI and common libraries for common components. However, that design doesn't optimize group autonomy over time. It might reduce code duplication, and therefore reduce rework in the long term, but it would require tight coordination between groups. Is it worth it? It might be a tradeoff between group autonomy and code duplication! That's a *business decision*.

If the founder asked the teams to create an architecture organized around the two functions of placement and routing, it wouldn't be an unreasonable request. Let's step back a bit and look at the broader landscape of design. This is in fact a complex problem—one with many “tops,” and choosing different tops optimizes different business goals.

#### 5.2.4 Dimensions of complexity

Let's start with a simple case. Your development team is collocated, you have one customer, and you're in the same business as ElectroCard. Now what does your architecture look like?

Instead of focusing on how to divide work by geographic location, you now create an architecture that will allow an individual, or a small group of individuals working together, to focus on one architectural component at a time with as much long-term autonomy as possible. What bits do you group together? As described above, you let the partitioning follow history, standards and convention, experience and common sense. Over time, subject matter experts notice repeated patterns in system after system. These patterns tend to follow recognized birds-of-a-feather areas. We call these areas *domains*. Domains commonly (but not always) reflect the end-user mental model of the domain. In some whimsical sense, domains are the mythological foundation of a given business that one finds rooted in culture or society.

### 5.2.5 Domains: a particularly interesting partitioning

A domain is a business area of focus, interest, study, and/or specialization. It is an area for which a body of knowledge exists. Sometimes this is just a tacit body of knowledge, but it is nonetheless a body of knowledge. All other things aside, domains are the primary “tops” of a system.

What might the domains be in our circuit card application? The human-guided nature of the product suggests that there is a domain for interactively editing circuits. The chips themselves form a domain: chips have functionality, size, power ratings and configurations of connections that form a body of knowledge relevant to the product. Both routing and placement are traditional domains in electronic design automation. Given that each of these is a body of knowledge, it is likely that we can find someone with training, experience, or special knowledge in *each* of these areas. We might even find small groups of such people, birds of a feather, who share such expertise in a small company. The most knowledgeable of them we call *domain experts*, and their area of knowledge is their domain.

Domain knowledge is one of the most distinguishing factors in making sound design decisions. Domain knowledge is a distillation of experience garnered from past systems, competitors’ and part-

ners' systems, and in general from being in touch with the area of discourse. Codified domains capture design decisions about years of tradeoffs between multiple stakeholders, including multiple communities of end users. These tradeoffs become refined over time to minimize the long-term cost of change. Therefore, if we want to give each group as much long-term autonomy as possible over its software, we form the organizational structure around the domain structure, and we organize the architecture accordingly. That's a key part of Conway's Law. The *Organizational Patterns* book [CopHar2008] describes this as a key pattern of software development organization.

So we add this to our list of techniques:

*Technique 5: Be attentive to domain partitioning. In particular, don't split a domain across geographic locations or across architectural units.*

What makes domains particularly interesting is that they commonly designate sets of closely related product variants grouped together in a *product line*. A product line closely corresponds to what most people would recognize as software reuse. A base set of software is common to several separately delivered variants. The source code in the base is large relative to the changes necessary for a given variant. Parameterization, selective use of derived classes with small snippets of custom code (using design patterns such as TEMPLATE METHOD [Ga+2005]), or even conditional compilation constructs (such as `#ifdef` in C++) can be used to configure individual variants.

Think about Conway's Law again. If domains encapsulate product variants, then their structure corresponds either to the business structure or to the structure of some market. Just on basic management principles, you *want* your organizational structure to reflect the structure of the business. If we structure organizations around domains, then by Conway's Law we have aligned the business structure, the organizational structure, and the architecture! That leads us to another tip:

*Technique 6: Be attentive to the opportunity to use product lines and use this insight to bolster support for domain partitioning where possible.*

Families often show up in the code as instances of generics or templates or as a collection of derived classes that make small refinements on a common base class. We'll discuss these issues more in Section 5.3.

### 5.2.6 Back to dimensions of complexity

We encountered two organizational patterns above. The first is called CONWAY'S LAW, which suggests that the code structure and the organizational structure be analogous to each other. If all members of a development team are collocated then product development has all the freedom it needs to put together the best organization. The Lean Secret is based on getting everybody together, but geographic distribution constrains our ability to do that. (As we'll see later, even when you do the "best" for a collocated team doesn't necessarily mean that the team or architecture are "ideal," because most products have built-on over-constrained problems.) But if team members are geographically distant it limits the organization's ability to mix and match individuals to teams. True team dynamics are found in true teams, and a true team is a collocated group of seven plus or minus two people working together under a common goal. In the end, CONWAY'S LAW says that the organizational structure and software structure should follow each other, but there are other patterns that constrain the formation of the teams.

In another pattern we found that the organizational structure followed geographic locations: one in New Jersey and one in China. That pattern is ORGANIZATION FOLLOWS LOCATION [CopHar2008]. That constrains the composition of true teams, since members of an ideal team are all drawn from the same location. If we want to give each group as much autonomy over its architectural units as possible in the long term, then we create architectural units that map onto the

locations. Note that these two organizational structures, and the resulting architectures, are potentially in conflict. This is what we mean about complexity arising from multiple “tops;” there is no single reasonable top-down (hierarchical) partitioning that could satisfy both organizations. This is a form of over-constrained problem. There is at least one more common “top” that arises when selling products into complex markets, and that is the structure of the market itself. ElectroCard has a special client in France that wants a French language interface. In theory, each chip needs to know how to display its signal names in French; the overall command structure for partitioning and placement should also be in French. Now we have a French product. Ideally we would have a team dedicated to that product. Maybe even more ideally, the team would be located in France, preferably in one of the Rhine wine regions. This is yet another organizational pattern: ORGANIZATION FOLLOWS MARKET [CoPhar2008].

The fact is that all three of these patterns have the power to shape the organization and, therefore, the architecture. Yet we also want cross-functional teams that bring together the talents and insights necessary to keep moving forward. What, then, is the guiding light to a good architecture?

Consider Figure 10. Assume that unless otherwise constrained by another staffing policy, we try to create cross-functional teams. The chart looks at some consequences of multi-site development and how they relate to specific architectural choices. To build a single, “clean” organization, and expect optimal results, is unrealistic.

Primary Architectural Structure	Pattern	Positive Consequence	Liability
Modules organized by business domain	Each location is staffed around a business domain	Good independence for general development	Special interests (target markets) may suffer
	Each location is staffed to support some market	Very few – chances for coordination within a location are random	General development requires heavy coordination across locations
Modules organized around markets	Each location is staffed around a business domain	Very few – chances for coordination within a location are random	Common development requires heavy coordination across locations
	Each location is staffed to support some market	Great responsiveness for local markets	General development requires heavy coordination
Modules organized by solution domain (programming language, etc.)	Each location is staffed around a solution domain	Reduces need to duplicate tools across location	Just about everything else
	Each location is staffed to support some market	Very few – chances for coordination within a location are random	Very few – chances for coordination within a location are random

**Figure 10: Organizational (and therefore architectural) drivers**

In the end, let the principles and common sense be your guides. One of the most important considerations is encapsulating change, so look at common changes and ask how many domain boundaries each one would have to cross. That is the same as the number of domain teams that would have to coordinate the change. If that number is worrisome then seek another partitioning that encapsulates change better.

Remember that if you have the freedom to staff teams (including teams at different locations) however you want, much of this complexity vanishes. Organizational freedom removes many of the constraints that arise when trying to align the organization, the domains, and the markets into a single architectural structure. There is

---

no single formula. Again: it's about the principles of long-term local autonomy, and common sense.

One common solution is to organize the architecture primarily by business domain knowledge (remember Technique #5), and to build teams around domain knowledge. This leads to an organization where most of the coordination take place within the software of a given domain, and the team members can coordinate with each other within the same location and time zone. In addition to these primary partitions, there can be additional partitions for each market that requires special focus. Those partitions are likely to have less than ideal coupling to the domain-based partitions, but cross-cutting techniques like AOP can help on the technical side. In any case, these points of interference can architecturally be made explicit as APIs.

Another boon of this ideal organization is that geographically isolated staff can work on the market-specific partitions. You can form groups around these market concerns and their realization in the architecture. Such groups are commonly located close to markets that align with geography. For example, a team could be placed in Japan to do the localization for the Japanese market. That not only meets the architectural objectives but also makes it easier to engage the key stakeholders for that architectural concern: the targeted end-users for that market.

Figure 11 is a stylized example of how you might map the architecture onto an existing organization. It looks ahead to the next step: dividing the subsystems into modules according to domains. There are development teams in North and South America, as well as in Japan. The domains are divided among the American teams subject to the rule that no domain has strong coupling to more than one location. In South America two different teams work on Domain Module 2. This isn't ideal because it requires coordination between the two South American teams. However, because of the Lean Secret and because the teams are collocated we can quickly bring together the right people to address any coordination issues that arise in Domain Module 2.

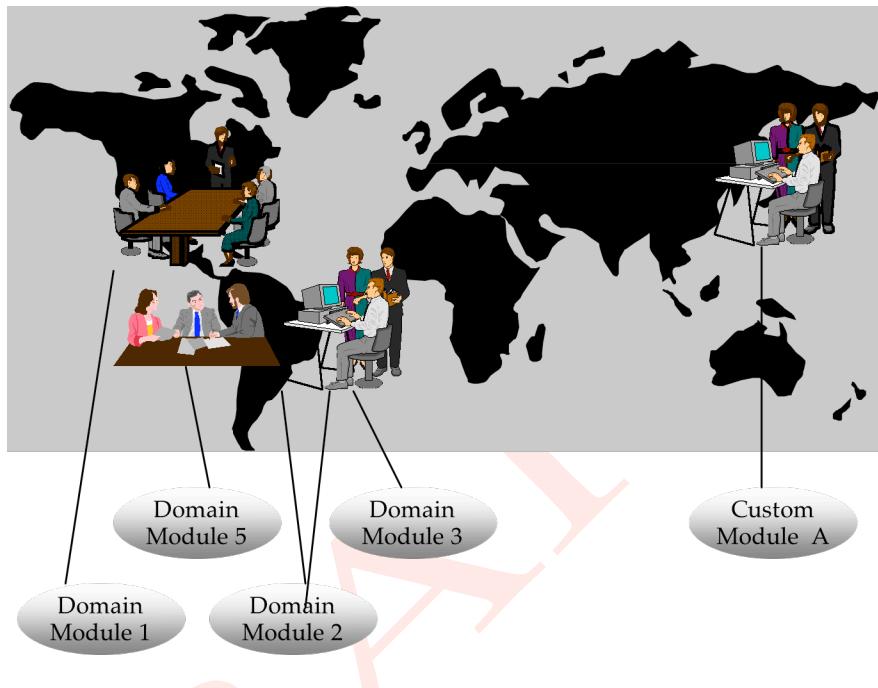


Figure 11: A typical organization-to-architecture mapping

### 5.2.7 Wrap-up on Conway's Law

It isn't enough to just say that one will optimize ROI and to quickly make a choice based on that: it's important to understand the dynamics of product evolution and to use that understanding to drive the architectural form. *That's embracing change!*

---

### 5.3 The second design step: selecting a design style

Attention, nerds! We're finally past all that people stuff and we're going to start using the kind of vocabulary that you might have expected from an architecture book, or a book on Agile development—which can perhaps whimsically be portrayed as the nerds' revenge. This section is about selecting the right design technique and about using it to find the right modules. By "design technique" we mean design or programming paradigm: object-oriented, procedural, generic and generative programming, databases, and the like.

Most nerds will think of this as selecting a technology. However, object-orientation isn't really a technology, nor are most of the other approaches that arise here or in the broader literature. (For a suitable definition of technology, these things *are* technology: Alan Kay says that we use the term "technology" for anything that was invented since we were born, and Danny Hillis says that we use it for "the stuff that doesn't really work yet." [Bra1999, p. 16]) Perhaps more in the spirit of classical architecture we use the term "design style" instead. In computer-ese we use the term *paradigm*. A computing paradigm is a set of rules and tools that relate to patterning or modeling. To select a design style is to select a paradigm.

From an Agile perspective in particular, much of design and architecture is about communication and, in part, the expressiveness of the code. The code should be able to capture and express the mental model of the end user and other stakeholders such as domain experts. The nice thing about paradigms is that we have programming languages whose syntax and semantics express their organizing power, and we can seek the right set of semantics to fit stakeholder semantics. That means choosing the right programming language feature and, sometimes, the right programming language or other "technology." We choose the design style that best fits the situation.

Almost every software paradigm has two important features. First, each one is a way to group related items by commonality. Our minds are good at noticing patterns of commonality among things in our environment, and software paradigms cater to that instinct and its place in language. What is in your hands? A book. It happens to

be a particular book with a particular ISBN and a particular history, made of a particular set of atoms unlike any other book in the world. But for sake of communication we call it “a book”, going into the differentiating characteristics when in a discourse about more than one of them. Architecture is both about these commonalities and the way in which items can vary, and paradigm serves us well.

Second, most paradigms encapsulate frequent changes. This is true even for the procedural paradigm, which gets a bad rap for coupling and cohesion. On old UNIX® systems, the `sort (3)` library function actually comprised three algorithms, and it selected the proper internal algorithm according to properties of the data it was given. The maintainer of `sort` might have added a fourth, but because the `sort` function encapsulated it we wouldn’t have noticed. That keeps the overall form—the architecture—stable even under a substantial change. In Agile, we meet change on the street and we embrace it. An architecture-conscious approach embraces change by encapsulating it.

### 5.3.1 Contrasting structuring with partitioning

A naïve view of structuring is as the next step of breaking down subsystems into smaller parts. To a degree that is true. We said that a good subsystem partitioning makes change easier by localizing common changes. The same is true at the structuring level. A paradigm helps us break down subsystems into modules according to rules of organization and grouping based on commonality. If any one of the major common kinds of change arises in a modular system, then ideally all of the work to accommodate that change can take place within a single module. A system has many modules, each one of which accommodates a common class of change or growth in requirements. This is, in fact, Parnas’ original definition of the term module: that it hides a “design secret” that can be changed internally without instigating a ripple effect across the rest of the code. [Par1978] Change is how we generate new revenues; if we can keep the cost of change low, we increase our profitability. Localizing change lowers cost and makes programming more fun.

---

Also, as is true for subsystems, we can use coupling and cohesion as a rule of thumb for how well the module structure is serving us. As with subsystems, this must be a dynamic model that considers how well the system handles change—that is, what fraction of changes can be handled locally.

However, there are important differences between the gross partitioning into subsystems and the finer partitioning into modules that go beyond difference in scale. Subsystems are largely administrative, while modules have a necessary relationship to business semantics. *Modules are the most direct expression of the end user's mental model of the entities in their business world.*

Most of the time, change within a module doesn't mean that we throw the whole module away and replace it with another one. In fact, major parts of good modules stay constant over time. So each module is a mixture of stuff that changes a lot, and stuff that stays the same a lot. There can be many loci of change in a system—i.e., many modules—at multiple levels of granularity.

At the very top level of the system, the what-the-system-does software structure changes much more often than the what-the-system-is structure. That is our top level of classification, or partitioning (Technique #1). At the next level we partition according to our intuition and experience of how subsystems will map onto birds of a feather. It is a rather unsophisticated process, and though we analyzed it formally from a business perspective in Section 5.2, it most often follows intuition and domain insight. It is taking the things in our business world and just putting them in baskets, creating new baskets as needed. It is simple classification with no transformation. Partitioning is largely an act of analysis; the only design in partitioning comes from the decision of what classification scheme to use. But there are elements of design here, too, since we are forging boundaries that will last into the implementation. Almost by definition, whatever affects the implementation (the solution) entails design. As a practical issue analysis can never be clinically separated from design. Just as a building architect knows his or her building materials, a software architect knows the strengths and weaknesses of the programming languages and environments that lay ahead.

Structuring refines these partitions according to some design style, according to some paradigm. Within each partition we analyze what aspects change frequently and what aspects are more stable over time. Consider our ElectroCard example, where we have made *placement* to be one of the domains. The placement group in China may find that the algorithms have a rhythm that remains constant over time, except for one set of algorithms that are sensitive to the hardware technology being used (they support several different chip packaging technologies including dual in-line pin chips (DIP chips), surface mount components, discrete components, and others). In each of these cases, we start with the result of a business classification and end with an additional level of separation between stable parts and changing parts.

Structuring moves from *what* is in the business to *how* we will structure it in the product. We are now beyond analysis alone and firmly in design. Domain experts have a key role, since much of their experience relates to how such structuring was done in the past. Using the principle of “yesterday’s weather,” the design we have used in the past is often as good or better than what we can come up with now. (My grandfather used to say that it was more reliable to predict that today’s weather would continue tomorrow than it was to read the weather forecast in the newspaper.) We are still in contact with the end users so we can choose a paradigm that expresses the domain in a way that even they might find natural. We have the business people on the team because issues of scope inevitably arise. Customers may want to configure certain domains for inclusion or exclusion from the product for specific markets. Programmers will soon find ways to express these forms in the code, so solution domain expertise becomes crucial at this point.

### 5.3.2 The fundamentals of style: commonality and variation

The structuring tasks in design build on the deepest building blocks of human cognition: being able to distinguish what is common from what changes.

---

To a domain analysis person or someone working on software families, commonality is something that exists *at any point in time* across software modules, and we use the term *invariant* to describe similarities across modules, and *variation* to describe how they differ. Simple object-oriented design is a good example: a base class captures what is common to all the modules (all the derived classes) and we capture the changes in the derived classes themselves. We can create a derived class object by “changing” the structure of a base class object according to the modifications explicit in the derived class. (Here, by derived class, we mean the source code of the derived class—the part that contains only the delta from the base class. That’s how most object-oriented programming languages express such differences.) The Lean angle on this is that it takes planning to understand how much variation to tolerate in a system at a given time.

However, to an Agilist, change is something that happens to a given artifact (such as object structure) *over time*. To be common over time is to (tend to) be *stable* over time. We use the terms *evolution*, *change in version*, or sometimes just *change* to describe the differences over time. Again, strangely enough, we can use the same example as before: in a style of object-oriented programming called programming by difference, Bertrand Meyer points out that you can evolve programs over time largely through subclassing ([Mey1994], ch. 14)

In the end, it doesn’t matter: change is change, and we have programming languages and other tools that raise the expression of change to the level of objects and classes that encourage and enable us to use everyday business vocabulary instead of computer-ese. We can call a `SavingsAccount` a `SavingsAccount`, and we can find it in the code. We can talk about the differences between `SavingsAccounts` and `CheckingAccounts` in the code as well. If it is architecturally important to talk about the difference between the `SavingsAccount` in the 2010 release and the `SavingsAccount` after the new banking laws passed in 2011, then we can express that in the code too. To the degree that domain experts anticipate such changes, such changes can be modularized. In this case, maybe we can use class derivation to express all of these changes well.

There is a common misconception (based on a grain of truth) that commonality is the purview of data structure and of what-the-system-is, and that variation lives in the algorithms and functions of what-the-system-does. This would be literally true if the data were invisible to the market, and if all we visibly sold was function. How often were you told in your programming class that you should think of a Shape object in terms of its behaviors, and that the data structure was left to the choice of the implementer? And that the important attributed that changed from Shape to Shape was the methods (e.g. for rotation, which is different for a circle than for a rectangle)? If this were literally true, we could divide the design between these two worlds (as we already have in Section 5.2.1) and we'd be done with architecture.

It may be that data structures are more stable over time than algorithms, but data structures have variation as well. We probably use different internal representations for circles than for rectangles. Architectures should encapsulate both kinds changes in a way that preserves the overall form as much as possible. Most of what we discuss in this chapter aims toward that objective: allowing changes to what-the-system-is without upsetting the architectural form. You know this in several common guises: for example, class boundaries and inheritance can hide data changes from a user of the base class API.

### 5.3.3 Starting with tacit commonality and variation

Pretend that we had a magic, universal paradigm, one that would allow us to create modules that would always encapsulate change as well as possible. What would that paradigm look like? It would capture the commonalities inside of a module and would allow the interface of the module to express the variations. Unfortunately, the world is a complex and sometimes messy place and we don't quite have such a single magic paradigm. The closest thing we have is classification by commonality and variation.

The good news is that we *can* tap into such a magic paradigm, albeit in an indirect way. When we look for the basic objects that users care about, we don't ask them to group things according to specific

---

attributes of commonality and variation, but we follow business intuition. What drives this intuition is the way we think about associations between things in the world. Our brain is very good at finding patterns (in the vernacular sense of the word, not the software design sense, though the two are related). A pattern in our mind captures some common properties of a recurring thing along with some understanding of the variation within the recurrence. You know ice cream as ice cream because of its temperature and texture—that’s what makes ice cream ice cream. However, chocolate and pistachio ice cream taste very much different, and our mind still classifies both of them as ice cream in spite of the fact that the primary end-user sense elicited by the eating-ice cream Use Case—that of taste—would cause us to classify them differently. Yet our mind captures the common pattern of ice cream being basically creamy and cold, and tucks away information that different kinds taste differently.

Perhaps the most creative part of software design is to partition the domain into units of decomposition that encapsulate change. These units of decomposition (and later, of composition) are called *modules*. This task requires a fine touch because this partitioning expresses so many important design desideratae at once. First, each module should correspond to the end user’s cognitive model of their world. Second, each module should be as independent from the others as possible. Third, each module should be cohesive. Fourth, Conway’s Law comes to bear: can you create modules so that they encapsulate the domain expertise that you find in a team or in an individual domain expert, so that it doesn’t take a village to raise a module?

Because we can talk so explicitly about commonality and variation in the software world, it would be nice to use the same tools in the analysis world as we extract the end user mental models. It would be nice to go to end users and ask, “What things in your cognitive model line up well along commonalities in data structure and have variation over time in the algorithms that implement the services you envision for those things?” If we can find consistent mappings from the end-user space to our software paradigms, then we’ll get an architecture that works—that encapsulates change well. Of

course it doesn't work that way. Ultimately we want to get to those commonalities and variations in our architecture, but we don't start there.

Fortunately, culture and human experience are often kind to us and gives us a historic perspective on what works and what doesn't. This knowledge is in the heads of your subject matter experts, your *domain experts*.

*Technique 7: Allow the module partitioning to follow domain knowledge. Think of domain knowledge as a timeless compression of the mental models of end users and other stakeholders—mental models whose patterns are tacitly driven by commonality and variation.*

Experience shows that this partitioning best captures the invariant and stable parts of the system. This isn't something that formally can be proven. On the other hand, it almost stands as a tautology: To be a domain means to have the property of being invariant across your markets or stable over time. In the end, a good architecture doesn't reduce to computer science principles, but to your ability to connect to the human end of your system and to distill that knowledge as domain expertise.

Sometimes, you just can't get the domain knowledge. Maybe it's a greenfield domain, or maybe you have a novice team. It happens. In that case, we go back to the principles of Agile:

*Technique 8: In the absence of domain knowledge, allow the module partitioning to follow the end user cognitive model of the domain. For every user "notion" create a corresponding architectural notion.*

The end-user's model of the moment, driven by current requirements, may be blind to the bigger picture. It's the nature of a modern project to want the software to work now. The question is: how long is *now*? If you don't have domain knowledge, you are stuck with the end users' sense of immediacy. Even though we have separated out the what-the-system-is part of the end user model from the domain

---

model, the current feature may color the end user's view of the domain. If we form the architecture around the end-user models of the moment, they are likely to be different on the next release or even next week. Even the best user experience people can't compensate for this shortsightedness.

The compensation must come from a historical perspective. Domain knowledge integrates many past *nows* into a "long *now*". [Bra1999] That's why it's better to form modules around domain knowledge than around the current end-user view. However, sometimes the end-user view is all you have. This is particularly common for new features. Use domain knowledge when you can get it, and the current end-user mental model when you can't.

### 5.3.4 Commonality, variation and scope

A software system may have hundreds of millions of entities. For example, a large business system may support tens of millions or hundreds of millions of accounts. (Think of your national tax system as an example.) Even though you may be a very productive programmer, you can't write that much code. So we use the commonality across these entities to *compress* the design.

We noted in the introduction that architecture is hard, and that one of the reasons it's hard is because of such complexity. This complexity comes both from the *mass* of data that we have, and from the intricacies of *relationships* between the data. Architecture is about slicing away as much of the mass of data as we can while maintaining the relationships. While compressing down the mass of data, we want to retain its essence, or form.

There are two ways of extracting the essence of a thing or set of things. I can focus on the single most important feature and put aside the rest. The essence of Cyrano de Bergerac is his nose (if we ignore his poetic talent). The essence of a savings account, to the end user, is to deposit, and withdraw and get interest (if we ignore how the actuaries view it). To put aside features of some entity while retaining others is called *abstraction*.

The other alternative is to *generalize* the overall business characterization of the thing or set of things. Generalization always takes place with respect to some *context*—some set of shared assumptions. We get these shared assumptions from culture or from deep knowledge of our business: domain expertise. It is like abstraction because in fact I am throwing away information—if a general description is more compact than the original, something has to go! But what I remove from the description will be the tacit, non-distinguishing properties that anyone familiar with the domain would take for granted. So, for example, if I’m generalizing the concept of *bicycle* I don’t need to explicitly mention the property of having two wheels. That information is in fact encoded in the name itself: the prefix *bi-* implies “two.” That is part of the business *context*, its shared assumptions.

Because the context lives on in domain knowledge, we lose no information in generalizing this way; we *do* lose information while abstracting. Too much computer science pedagogy these days emphasizes abstraction. Dick Gabriel admonishes us about the dangers of abstraction in his classic essay, *Abstraction Descant* [TODO]. He points out that the shared domain knowledge or cultural knowledge is a kind of “code,” and that generalizing this way is a form of *compression* that uses the domain knowledge for the encoding and decoding. It is that compression, using tacit shared information, that makes it compact.

*Technique 9: Capture your initial architecture at the most compressed level of expression that covers the scope of your business. Avoid abstraction or the discarding of any priceless information that may give insight on the form.*

Let’s look at a trivial example. Let us ask you to envision the interface `ComplexNumber` in your head. Now consider the complex numbers  $(1, -1)$  and  $(1, 1)$ . Let’s say that I created objects of those two complex numbers, and multiplied one by the other to produce a third. What is the result? You knew the answer without having to look at the code of `ComplexNumber`, and without even having to

know whether it used an internal representation of  $(r, \theta)$  or *(real, imaginary)*. You probably used your cultural knowledge, particularly if you are an engineer, to recognize the shorthand form as indicating real and imaginary parts. And you used your cultural knowledge of what a complex number is to do the math.

Compression is Lean, in part because it reduces the waste of writing down information that is common knowledge to the business. There is of course a great danger here: that you have team members who aren't on board to the domain yet. [TODO: how to fix that]

To compress effectively, you need a context, and the context is often delineated by a business scope. Who determines the scope? This is a business decision, and comes from the Business stakeholders (Section 3.1.2). Scope is tricky. For the banking account example: are both savings and checking accounts part of the same business? It depends how you view them: do you group them as having enough commonality that you view them as the same? Or do the variations over-power the commonalities so that they are different? These insights in fact can lead to an even higher-level partitioning:

*Technique 10: In fact, the initial partitioning is to create decoupled businesses whose concerns can be separated from each other.*

These businesses can of course cooperate with each other, use each others' services, or share large amounts of common code. So-called infrastructure organizations can view other internal projects as end users, and can produce a software artifact that is shared across several organizations. There are of course serious challenges with this approach and we discuss those in Section TODO. (*500-pound gorilla problem, being truly general, having good enough communication with collocation if the coupling is tight enough, otherwise as an OEM, etc.*)

Once this high-level partitioning is done you can work with what you have and find the top-level, most compressed level of expression. Such expressions will become your top-level architecture. The fact is that we compress all the time: we don't take the design all the

way to machine code, and register transfers, and bus design, and hardware architecture, and gate level design, and solid state design of the transistors, nor the electrons and holes across a PN junction in a chip, nor down to the Fermi levels of the electrons in the silicon and isotope atoms of the devices. We cut it off at some level. You might argue that this is abstraction; fair enough, we won't take the word away from you. We don't view it as abstraction because, in fact, all that stuff is *there* already. We compress our understanding of something already built, something that exists, to use it: the dangers in doing so are minor (because there is an oracle we can go to if there are any arguments). If we abstract our understanding of some domain and use the resulting information to build something that does not yet exist, we are likely to miss something important. That will hurt evolution. To embrace change requires the discipline of understanding what is likely to change, and what will likely remain stable.

In summary, we deal with complexity in three ways. We use partitioning to understand a complex system a piece at a time; partitioning groups things by their *common* attributes. We've already addressed partitioning in Section 5.2. We understand things that already exist by abstracting; we try to avoid that in architecture because the artifact doesn't exist yet. And last, we use compression to generalize complex systems, building on *common* domain knowledge and on the *common* properties of the concept under discussion. This information in hand, we are one step closer to making it explicit in the code.

### 5.3.5 Making the commonalities and variations explicit

A user usually has more than one flavor of a concept in mind at once. A bank account holder may talk about accounts, but in fact might think concretely of savings and checking accounts. Electro-Card clients may think of printed-circuit cards, but in fact there are different kinds of cards: surface-mount technology, discrete components, and others. In all of these cases, the differences matter. However, the differences are qualitatively different than the differences between domains. What makes them different is that these elements

---

are bound by some invariance. Savings and checking accounts have remarkably similar places in the architecture and remarkably similar algorithms (reporting a balance, making a deposit, etc.) All printed circuit cards share basic principles of routing.

Each account, and each routing algorithm, is a module in its own right, yet these modules have a special relationship to each other: their commonalities. Programming languages are good at expressing invariants like these, as well as the variations.

To the end user with a long view, and also to the programmer, we can view change over time the same way. We have today's savings account with its relatively stable form, but we also have its evolution into tomorrow's savings account that reflects changes in the way we give interest to the customer. Eventually, we might have both in the system at once and give end user a selection of interest-bearing accounts. Over time we have a succession of modules that maintain a large degree of stability.

We use the term *commonality* both for the invariance across concurrent options and stability over time. All of the variant modules are members of the same domain. So we can talk about a domain of accounts that includes savings and checking accounts—both today's and tomorrow's. We can talk about a domain of routing algorithms that includes Hightower routing and Lee routing—both for today's chip-based boards and tomorrow's surface-mount boards. These domains are the same ones driven by the end-user model or by domain expertise. It's just that we're broadening our scope a little bit.

The interesting question of what-the-system-is architecture is: *how* are these modules similar? Said another way: what is their commonality? We can answer the question in business terms, as we did in the preceding paragraph. Instead, we will start to think about how we will express these commonalities in code. This transformation from business conceptualization, to program conceptualization, might best be called *design*. It is a (probably tacit) activity you do whenever you do software design, whether using procedural design or object-oriented design.

Before jumping to procedures and objects and templates and other paradigms, we're going to keep it simple at first. We also want

to distance ourselves from programming language and other implementation biases. We look at the basic building blocks of form as computing has understood them for decades. We call them *commonality categories*. [Cop1998] These building blocks are:

- Behaviors, often communicated as names (external behavior: e.g., to *rotate* (a name) or *move* (another name) a Shape)
- Algorithms (the sequence implementing the behavior: e.g., the algorithm for *moving* or *rotating* a shape)
- State
- Data structure
- Types (in the vernacular programming language sense)

That's just about it. These are the criteria that we seem to use when grouping software. We tend to look for the ways in which one of these properties stands out as a *common* feature across concepts in our mental models, and then form a partitioning or grouping around that commonality. We also look for how these same properties delineate individual concepts in our groupings.

This list is important because it seems to characterize the groupings that Von Neumann programming languages express. (By Von Neumann languages, we mean those that tend to express the form of software written in some Von Neumann computational model. This includes paradigms such as the procedural paradigm, the object paradigm, and modular programming, but is less applicable to rule-based programming, or the functional programming style of spread sheets or of Scala's methods, for example.) Most popular programming languages of the past 50 years have drawn on this list for the kinds of commonality and variation that they express. It is a remarkably small list. In some sense, this list forms a simple, "universal paradigm" of design. If we can identify the commonalities and variations in the domain model and end-user model, we can use this list as a powerful tool for translation to the code.

---

Let's illustrate with some examples. Think again of savings and checking accounts. To the end user, they are all the same underneath—it's just money. The end user doesn't care very much about where the money is kept. The programmer using a `SavingsAccount` object or a `CheckingAccount` object doesn't care how the objects represent the money: whether they hold the amount as a data field or whether they partner with a set of transaction auditing objects to keep track of it. To the end user, it's an amount; to the programmer, this suggests a commonality in *data structure*. The *state* of that data structure varies over time. How do savings and checking accounts vary? What would make me choose one over the other is *behavior*. It is clumsier to *withdraw* (the name of a behavior) from a savings account than a checking account: there are different Use Cases, or algorithms, elicited in this same general behavior. Most savings accounts draw interest; many checking accounts don't. Most checking accounts have annual or per-use fees; savings accounts don't. Those are variations, and each one can be exemplified in a Use Case. They are behaviors.

So you're in the banking business. You want to divide your business into autonomous segments. Do checking and savings have enough in common that they belong together in the same business? I compress checking and savings accounts into a notion called Consumer Account and conclude: yes, I want a line of business called Consumer Banking. Of course, I can't do this on the basis of two account types alone or even on the basis of accounts alone, but the concept generalizes. Alternatively, you can start by asserting that I have a line of business called Consumer Banking (a scope decision made by the business) and then ask the question of how to organize these account types into an architecture. The answer is: *according to some paradigm that can express commonality in structure and variation in state of the data and in the behaviors associated with the data*.

How about our guys at ElectroCard? They have two basic routing algorithms based on the classic Lee algorithm and the Hightower algorithm. These algorithms take a graph data structure as input and produce a multi-layer topology as output. (Think of the output as a geographic map of lands and seas that will be used to manufacture a

circuit card. The lands (yes, they really use that term) will be manufactured in copper to connect electronic devices together, and the seas are a kind of plastic that separates the copper areas from each other.) The algorithms both have the same business *behavior*: they route lands on a circuit board. However, the Lee algorithm and Hightower *algorithms* are different. Furthermore, there are more refined differences pertaining to the differences in board technology (surface mount versus DIPs), but these are differences in *behavior*.

In both cases, we can use this knowledge to refine our understanding of partitioning and to choose suitable paradigms. We'll talk about that in the next two sections.

### ***Next steps***

Sometimes, in a complex domain, even the domain experts may not agree on what the most fundamental organizing modules are! We talk a bit about that in Section 5.6 below. But, first, let's get into more of the nitty-gritty of structuring.

#### **5.3.6 The most common style: object orientation**

Object orientation has probably been the dominant design style (paradigm) on the planet for the past 15 or 20 years. That's a long time in Internet years. What makes it so enduring? Arguments abound. One of the often-heard arguments in the 1980s was that the object paradigm is "natural" for the way human beings think. A slightly more cynical view is that people associate computing with the Internet, the Internet with Java, and Java with object orientation. Another is that C++ rode the wave of the then-ubiquitous C language and for reasons of fashion led to the widespread use of the term "object-oriented."

Perhaps the most persuasive arguments come from the *direct manipulation metaphor*. The Smalltalk programming language was taking root at Xerox's Palo Alto Research Center (PARC) laboratory in the same era that interactive interfaces were coming into vogue there, and about the same time that Doug Englebart invented the

mouse. Dahl and Nygaard, who invented object-oriented programming in their Simula67 programming language, had the notion that objects should reflect the end user mental model. The object-oriented Smalltalk people wanted to extend that notion to the user interface in a way that the end user had the experience of directly manipulating the objects of their domain—objects that had a living representation or kind of proxy living in the memory of the machine. The end user should have the feeling of manipulating the objects directly. Brenda Laurel powerfully explores this metaphor in her book, *Computers as Theatre* [Lau1993]. The point is that there is a long tradition not only in object orientation, but also in interactive computing, of linking together the end user mental model with the structures in the software.

In the end, we don't care which of these rationales is the “right” one. They all ring true and, in any case, object orientation dominates contemporary interactive software. So much for the *why* of object orientation. But next,...

### ***Just what is object orientation?***

This has been a much-debated question with quite a few final answers. Object-oriented programming started with the Simula 67 programming language. Its inventors, Ole Dahl and Kristin Nygaard, viewed it as a way to capture the end user's mental model in the code. Such foundations are ideal for system simulations, which was what Simula was designed for. The vision of Dahl and Nygaard bridged the gap between the user world and the programmer world.

In 1987, one of main organs of object orientation—the ACM OOPSLA conference—published a “treaty” that unified competing definitions for object orientation at the time. The paper was called “The Treaty of Orlando” after the OOPSLA venue where the discussions behind the paper had taken place. [StLiUn1989] The paper defines object orientation mainly from the perspective of substitutability, which can be achieved using a number of programming mechanisms such as templates (the idea that a class keeps all of its objects consistent at some level) and empathy (which is more applicable to

classless languages like self, where two objects share a relationship analogous to the subclass relationship of class-ful languages). There is an important concept lurking here, which is the introduction of the notion of class into the design space, which we'll revisit numerous times in this book. The object view is the Agile, end-user-centric view; often, the programmers have a class view of the same architecture.

If we view the current activity (of CHAPTER 5) as “technology selection,” then we view object orientation as a “technology.” We want to know when object-orientation is the right match for the domain model or end-user mental model. There should be a good match if both express the same commonality and variation. If the domain model visualizes market segment differentiation or variation over time as requiring changes to algorithms, without having to modify the existing data structures, then the object paradigm may be a good fit.

You know obvious examples from your basic education in object orientation. The perfunctory OO example of geometric shapes—circles, rectangles, and squares—is based on a set of variants that share common data members (such as their center and angle of rotation) and that vary in the algorithms that implement behaviors such as rotation, scaling, and area computation. It is unlikely that you thought of geometric shapes in terms of this derivation from commonality and variation. But the analysis works well.

More often than not, the modules in Agile programs end at the same place: object-oriented programming. In fact,

*Technique 11: Most domains in simple, interactive applications lead to modules that are implemented using object-based or object-oriented programming. This is particularly true for entities that the program presents on an interactive interface for direct manipulation by users.*

In a very general sense the end user is himself or herself one of the objects in the system. We have yet to investigate other key architectural components that tie together the wetware of the end user with

the software written by the designer. We'll do that with the interaction framework called Model-View-Controller-User (emphasizing that the User is also an object—in fact, the most important one). We implement Model-View-Controller-User with the familiar architectural pattern Model-View-Controller, or MVC for short. That leads us to another important technique:

*Technique 12: The object structures in the what-the-system-is part of the architecture will become part of the Model in a Model-View-Controller(-User) architecture.*

We'll cover that idea more in CHAPTER 7 and CHAPTER 8.

There is one thing more, and we hinted about it at the beginning of this chapter. End users conceptualize the world in *objects*, whereas programmers are frequently stuck using programming constructs called *classes*. If we look solely at commonality and variation—and particularly the commonality of data structure and behavior and the variation of algorithm—the only difference between objects and classes is that groups of otherwise similar objects can vary in their state and identity. We'll see later that this simplistic model of object orientation doesn't fit the end user model of the world as well as it could because we're missing the notion of role: objects play roles according to what's going on at the time. This "going on at the time" necessarily evokes system dynamics, and we can't express anything that dynamic in anything as static as a class. However, that knowledge lives in the what-the-system-does part of the system, and classes serve us just fine to express what the system *is* (at least when domain analysis points us in the direction of the object paradigm).

So far we've focused only on what the system *is*; we also must accommodate what the system *does*, which will be the topic of the following chapters. Because of Technique 12, most the remaining material in this book will assume that the object paradigm drives the primary shape of the what-the-system-is architecture: that's where much of the leverage comes from in an Agile system.

In the mean time, you shouldn't take it for granted that everything will be object-oriented. Take our ElectroCard case as an example. In the next section we investigate the more general approach to selecting paradigms.

### 5.3.7 Other styles within the Von Neumann world

As introduced in the previous section, object orientation is the prevalent paradigm in Agile projects. Even in Agile projects there is software that doesn't naturally align to the Agile values of frequent change or of close connection to the end user or customer, but you need it to support your business.

Because these bits of software are often out-of-sight and out-of-mind to the end user, you'll need to learn about them from your domain experts. Solution domain experts should be particularly high on your list for their insight both into the necessary supporting domains and for their knowledge of what design styles are most suitable for them. As is true of most software architecture (and of architecture in general), history, taste, and experience are excellent guides. Architecture is more art than science and it's important to defer to people and interactions over processes and tools in setting your architectural forms.

Nonetheless, commonality and variation help shape the architecture. You can use them to justify the selection of a given paradigm or design style in your code commentary or architecture documentation. You can use them to guide your exploration of new domains or of the cost of bending an area of your architecture to fit a new market. Most important, the split between what is common and stable, and what is variable and changing, supports the central role of architecture in embracing change. Analyzing the commonalities and variations can help keep your team members collectively honest as they choose paradigms, design styles, tools and programming languages to achieve objects for maintainability.

Commonality	Variability	Binding Time	Instantiation	C++ Feature
<b>Function Name and Semantics</b>	Anything other than algorithm structure	Source	N/a	Template
	Fine algorithm	Compile	N/a	#ifdef
	Fine or gross algorithm	Compile	N/a	Overloading
<b>Data Structure</b>	Value of State	Run time	Yes	Struct, simple types
	A small set of values	Run time	Yes	Enum
	Types, values and state	Source	Yes	Templates
<b>Related Operations and Some Structure</b>	Value of State	Source	No	Module
	Value of State	Source	Yes	struct, class
	Data structure and state	Compile	Optional	Inheritance
	Algorithm, Data Structure and State	Run	Optional	Inheritance with Virtual Functions

**Figure 12: Commonalities and variations for C++ language features**

Figure 12 shows common configurations of commonality and variation that easily can be expressed in the C++ programming language. Note that in addition to the commonality categories (function, name, data structure, etc.) we also take binding time and instantiation into account. These indicators together suggest unique programming language features that are well-suited to express the form of the business domain.

Commonality	Variability	Binding Time	Instantiation	C++ Feature
<b>Function Name and Semantics (must be within a class scope)</b>	Anything other than algorithm structure	Source	N/a	Generic
	Fine algorithm	Compile	N/a	#ifdef
	Fine or gross algorithm	Compile	N/a	Overloading (only of non-built-in operations)
<b>Data Structure</b>	Value of State	Run time	Yes	struct, simple types
	A small set of values	Run time	Yes	Enum
	(class) Types, values and state	Source	Yes	Generic
<b>Related Operations and Some Structure</b>	Value of State	Source	No	Module
	Value of State	Source	Yes	struct, class
	Data structure and state	Compile	Optional	Inheritance
	Algorithm, Data Structure and State	Run	Optional	Inheritance with Virtual Functions

**Figure 13: Commonalities and variations for Java language features**

Note the very last row of the table: If we have commonality in related operations, and potentially in at least some of the data structure, with variation in algorithm and data structure (and of course in the state of the data) with run-time binding and possible instantiation, you use inheritance with virtual functions—the C++ way to implement object-oriented programming.

Figure 13 shows the analogous relationships for Java. We show these graphs not to help you formalize your architectural decision process. Perhaps they can support or inform your common sense. More importantly, we want to demonstrate that architectural decisions must take the solution domain into account. Just as a builder or architect takes building materials into account while building a house, so should software designers take paradigms into account when shaping the initial forms of a program.

If we choose our paradigm carefully then it is easy to find where to make the kinds of changes that our analysis anticipated. In the object paradigm, this might mean changing the behavior of a method. The domain model is said to have *parameters of variation*, and by associating the right code (algorithm, class, value, etc.) with that parameter of variation we can generate different entities in the domain. Members of a domain are sometimes said to form a *family*, and the parameters of variation are like genes that we can turn on and off in the family's genetic code. Each complete set of parameters defines an individual *family member*. When we reduce the domain to code, the parameters of variation should be explicit. In the object paradigm, this might mean adding a derived class with a modified method. For generics or generative programming, it may mean changing a template parameter.

In this section we have overviewed domain analysis suitable to Lean architecture and Agile construction. If you want to explore these ideas in more depth, see either [Cop1998] or [Eva2003].

### 5.3.8 Domain-specific languages and application generators

So far we have discussed how to apply the results of commonality-and-variation-based domain analysis to a design using general-purpose languages. Some of the source code remains relatively stable, while the parameters of variation allow small changes, or provide well-defined hooks that connect with programmer-supplied customization code. Another approach to realizing a domain analysis is to create a first-class language that expresses the parameters of variation—not in some general-purpose syntax, but with the syntax

and semantics of the business itself. Such a language is called an *application-oriented language* or *domain-specific language*.

One of the original goals of domain engineering was to provide domain-specific languages that perfectly captured business semantics. The term “domain-specific language” (DSL) in the 1980s and 1990s meant a language designed from the ground up to support programming in a fairly narrow domain. More recently, the term has been co-opted to mean any domain-focused programming effort embedded in a general-purpose language. It pays to survey this landscape a bit, and to consider their place in your architecture.

### *The state of the art in DSLs*

It may be that the new association for the name is a reaction to the relative failure of the old one. True domain-specific languages are not very Agile because they encode commonalities and variations in a narrow, concrete expression of the business form. If the language is not perfectly designed, its lack of general-purpose programming features makes it difficult for the programmer to accommodate missing business concepts. Furthermore, if the domain evolves then the language must evolve with it—potentially leaving the previously written code base obsolete. There are a few efforts that have had the good fortune to succeed, but there have also been countless failures. The usual failure mode is that the language becomes brittle or awkward after a period of about five years. On the Øredev 2008 panel on domain-driven design, the panelists (Jim Coplien, Eric Evans, Kevlin Henney, and Randy Stafford) agreed that it is still premature to think of domain-specific languages as reliable production tools.  
[Øre2008]

There are, however, notable broad successes. Parser generators like **yacc** and **bison** are good examples.

There are other techniques that are related to the modern definition of DSLs and which have had moderate success in niche areas. One of these is generative programming, which raised the bar for embedding domain semantics in a general-purpose language in 2000 [EiCz2000].

### *DSLs' place in architecture*

A DSL smoothens the interface between the programmer and the code that he or she is writing. Unless the DSL is specifically designed to tie together architectural components in a general way (such as CORBA'S IDL [TODO: CITE NEEDED]) the DSL doesn't have much benefit for capturing the relationship of its domain to the rest of the software in the system. This latter concern is our dominant one in this chapter because we are concerned with managing the long-term interactions between domains, assuming that changes within a domain are encapsulated. A DSL provides the encapsulation (within the code provided by the domain engineering environment).

That means that for each DSL, you need to define an interface between the code produced by the domain engineering environment and the interfaces to the rest of the domains. The rest of the system should interface to the DSL through a single, relatively static interface instead of interfacing directly with the generated code—that's just basic attentiveness to coupling and cohesion.

*Technique 13: Provide an API to the code generated from a DSL for use by the rest of the architecture.*

Because you will not be managing the changes to the domain at that interface, the interface will probably not be built as the result of any long-term analysis of commonality and variation. Let common sense rule. If it makes sense for this interface to be a class, make it a class. If it makes sense for it to be a function, make it a function. Many DSL environments pre-define these interfaces: for example, **yacc** uses a combination of macros (for declaring the types of parse tree nodes and lexical tokens) and procedural interfaces APIs (for starting the parser and determining success or failure of the parse). Any other needed interface is up to the programmer.

### 5.3.9 Codified Forms: Pattern Languages

A pattern language is a collection of structures, called patterns, that describes their relationship to each other as they can be composed into a system. Individual patterns are elements of form that encapsulate *design* tradeoffs called forces, much as modules encapsulate the relationships between *implementation* procedures and data.

Patterns work best in an incremental fashion, applying one at a time as needs be. However, designers apply patterns in an order suggested by the pattern language: each pattern depends on a context provided by the patterns that were applied earlier. A pattern language, then, is in fact an architecture for some domain. A pattern language has the same level of freedom in the details as the programming paradigms we have discussed in this chapter, but it has the narrowness of application of a domain-specific language.

Pattern languages were popularized by an architect of the built world, Christopher Alexander, in work that has progressed from the late 1950s to the present time. The software community took notice of Alexander's work as early as 1968; one finds them mentioned at the first conference on software engineering. [NaRa1968] But they didn't start gaining a real foothold in software until the pattern conferences, called PLoPs (Pattern Languages of Programs) started taking place in 1994.

Most early patterns established a trend of capturing isolated structures of important but uncelebrated knowledge, a trend that in large part persists to this day. However, as parts of the community matured, pattern languages started to emerge. Some of the most noteworthy pattern languages of software architecture today include:

- The POSA series [Bu+1996], [ScStRo2000], [KiJa2004], [Bu-HeSc2007a], [BuHeSc2007b];
- Fault-tolerant software patterns [Ha2007];
- Model-View-Controller [Ree2003].

This list is meant to characterize some of the good available pattern languages but it is hardly complete.

Ideally, pattern languages are roadmaps for building entire systems or subsystems in some domain. They are an extremely Lean approach to development in that they build structure as the need for it arises. Patterns build on two fundamental approaches: local adaptation of form according to system needs, and incremental growth instead of lump-style development. Both of these are done in a way that preserves the form of the artifact as it has been developed so far, so a pattern is a form of *structure-preserving transformation*. They differ from the domain analysis approach in that they capture the architectural form in documentation rather than in the code, and in that their purpose is to create structure according to a form rather than to guide the creation of form (architecture) for its own sake.

A few pattern languages exist that can guide you through evolutionary architecture development using local adaptation and piece-meal growth. So we add:

*Technique 14: If you have a trustworthy pattern language for your domain, use it instead of domain analysis or end user input.*

More realistically, patterns suggest architectural styles that can selectively guide the structure of your system. MVC is a good example: we'll talk a lot more about MVC in CHAPTER 7.

There are actually few mature pattern languages in the public literature. Much of the use of pattern languages may lie within your own enterprise, growing over time as you use them to document architectural learning. Such documentation becomes a legacy (in the good sense of the word) for the next generation of product or product-builders. We'll discuss this issue more in the documentation section at the end of the chapter.

### 5.3.10 Third-party software and other paradigms

You might be asking: Where does my database management system fit? Good question! Software is complicated and often doesn't fit into a neat methodological framework. Software that you don't manage in-house, or which follows another design style, leads to the opportunity for a mismatch in the interface between your software and the foreign software. Such software fits into two major categories: ones whose APIs are already determined by standards or by a vendor, and frameworks that you build in-house. Each of them is reminiscent of the approaches we recommended for DSLs.

First, let's consider third-party software and software whose form is dictated by standards. Too few companies consider software outside their own four walls, and end re-inventing the wheel (and it's usually a much poorer wheel). If you manage architectural variation well, procurement of external software or partnering with vendors can be a huge business win. Do a cost/benefit analysis before assuming that you should build it yourself.

Databases are a good example. Occasional rhetoric to the contrary notwithstanding, database software is rarely within the values of Agile. Its vocabulary and dynamics are in terms of computer science constructs rather than end-user constructs, and database work is dominated more by tools (e.g., commercial database servers) and methods (e.g., formalized relational models) than by the Agile values. If an end user has to see SQL, they are really a programmer who is working in the framework of a relational (or network, or hierarchical) model rather than their cognitive model of the business, based on grouping by commonalities and variations. Nonetheless, database approaches are a good match for many domains, and the software has to be there.

There are several considerations competing for your attention: uniformity, autonomy, and suitability.

- *Uniformity:* You want to keep a small inventory of paradigms in your development shop. It's expensive to train everyone in multiple paradigms, to purchase (and maintain) the tool and

language support for multiple paradigms, and so-forth. You can wrap a module in a paradigm suitable to the dominant project style: e.g., by using the ADAPTER and PROXY design patterns [Ga+2005] to encapsulate the foreign software and its design style.

However, this approach has risks. Adding layers of transformation between the end user and the internal data model can lead to a mismatch between the end user's model of what is going on in the software and what *is* actually going on in the software. You can overcome that with an illusion, but illusions are difficult to sustain. Draw on deep domain knowledge based on what has worked well in the past, and otherwise view such encapsulation with skepticism.

The Lean ideal is to partner with your business suppliers to seek solutions that meet the needs of both partners. This is often a difficult ideal to sustain in the software world, but we have seen this approach succeed in some of our clients.

- *Autonomy:* If your vendor is remote, then it will be difficult to include their staff on a collocated cross-functional team. That means that any changes to the semantics of architectural interfaces require coordination across huge organizational gulfs. That goes against the very purpose of architecture. You can address this using two techniques:

*Technique 15: Leverage standards when dealing with third-party software.*

and

*Technique 16: Factor changes to third-party software into local parameters of variation or modules that are loosely coupled to the foreign software.*

Leveraging standards is a staple of Lean. Defacto standards are great; industry-wide standards are even better. The ideal interface between mutually foreign components should be a very thin pipe whose shape is dictated by the standard. By “thin,” we mean that it doesn’t know much about the business semantics. Using our database standard again, SQL is a thin pipe that stands entirely ignorant of any domain semantics. Even SQL statements embedded in Java or C# code can maintain this nature of a thin pipe.

Keeping the changes within your own shop reduces the coordination between mutually disjoint teams. Good frameworks have well-defined points where you manage parameters of variation. You can pull all the “parameters” (whether simple integers or entire blocks of code) within your own organization. For example, if you maintain the database design in the same shop where the users of the database work, then you can coordinate your SQL query changes with the DB admins who change the relations once in a while. You have the flexibility of organizing as a cross-functional team. In the mean time, standards keep the interface to the database server itself stable.

- *Suitability:* You want the design style to fit your domain model and the end user model. Maybe you’re building a client-management system whose architecture is patterned after a notion of “client objects” and “market objects” that arise from analysis. You can gain a huge business advantage by tapping into existing relational databases of clients. Which paradigm do you choose? There are no universal answers, but widespread experience offers a lot of hope that these paradigms can enjoy a happy marriage. If you can encapsulate the database-ness of the implementation inside the client and market objects you get the best of both worlds. Again, this means that these objects must maintain a bit of illusion. Experienced programmers are up to the challenge, and you can use many of the same techniques mentioned under *Uniformity* just above.

If your software uses an Internet service or other remote API, you can treat it much the same as third-party software.

As for such software that you build in-house, it's more or less the same song, second verse. Leverage standards in the design of such software where possible. For example, it would be unwise to re-invent an SQL clone as an interface to a DBMS that you develop in-house. Keeping to standards affords you the option of migrating to a standard server should you need to do that for reasons of scaling, cost reduction, or some other business forcing function.

---

## 5.4 *The rough framing: Delivering the First Code*

It's finally time to write code. The Agile people out there have been screaming, "what's with all this documentation? We want some *code!*" The Agile people will have their revenge when we come to the what-the-system-does chapter; here' we're endearing ourselves to the Lean perspective.

And, in fact, this section is a little anticlimactic. The code should be *really* lean. We'll start with the basics and then investigate add-ons little by little.

From a nerd-centric perspective, the following technique is perhaps one of the two most important statements in the book:

*Technique 17: The essence of "Lean" in Lean architecture is to take careful, well-considered analysis and distill it into APIs written in everyday programming languages.*

That means that the product produced by the architecture activity of the ElectroCard routing team might be these lines of code:

```
#include "BoardLayout.h"
class CircuitGraph;

extern BoardLayout
route(const CircuitGraph theCircuit);
```

Remember the goals of Lean:

- To avoid producing artifacts without end-user value;
- To deliberate carefully before making a decision, and then to act decisively;
- To create processes and an environment that reduce rework;
- Last bullet.

Lean architecture avoids producing wasteful artifacts simply by avoiding any artifact. In the simple version of route above, we don't produce a function body. That's isn't form; that's structure. Its details aren't fundamental to what the system *is*; they focus on what the system *does*.

But that doesn't mean that it is a casual declaration. In a way, we feel cheated after all that domain analysis. Many projects feel the need to show a mass of artifacts proportional to the work done.

Here, the work isn't in the mass of the artifacts. The work is in dividing up the world and in carefully analyzing what the structure of the artifacts should be. It's a bit like Picasso, whom legend says charged \$20,000 for a portrait he completed in ten minutes. The client was alarmed at the price, complaining, "But it only took you 10 minutes!" Picasso allegedly replied, "Ten minutes, and 40 years."

Last, we are building heavily on wisdom of the decades, if not the ages, by proactively standing on domain knowledge instead of hopping around in response to the learning of the moment. The learning of the moment is important, too, but it's not everything. The domain knowledge provides a context for such learning.

#### 5.4.1 Abstract Base Classes

If most of the domains will be object-shaped—whether justified by domain knowledge, patterns, or input from the end user mental model—most of the architecture code will be abstract base classes. Each domain will be delivered as an abstract base class, perhaps together with a small set of declarations of supporting types, con-

stants, or supporting procedures. Abstract base classes (ABCs) represent an interesting paradox. They are real source code but generate no object code on their own. They contribute nothing to the footprint of the product. Their main function is to guide the programmer as they weave Use Case code into the product.

It is instructive to think of ABCs as tools rather than as part of the product. One of the central concepts of Lean practice in Toyota today is something called *poka-yoke*. Abstract base classes may not insure that developers' code will work perfectly, but they do guide developers to write code that preserves the system form.

This might be an abstract base class for Accounts:

```
#include <string>
using namespace std;

class Account {
public:
    string accountID(void) const;
    Account(void);
private:
    string acct;
};
```

The whole purpose of ABCs is to relieve us of the burden to understand all the variants beneath it: within the graph of types beneath the ABC, one size fits all. This works because of the kind of commonality that ABCs represent: all of the subtending types have the same behavior as indicated in the ABC's interface. Of course, each one implements those behaviors differently: that's the variation in algorithm. For most architectural purposes we want to encapsulate that variation, and the subtyping graph encapsulates it well. Nonetheless, it can be good to remember key derived types that arise during analysis, and we can go one level deeper and include the base class for Savings Accounts.

```

class SavingsAccount:
    public Account,
    public TransferMoneySink<SavingsAccount> {
public:
    SavingsAccount(void);
    virtual Currency availableBalance(void);
    virtual void decreaseBalance(Currency);
    virtual void increaseBalance(Currency);
    virtual void updateLog(string, MyTime, Currency);

private:
    // Model data
    Currency availableBalance_;
};
```

This is in fact not an ABC, but a real class in the system. You will have to make a project decision whether to defer these as part of implementation or to consider them as part of architecture. One important guiding principle is: abstraction is evil. To throw away this declaration in the interest of keeping the more general, pure `Account` declaration is to abstract, to throw away information. A reasonable compromise might be to separate the two administratively. But the architecture police won't come looking for you if you put it together with the `Account` declaration as a full first-class deliverable of the architecture effort.

Note—and this is a key consideration—that the `SavingsAccount` object is *dumb*. It doesn't even know how to do deposits or withdrawals. We're sure that your professor or object-oriented consultant told you that a good account object should be smart and should support such operations! The problem with that approach is that it fails to separate what the system *is*, which changes very slowly, from what the system *does*. We want the above class declaration to be rock-solid. The changes that are driven by Use Cases need to go somewhere else.

---

Where do they go? We talk about that in CHAPTER 7 and a lot more in CHAPTER 8 but we can give a preview now. The declaration makes a curious use of the indirect template idiom [Cop1996]:

```
public TransferMoneySink<SavingsAccount> {
```

Mechanically, what this line does in C++ is to compose the what-the-system-does code in `TransferMoneySink` with the what-the-system-is code in the `SavingsAccount` class. This line makes sure that objects of `SavingsAccount` support one of the many Use Cases that a `SavingsAccount` object participates in: transferring money between accounts. In that Use Case a Savings Account plays the role of a `TransferMoneySink`. This line is the connection from the what-the-system-is world up into the what-the-system-does world. There can, in general, be many more such derivation lines in the declaration. Again, we cover this technique more fully in CHAPTER 8.

Analogous to these class declarations for domains that are object-shaped, you'll also have:

- Procedure declarations (potentially overloaded) for procedural domains;
- Template or generic declarations for generative domains;
- Constant declarations for trivial parametric domains.

You get the idea. Keep name choices crisp and enlightening; see some good tips about this in the *Clean Code* book. [Mar2009] Remember that code is read 20 times for every time it is written. For architectural code the ratio is even more extreme.

Of course, system form evolves—albeit slowly. These declarations change over time. Good domain analysis, including careful engagement of all the stakeholders, helps to slow the rate of change to domain classes. That gives you a good architecture, a good foundation, to stand on as the project goes forward.

### 5.4.2 Pre-conditions and post-conditions

Architecture provides great opportunities for supporting communication in a project. Alistair's diagram (Figure 3 on page 5) gives a low score to written forms of documentation. However, he doesn't explicitly talk about code as a powerful medium of communication. Code is like documentation when people are sitting remote from each other; however, it can be more precise than natural language documentation. It is very powerful when combined with the other powerful media in Alistair's diagram. More powerful than two people at a whiteboard is two people at a keyboard.

The ABC declarations look a bit too malnourished to serve this purpose. We can say more, and still be Lean. In the interest of having *poka-yoke* guidance for the programmer, we should look at the class as a whole to document its scope. And since programmers typically write a procedure at a time, we need to delineate member function input and output ranges as well. In general, we can view these interfaces as a contract between the supplier and the user. Indeed, Bertrand Meyer long used the term *design by contract* for interfaces like this, suitably annotated with contract information. [Mey1994] That information can take the form of invariants (remember commonality from above?); in many programming languages, we can express such invariants as *assertions*.

Assertions were first popularized in the C programming language on the UNIX Operating System, and they have handily carried over to C++ and are found in many other languages, including Java. These assertions work at run time. If we are to use them, we have to run the code. That means that we can no longer use ABCs, but that we need to fill in the methods. So we'll write code that looks like this:

```
#include <assert>
```

```
SavingsAccount::SavingsAccount(void):  
    availableBalance_(Euro(0.0)) {  
    assert(this != NULL);  
  
    // application code will go here  
  
    assert(availableBalance_ == Euro(0.0));  
    assert(true);  
}  
  
Currency SavingsAccount::availableBalance(void) {  
    assert(this != NULL);  
  
    // application code will go here  
  
    assert(true);  
    return availableBalance_;  
}  
  
void SavingsAccount::decreaseBalance(Currency c) {  
    assert(this != NULL);  
    assert(c > availableBalance_);  
  
    // application code will go here  
  
    assert(availableBalance_ > Euro(0));  
    assert(true);  
}  
  
void SavingsAccount::increaseBalance(Currency c) {  
    assert(this != NULL);  
  
    // application code will go here  
  
    assert(true);  
}
```

```

void SavingsAccount::updateLog(
    std::string logMessage,
    MyTime timeOfTransaction,
    Currency amountForTransaction) {
    assert(this != NULL);
    assert(logMessage.size() > 0);
    assert(logMessage.size() < MAX_BUFFER_SIZE);
    assert(timeOfTransaction >
        MyTime("00:00:00.00 1970/1/1"));

    // application code will go here

    assert(true);
}

```

Some of these assertions look trivial, but experience shows again and again that the “stupid mistakes” are common and that they can be the most difficult to track down. Furthermore, most “sophisticated mistakes” (if there is such a thing) usually propagate themselves through the system as stupid mistakes. So these assertions serve a secondary purpose of increasing the software quality right from the beginning. But their primary purpose is to convey information about the arguments—information that the types alone may not convey. For example, we see that there is an upper bound on the size of the log string to `SavingsAccount::updateLog`.

These C-style assertions depend on information that is available only at run-time. We can also create assertions that the compiler evaluates at compile time using information that’s available at compile time. That is more in the spirit of architecture as form. Such assertions have long been available in many programming languages through tricks with macros, templates and generics, but they are becoming more common as full first-class language features.

C++ now has a static assertion feature as part of the language.

Some languages such as Eiffel can check consistency between assertions at compile time. So, for example, you can define the contract terms for a base class method in the base class, and for the overridden method in the derived class. The Eiffel compiler ensures that the contracts obey the Liskov Substitutability Principle (LSP) [Lis1986],

which is a classic formalization of proper object-oriented programming. Eiffel ensures that no derived class method expect any stronger guarantees (as a precondition) than the corresponding base class method, and that methods promise no less (as post-conditions) than their corresponding base class methods. This makes it possible for a designer to write code against the base class contract with full confidence that its provisions will still hold, even if a derived type object is substituted where the base class object is “expected.”

#### 5.4.3 Algorithmic scaling: the opposite of static assertions

Assertions are a stopgap measure to ensure at compile time, or at worst at run time, that the system doesn’t violate architectural and engineering assumptions. Good code can make the assumptions true by construction.

ALGORITHMIC SCALING is a pattern from Fred Keeves at AT&T in the 1990s. The idea is that the entire system is parameterized with compile-time constant parameters at the “top” of the architecture, such as the amount of total configured memory, the number of subscribers that the system should support, etc. These parameters can be combined in expressions that generate other parameters for use at lower levels in the architecture, which in turn propagate them to lower levels, and so forth. Such static parameters can be used to size data structures, for example.

#### 5.4.4 Form versus accessible services

#### 5.4.5 Scaffolding

Stubbing and such. This is the stuff that we get in place to keep it out of the way of the important stuff: the domain logic. This includes key patterns such as separating initialization from instantiation, treating instantiation as a scaffolding problem and initialization [Cop1992, 79—82] as something related to the business.

### 5.4.6 Testing the architecture

Though this is not a book about testing, most testing books look at the imperative and procedural aspects of testing. It's important to bring out two more facets of testing more pertinent to this book: usability testing, and architecture testing.

After you reach your first architecture deliverable, you can do both usability testing and architecture testing. In the spirit of Agile you can of course do partial testing along the way; don't leave it all until the end. And in the spirit of Scrum, you want your architecture work to be done-done-done before you start building on its foundations.

#### *Usability testing*

Usability testing focuses on the interaction between the end user and the system, ensuring that typical end users who fit within the user profiles can accomplish what they need to accomplish. Good usability testing requires the skills of specially trained and experienced testers. Such testing is usually done by walking end users through the interactions and screens that reflect the behaviors of the envisioned system.

What usability testing does for the architecture is to validate whether we've captured the end user mental model.

You can go one step further by combining usability testing with dynamic architecture testing, using CRC cards. Bring together your team and have each team member play the role of one or more objects in the system. Each team member holds an index card representing either a system domain object, or a system role (we will talk about roles in Section 6.6). The facilitator asks the end user to run through a Use Case scenario. As the user tells of performing some gesture (pushing a button on the screen or pulling down the menu) the appropriate role or object rises to the occasion by providing the necessary service. These responses are the responsibilities of the object or role's public interface; the team member writes these down on the left column of the card as they arise. If a role or object requires

---

the cooperation of another role or object to further the transaction, then the corresponding team member is asked to carry on the “execution” and to write the appropriate responsibility on his or her card. When the first team member hands off control to the second, he or she writes down the role or object of the second in the right-hand column of the card. It is useful to use a talking stick or cat toy or other “sacred artifact” as the program counter in this exercise.

When you’re done, compare what’s on the cards with what’s in the architecture

### ***Architecture testing***

The goal of architecture testing is to establish that What can “done” mean at this point?

Gedanken experiments with crazy sales people.

---

## ***5.5 Relationships in architecture***

Object-oriented architecture doesn’t just mean “a bag of objects.” Just as in housing architecture, architecture is more than a parts inventory. It is about *relationships* between parts.

Much of this is dynamic and will come in CHAPTER 7 and CHAPTER 8.

#include nesting  
static assertions  
patterns

---

## ***5.6 Not your grandfather’s OO***

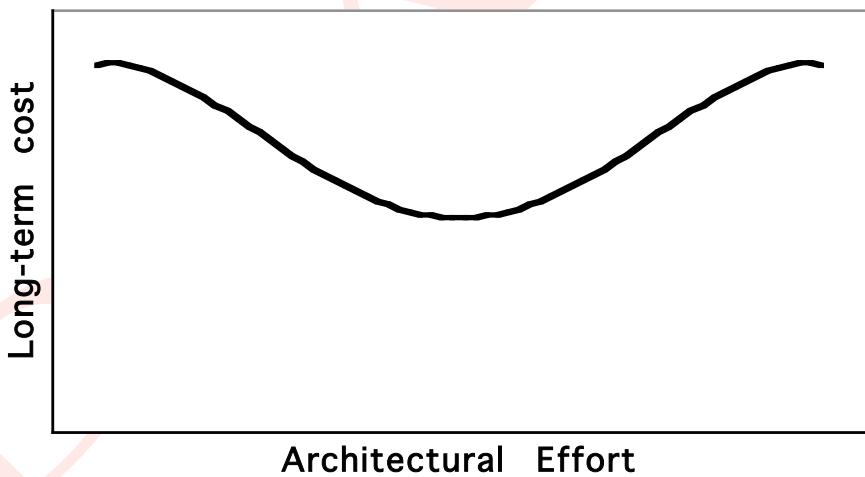
Here we talk about the three main kinds of “things:” roles, context objects, and data objects. Some things that look like domain objects will end being contexts. In some cases we won’t have substantial Use

Cases and won't even have Context objects; that's a different architectural style.

Don't go into too much detail, but introduce the vocabulary and concepts. Maybe use the example of banking accounts where, to one set of users, what the system is comprises Accounts, but to another set of users it's transactions. So you have a hybrid layer that will end being context objects that bridge the two worlds. There can be arbitrarily many of these layers but humans have difficulty going beyond about five (the number of things that will fit in short-term memory). These things start to tangle with code above what-the-system-is and get into the MVC framework (I think...)

---

### 5.7 How much architecture?



---

Figure 14: How much architecture?

## 5.8 Documentation?

We want the documentation that packs the punch. “That’s the page I read that made the difference.”—Kevlin Henney, QCon. Much of the documentation of this nature is that which describes relationships that aren’t immediately visible in the code. For example, we might have designed a given encryption algorithm to work particularly well with a separate compression algorithm. We wouldn’t be able to see that in the code itself. Code commentary of course can help, but we should also capture such design decisions in the document that overviews the architectural partitioning.

In large building construction the real blueprints, called *as-built*s, aren’t finalized until construction is complete. They show where every heating duct and ever water pipe lies within the walls, so the maintenance people can get at them should the need arise. To assume exact locations for those artifacts before actually installing them would be guessing.

### 5.8.1 The Domain Dictionary

### 5.8.2 Architecture Carryover

Successive generations of products in a given product area share much of the same architecture. Basic domain concepts change slowly; it’s the features and technology that change more rapidly. This suggests that we might be able to re-use architecture. What might that mean?

The traditional view is that one can re-use platforms. However, success with such reuse has been slim at best, except in the case of generic software such as low-level libraries and operating systems, which have evolved to be domain-neutral. The pattern discipline evolved out of the belief that the best carry-over from the past is ideas and basic form, rather than code.

But what if the code *expressed* basic form?

[Architectural efforts should both capture patterns across the entire domain and should use them as input to the domain analysis.]

---

### 5.9 *History and Such*

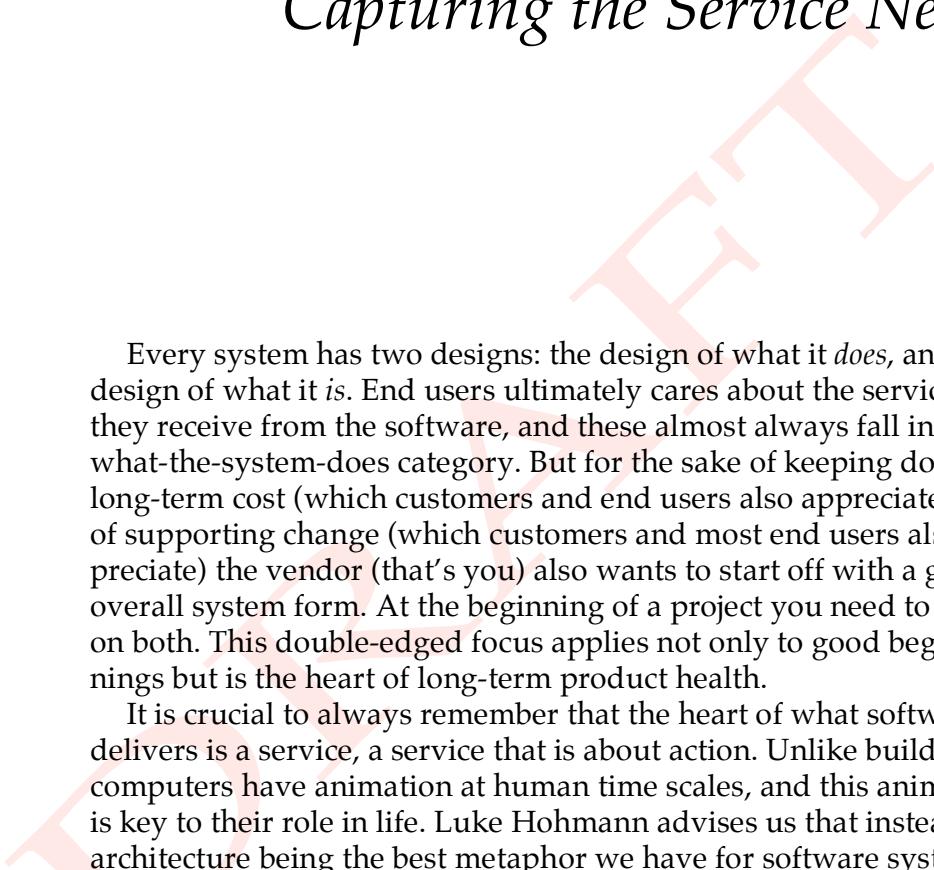
Neighbors: domain analysis

Multi-paradigm design: domain analysis in an OO world.

Weiss and Lai: DSLs: not a great success.

Eric Evans Domain-Driven Design.

# *What the System Does: Capturing the Service Needs*



Every system has two designs: the design of what it *does*, and the design of what it *is*. End users ultimately cares about the services they receive from the software, and these almost always fall into the what-the-system-does category. But for the sake of keeping down long-term cost (which customers and end users also appreciate) and of supporting change (which customers and most end users also appreciate) the vendor (that's you) also wants to start off with a good overall system form. At the beginning of a project you need to focus on both. This double-edged focus applies not only to good beginnings but is the heart of long-term product health.

It is crucial to always remember that the heart of what software delivers is a service, a service that is about action. Unlike buildings, computers have animation at human time scales, and this animation is key to their role in life. Luke Hohmann advises us that instead of architecture being the best metaphor we have for software system design, that we should look instead to another one of the arts: dance. Like architecture, dance is also about form, but it is alive and dynamic. CHAPTER 5 gave us our dance hall; now let's proceed to the dance.

Given that this is an Agile book, you might have expected the more user-facing what-the-system-does section to precede the architecture section. We would do that in an ideal world: we would drive design according to the pure desires of the end user and let nothing

else distract us. We investigate architecture a bit before discussing functionality in detail because it is more efficient that way. If the architecture terminology supports mutual understanding between the provider and user, we can move on to functionality with confidence.

---

## 6.1 *Users and Customers*

Indeed, having an outward focus is a key Agile tenet. That's where most change comes from. We embrace that change. We love it, because it is this change that brings the opportunity to create and deliver support for new services, and that's our business. We give customers product; they give us money or some other consideration of like value in exchange for the money. They in turn support a user community, usually under similar considerations of exchange. But for us in the software craft, it's often useful to view the customer as a middleman who adds value through sales or marketing efforts. While they add value, they also bring liabilities. If requirements have to work their way from the users to us back up the value stream, the customer may in fact add noise. We want a direct path to end-users.

The first thing to capture is the way that your end user does business. Software should help them do their business with higher throughput or quality, or lower cost or latency, than is currently possible. Sometimes, software makes whole new services possible. Trained observers can spend time living with your end users, observing their day-to-day business and making models of what they do. This is not a matter of interviewing your end users, or of bringing them on-site as oracles to "fill in" story-based models that someone creates outside the workplace. That makes no more sense than creating behavioral models of a newly discovered species of ape by observing its behavior while captive. Outside their native habitat, end users let their imaginations run wild with possibility. Possibility will have its day, but not yet. For now, the analyst is student and the end user is teacher. A wide range of talents can be brought to bear here: user experience people who capture business models; psy-

chologists who capture end-user cognitive models of their use space; screen designers who build prototype user interfaces that evolve into the interface that eventually becomes the program. But it is always important to keep the actual coder in the loop, integral to these dialogues that specialists might help facilitate.

The second thing to analyze is the software that's already there. If you're building software from scratch, you'll have the good fortune to avoid this step. But there is hardly any software development that starts in a vacuum. Most coding over a product's lifetime takes place in the context of code that came before. Even the newest code sometimes needs to work and play well with other software systems, and *that* code bears analyzing as well.

This is an exploration activity but, in the spirit of Scrum, we always want to focus on delivering something. The artifacts that we deliver include:

1. Screen Designs
2. Input to the Domain Model
3. Behavioral service descriptions: Use Cases or a custom formalism

Why screen designs so early in the process? Too often, programmers design the screens around the facility of some GUI-builder or their own metaphors of interaction. Agile regrettably says nothing about our product being *useful* or *usable*; only that it works. Here, we raise the bar.

What kind of input to the domain model? Knowing a bit about the range of functionality that users expect can help scope the domain.

Why Use Cases? Because you want to make key decisions early. User stories are a way of avoiding decisions and fall into the trap of a small gain in reducing design rework at a large later cost of production rework. Use cases are a way of bringing more of the important issues to earlier design phases. Knowing Use Cases early can help designers better pre-order design decisions and can establish dependencies between requirements, and between work tasks.

Sometimes, Use Cases are overkill. Use Cases exist to capture sequencing requirements that occur while working towards a goal in a specific context. Let's say you were building a graphical shapes editor and wanted to capture the requirement that there be a delete command, and you wanted to capture the behavioral components of those requirements. Can you define the sequence of operations in some context, towards some goal? You can do that at the level of keystroke, but a keystroke-level description is just an implementation of a Use Case scenario: it is not a Use Case. What's the alternative? The answer is a somewhat unsatisfying: it depends. We'll revisit that question at the end of the chapter. In the mean time, we'll focus on systems and requirements that are well suited to Use Cases, recognizing that many systems will employ a mixture of techniques for defining user services. Some of the practices described in the "Use Case thread" below are hardly unique to the Use Case formalism. For example, User Stories can always be a good way to get started informally. We encourage you to think and to view this chapter as a department store; put those things in your shopping cart that fit together to meet your needs. Keep your shopping cart small.

---

## 6.2 *Getting Started: User Stories*

User stories are an informally written way of combining the user stake in some feature with the user motivation [CITE needed: from dannorth.net]. Here are some examples of user stories:

- As a user I want to be able to set the alarm on my cell phone so I can get up in the morning.
- As a snoozer I want to be able to activate 'snooze' when the alarm goes off, so I can sleep 10 minutes more.
- As a user I want to set the alarm so I can get up at the same time every morning.

### 6.2.1 The importance of the End User Motivation: Goal-driven Use Cases

---

## 6.3 User Stories to Use Cases

### 6.3.1 The Prelude

The prelude starts with the Use Case name

#### Use Alarm

- We want to support a Homebody in using his / her cell phone alarm to be awakened in the morning

#### Precondition

- The cell phone is on.
- The cell phone clock is right.

#### Post-condition

- The cell phone alarm is ready to receive new alarm settings

### 6.3.2 The Basic Flow

Also called the “main success scenario.”

Step	Actor	System	Comment
1.	The Homebody chooses to use the alarm	The System provides setting options	Hour: Minutes? Define "setting options."
2.	The Homebody sets the alarm	The system alerts the Homebody according to the alarm settings	
3.	The Homebody stops the alarm	The system forgets the alarm settings	Not true if "Recurrent" is on

After the basic flow you should also describe alternative flows. We discuss those below in Section 6.4.

### 6.3.3 The Postlude

A few remaining sections

#### Non-functional Requirements

- The alarm should go off though battery is low

#### Open Issues

- Is 'snooze' or 'stop' the most used – and what will we support as the normal flow?
- Will 'recurrent' be the normal for a Homebody?
- Would we like to support a graphical interface showing an analog clock for the alarm settings?

---

## 6.4 Capturing Alternative Flows

In other formalisms, called extensions and deviations.

Ref. Step	Alternative Flow	Comment
1a.	Set recurrent alarm	Should recurrent be part of the basic flow?
3a.	Snooze	Should snooze be a part of the basic flow?
2a.	Reset alarm	
2b.	Cancel alarm	Please make it easy—but not too easy—for the Homebody to do this!
1b.	Set new alarm tone	Is it out of scope of this Use Case?

#### 6.4.1 Packaging “Non-Functional” Requirements in Use Cases

Then in Scrum you can have a consistent, homogenous Product Backlog

---

#### 6.5 Managing Use Cases Incrementally

No big-bang up-front requirements!

Use Cases scenarios become elaborated as the scenarios come to the top of the Product Backlog in Scrum.

---

#### 6.6 Use Cases to Roles

Blah

### 6.6.1 Framing it Out

### 6.6.2 Language Support: Interfaces

### 6.6.3 Later: Use Case scenarios to algorithms

---

## 6.7 Use Cases and Architecture

### 6.7.1 Delimit project scope

### 6.7.2 How about interactions with the architecture?

Not yet, because the translation to algorithm is tricky. On the other hand we'd like to keep everything in terms of form... That's a nice idea but it will lead to rework.

---

## 6.8 "It Depends": When Use Cases are a bad fit

Use Cases capture a sequence of events—a scenario—that takes the end user toward some goal in some context. What if you have difficulty expressing a user interaction in that form? Sometimes, Use Cases are a bad fit. If you look at a scenario and you can't answer the contextual questions, "Why is the user doing this?" and "What is his or her motivation in context?" then Use Cases are probably a bad fit for your domain. You should seriously explore alternative formalisms to capture requirements.

What alternatives, you ask? *It depends* is the answer. Some designs in fact are dominated by such operations. Think of a graphical shapes editor that features "Use Cases" such as rotating, re-coloring, re-sizing, creating and deleting objects. From a Use Case perspective,

these are atomic operations. Though each of these operations may require multiple end-user gestures, each is still a single step in a Use Case. The number of gestures or mouse clicks required to achieve a goal is a user interface design issue, not a requirement issue. Moving an object, or re-coloring it, is not an algorithm in the mathematical or even vernacular sense of the word; it is atomic. If there is no business need to group such operations together, then each is a command in its own right and we really don't need Use Cases. The user is not working with *a sequence of tasks to reach some goal in a context*. Each command reduces to a primitive operation on the domain object itself, and MVC alone with its direct manipulation metaphor is enough to get the job done. If you use Use Cases to capture the input to a system dominated by atomic operations, you'll end up with hundreds of Use Cases that belabor the obvious.

### 6.8.1 Classic OO: Atomic Event Architectures

One way to explain the success of object-oriented programming is that it rode the coattails of graphical user interfaces and mice. These human interaction styles captured our imagination and led to computing as a companion thinker rather than as a distant subcontractor. The human satisfaction with immediate feedback, the speed (or at least the illusion of speed) with which one could drive to a solution, and the human-like behaviors of these interfaces at Xerox PARC paved a path for Smalltalk's influence on the world.

The link between this user engagement and Smalltalk in particular, or object orientation in general, is more than incidental. There is a continuum of representation from the end user's mental model of the world, to the graphical interface, to the structure of the objects beneath the interface. The idea is that the user manipulates interface elements in a way that fools his or her mind into thinking that they are manipulating the objects in the program—or, better, the real world objects that those programmatic objects represent. To place an item in one's shopping cart in an online bookstore gives the end user, at least metaphorically, the feeling of actually shopping with a shopping cart. This is sometimes called the direct-manipulation

metaphor [Lau1993]. We'll later talk about the Model-View-Controller-User architecture (Section 7.1) as the most common way to create the illusions that support this metaphor.

To get something done on a direct manipulation GUI, you first choose an object (like a book) and then "manipulate" it (put it in your shopping cart). It feels awkward to instead first say, "I'm going to buy a book" and then select the book. This is because we are *thinking* about the book before the actual *doing* of deciding to purchase. We probably wouldn't have come to the website if we had to dwell on the question of whether we wanted to buy a book. This is called *a noun-verb interaction style*: we choose the noun (the book) first, and the action (to buy) second.

Notice the close correspondence to our architecture, which has a what-the-system-is part (the books) and what-the-system-does parts (the Use Cases such as: buying a book, finding earlier editions of the book, and retrieving reviews of the book).

The noun-verb metaphor leads to interfaces that take us through a sequence of screens or contexts. When we select the book, we are in the context of thinking about that book. The book has a physical representation on the screen (as a picture of the book, as its ISBN, or its title and author) that fills our conscience mind. The conscious processes of our mind focus on that object. There are many things we can do with that object, and our thinking process leads us to where we want to be. Only then will we "execute a command." The "command" is something like "put this book in my shopping cart" and on most graphical interfaces would be enacted by pushing a button thus labeled.

This means that the object on the screen (the book) is like the object in the program, and from the screen we can see how to manipulate that object in the ways that make business sense (put it in the cart, retrieve reviews, etc.) Those are, in effect, the member functions on that object. The customer works, and invokes commands, in a very focused context: that of the object. The object becomes the command interpreter—one primitive operation at a time.

This style of interaction in turn leads to an architectural style, which in this book we call *atomic event architectures*. In such a style

---

we focus on the objects and what we can do with them, rather than on the “scenarios” that exist in retrospect as the catenation of these events and commands. We can take the context for granted. More precisely, we focus on the roles that the objects can play in the current context (though a book can play the role of a doorstop, we focus on a different role when purchasing a book online). We give the user choices, allowing the user to interact with the program in an Agile way: one choice at a time. If we focused on the scenario instead, we’d have a master plan that would preclude choosing different books in a different order, and which would remove the user’s option to change course at any time (to just forget the current book and go onto another, or to take a book back out of the shopping cart).

### 6.8.2 Business Rules

Business rules capture yet another important set of behavioral requirements outside Use Cases. If you put business rule decisions in Use Cases, the Use Cases can become unreadable. Many domains have found their own way to capture and communicate business rules, most based in natural language, but some based in formal languages (“formal” in the sense that a program can analyze *something* about them).

---

## 6.9 Another Example: Interface to Bank Accounts

**Use Case Name:** Transfer Money

**User Intent:** To transfer money between his or her own accounts

**Motivation:** The Account Holder has an upcoming payment that must be made from an account that has insufficient funds

**Preconditions:** The Account Holder has identified himself or herself to the system

**Basic Scenario:**

1. Account Holder requests an account transfer
2. System displays valid accounts
3. Account Holder selects accounts
4. System displays selected account balances
5. Account Holder chooses the amount to transfer
6. System moves money and does accounting

**Variations:**

- 1a. Account Holder has only one account: tell the Account Holder that this cannot be done
- 2a. The accounts do not exist or are invalid...

**Postconditions:**

- ✓ No money is “lost” in the transaction
- ✓ The periodic statements reflect the exact intent of the transaction (a transfer is a transfer — not a pair of a withdrawal and a deposit)

**Figure 15: Transfer Money Use Case**

**Figure 15** shows a simple Use Case for transferring money between accounts. It captures the essential business process between the end user and the system under construction. We avoid implementation decisions; the people gathering requirements in the Use Case have much less insight on how to do a good job of coding this

---

up than the developers will, so the requirements engineers defer to their judgment.

Notice that the Account Holder selects both accounts in Step 3 of the Use Case: why is this not two steps? Our rule of thumb is that we break out an activity into multiple steps only if the system does some work between them that makes the second step easier for the user. Here, serializing the Account Holder's selection of the accounts doesn't imply that the system should do anything between them that makes the second selection easier.

**Use Case Name:** Move Money and Do Accounting

**User Intent:** To transfer money between his or her own accounts

**Motivation:** The Account Holder wants to move money between funds to avoid restrictions on certain kinds of transactions on the source account, or to take advantage of the convenience or financial gain resulting from the money resting in the destination account.

**Preconditions:** The Account Holder has identified the Source Account, Destination Account, and the amount to be transferred

**Basic Scenario:**

1. System verifies funds available
2. System updates the accounts
3. System updates statement information

**Variations:**

- 1a. Funds not available in the Source Account: system displays an error message

**Postconditions:**

- ✓ No money is “lost” in the transaction
- ✓ The periodic statements reflect the exact intent of the transaction (a transfer is a transfer — not a pair of a withdrawal and a deposit)

**Figure 16: Move Money and Do Accounting Use Case**

In **Figure 16**, we show the Move Money and Do Accounting Use Case. A properly formed Use Case scenario captures only essential business relationships. Ideally, the only sequencing in a scenario is that which is essential to business semantics. The Use Case intentionally omits sequencing detail that go beyond the level of end-user awareness or concern, such as the transaction semantics between the balance reduction on one account and the balance increase on the

---

other. These are not requirements issues, at least not at the level of scenario definition. There may be a requirement that the bank not steal any money from the customer, but the programmer should be given the freedom to implement that in any way that balances remaining requirements.

---

### 6.10 Documentation?

As should be obvious by now, the Use Cases themselves are important Agile documents. Remember that documents serve two purposes: communication in the now, and corporate memory in the future. Use Cases as a form help structure thinking in the now. They provide a place to track dependencies between work items for yet-to-be-implemented items on the Product Backlog and for work in progress on the Sprint Backlog in Scrum.

Whether Use Cases should be maintained as historic documents is a matter for individual projects to decide. You want to keep Use Cases if they codify important learnings to which the team may want to return in the future. Sometimes you want to remember why you did something a certain way when the reasoning may not be obvious from the code itself. A good example is code that deals with a side effect from an interaction between two Use Cases. So if you have a need for such history, and if you have a way easily to recover such documents on demand, you might consider keeping them.

Most projects find it worthwhile to maintain Use Cases in the long term because they keep growing over time: they are never done. It's always good to return to the original basic flow as a starting point for reasoning about a new alternative flow. To not keep the Use Case around will lead to rework in development, to re-inventing the reasoning and mechanisms that were understood when design decisions were made in the past. Therefore, throwing away the Use Case wouldn't be lean. Keep it around.

The alternative to keeping Use Cases in the long term is the idea of "disposable analysis," where the Use Case serves the communica-

tion function without the overhead associated with its role as an archival artifact. If that's the case, throw the Use Case away after it's coded up. To keep it around isn't Lean.

---

### *6.11 History and Such*

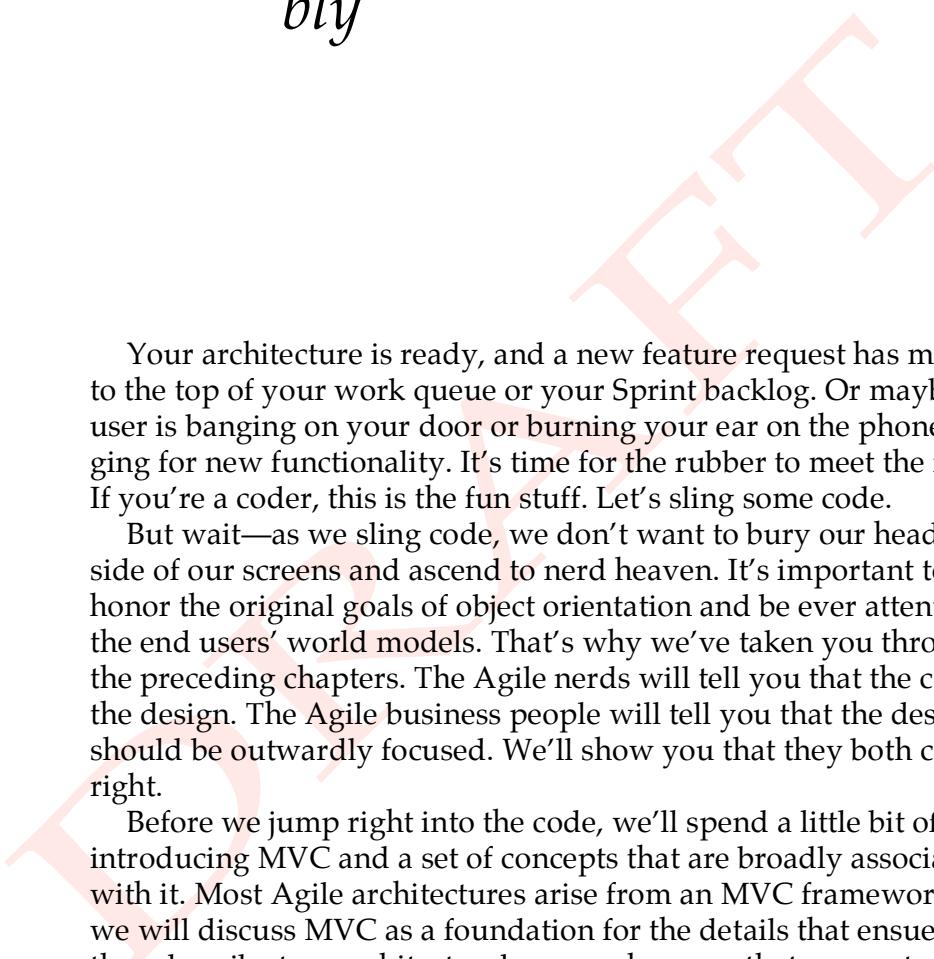
Ivar Jacobsson pioneered Use Cases in ???? and tried to offer them to the market through his CASE tool, Objectory.

Use Cases became an official part of UML in ????

Alistair Cockburn re-interpreted Use Cases and started to bring his vision into the software world starting in about ???? Cockburn's vision is completely in harmony with the Agile vision he helped shepherd as one of the organizers behind and signers of the Agile Manifesto.

The XP community (whose origins date from around 1997) viewed the UML-based history of Use Cases in the 1980s as something to be avoided, and supplanted them with User Stories. User Stories worked together with other XP principles and practices such as on-site customer and test-driven development to support rapid development of small systems. However, user stories as self-contained minimalist units had problems scaling to systems thinking. In complex systems, the relationships between stories were as important to the stories themselves. Methodologists gradually started adding features to user stories to accommodate some of these concerns: for example, Mike Cohn recommends adding information for the testers. At this writing these individual variations have caused the definition of "user story" to diverge broadly and there is no single accepted definition of the term.

# *Coding It Up: Basic Assembly*



Your architecture is ready, and a new feature request has made it to the top of your work queue or your Sprint backlog. Or maybe a user is banging on your door or burning your ear on the phone begging for new functionality. It's time for the rubber to meet the road. If you're a coder, this is the fun stuff. Let's sling some code.

But wait—as we sling code, we don't want to bury our heads inside of our screens and ascend to nerd heaven. It's important to honor the original goals of object orientation and be ever attentive to the end users' world models. That's why we've taken you through the preceding chapters. The Agile nerds will tell you that the code is the design. The Agile business people will tell you that the design should be outwardly focused. We'll show you that they both can be right.

Before we jump right into the code, we'll spend a little bit of time introducing MVC and a set of concepts that are broadly associated with it. Most Agile architectures arise from an MVC framework, so we will discuss MVC as a foundation for the details that ensue. We'll then describe two architectural approaches: one that supports short, snappy event-driven computation, and the other that supports goal-oriented task sequences that the user wants to complete. In this chapter, we will discuss classical object-oriented architecture, where we distribute Use Case responsibilities directly into domain classes. We'll leave the incorporation of full Use Case requirements to

CHAPTER 8. We end up the chapter with guidelines for updating domain classes to support the new functionality.

---

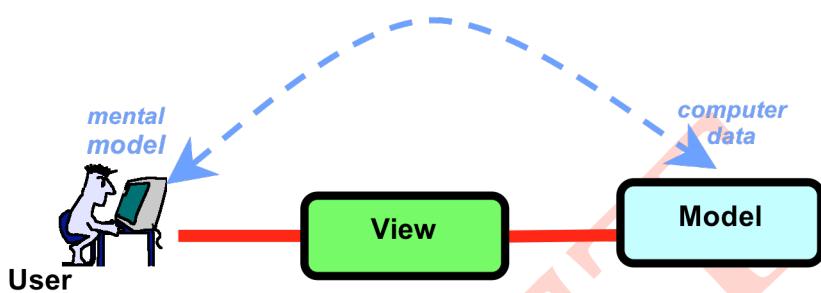
## 7.1 *The Big Picture: Model-View-Controller-User*

MVC is a time-honored (since 1978) architecture framework that separates information representation from user interaction. Calling it MVC-U emphasizes the most important object in the framework: the user. It is the user's system interaction model that gives rise to the rest of the MVC structure. We'll use the terms interchangeably in this book, adding the *U* particularly when we want to emphasize the user engagement.

We are right at home with MVC-U in separating what the system *is* from what the system *does*. It's not a coincidence. The early vision of object-orientation was to capture the end user's mental model in the code. This dichotomy, of thinking and doing, is a widely embraced model of the end user's mental organization, and has found some kind of home in almost every paradigm in Western history. Computing has not escaped that dichotomy, and objects are no exception to the rule.

### 7.1.1 **What is a program?**

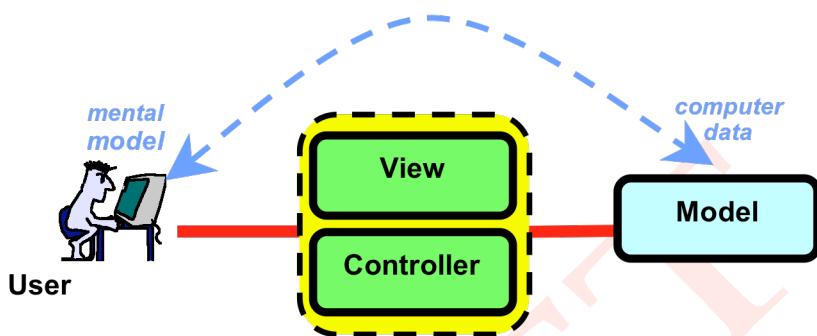
Let's talk video games for a second. If you're playing an interplanetary war video game, you feel that you really *are* in that fighter ship, piloting it—if it's a good game. The computer or gaming system disappears and you are absorbed in total engagement. Brenda Laurel describes this experience of engagement in her insightful book, *Computers as Theatre* [Lau1993].



**Figure 17: The View bridges the Gap between Brain and Model**  
*From [Citation Needed]*

The goal of a program is to create a reality that you can experience as such. For most of you reading this book, these realities are often modeled after your business world. When I use a program to logically retrieve an item from a warehouse, I am not literally touching the crates. The program may enact robots or may print out an instruction for a warehouse worker to fetch an item, but the end result is as if I had done it myself. Therefore, the program creates an illusion: *I didn't really move the item.* At some level, all of programming is about creating illusions. Good illusions are contrived to fit the models that lie latent in the end user's brain.

MVC-U helps the programmer create such illusions. The first step is to bridge the gap between the end user's mental model in his or her brain, and the computer data in the program (Figure 17). That works well for simple models, as one might find in a graphical shapes editor. More often than not, things are more complex than that. My old friend Dennis DeBruler used to say that complexity is proportional to the number of distinct, meaningful ways you can look at a thing. MVC-U allows multiple views of the same data—for example, a pie chart and a bar graph view of simple data, or a floor plan view and front elevation view of a house in a CAD program.



**Figure 18: The Model-View-Controller Paradigm**

*From [Citation Needed]*

---

We need a tool to coordinate the views, if multiple views are in play concurrently. The coordination itself is done by a part of the tool called the controller (Figure 18).

### 7.1.2 What is an Agile program?

Interactive programs live close to human beings. We can in fact think of them as an extension of human reasoning and mental processing, much as Alan Kay envisioned the DynaBook as a lifelong companion computer that extends human brainpower. Humans live and react in human time scale and their mental activities face an ever-changing world. If the software doesn't keep up with that change, it leaves the human mentally behind their world's pace. As Agile is about embracing change, it is particularly interesting to consider how an Agile mindset can help the world of interactive computing.

Contrast the interactive world with a batch-processing world. Human users don't interact with batch programs, at least not while the programs are running. Users may type the data that is fed to the program and they may read the printout from the program, but only

long (by machine time scales) after the program execution has been extinguished. Such programs tend to structure themselves around relatively stable business processes. In fact, over time these programs tend to become constrained by other programs and processes in the batch stream. This isn't always true: for example, all the popular search engine sites depend on long-running batches to keep the search data fresh, but those kinds of systems tend to be the exception rather than the rule.

We can also contrast interactive code with embedded code. Embedded code is often fast running and often must meet real-time deadlines, such as one might expect from the firmware controlling the fuel injection in your car. But that software is *embedded* in the hardware it controls and is often quite distant from any direct user interface. It is usually constrained by the industrial processes for the hardware in which it is embedded, so its rate of change is usually low.

Most of the time, Agile code has an interactive interface, and that interface usually gives the user a feeling of engagement with a real world where change is the name of the game. That includes the programming styles discussed both here and in CHAPTER 8. Such an interface is often a graphical interface but it can use any technology that helps the end user engage with the software. What makes it Agile? This level of engagement pushes change into the user's face, in real time: Agile embraces change. In the same sense that teams are asked to engage their customers in Agile development, so an end user engages Agile *software* after the shrink-wrap comes off. It's about individuals and interactions—not only with other people, but also with the software, and one doesn't find as much of this kind of interaction in batch or embedded systems.

If one looks at the foundation of object-orientation, it comes back to capturing the end user's mental model of the world in the code. If that world is a dynamic world, then we're likely talking Agile software—embracing change along with collaboration between the end-user (beyond-the-customer) and the code. How do we manage interaction between the end user and the software? The Model-View-Controller-User framework has become a de facto standard for such

systems. *There exists a fundamental, deep and incestuous relationship between Agile, object orientation, and Model-View-Controller-User.*

Before we elaborate on MVC in the next section, it's probably important to answer the question: "How do I apply Agile development techniques to my embedded platform code?" Similar questions arise for batch code. While there is no universal answer, there is a strong tendency for the techniques of Agile software development to serve interactive applications, and to be less useful or relevant in batch and embedded applications. But the exceptions are too numerous to ignore. The rule of thumb is to use common sense: if the shoe fits, wear it. If you know your requirements, and know that they won't change, then maybe Agile is overkill for you and waterfall is just fine. There, we said it. It's just that it's very rare that we find real projects with the luxury of being in that position. If you are, be conscious of the gift that fate has given you and take advantage of it!

So much for Agile: how about Lean? While Lean development's benefits really shine in an environment where user needs evolve or vary widely, they also add value even to stable systems that may be more distant from direct user interaction. It is always a good idea to reduce waste and inventory, and to continuously and passionately strive towards process improvement. Many of this book's techniques therefore apply beyond so-called Agile projects. So, which of the book's practices are Lean and which are Agile? We'll leave that to your reasoning; the answer may not even be the same all the time. We hope that the book provides enough background and context that you can use your common sense to judge the answer.

O.k., now let's move into the nerdy stuff.

### 7.1.3 MVC in more detail

Model-View-Controller (without the *U*) has found expression in countless articles and books as a framework with a familiar set of APIs. We won't repeat them here. Unfortunately, while many of these descriptions satisfy MVC-U's goal to separate the interface logic from the business logic, they miss the other goals of direct manipulation, of supporting multiple business domains, of supporting

Model	View	Controller
<ul style="list-style-type: none"> <li>Updates its data at the request of commands from the Controller</li> <li>Notifies appropriate views when data change</li> <li>Can be asked to register/de-register Views</li> </ul>	<ul style="list-style-type: none"> <li>Presents a particular representation of the data to the user</li> <li>Can ask Model for the current value of data it was created to present</li> <li>Manages cursor position; together with Controller, handles selection</li> </ul>	<ul style="list-style-type: none"> <li>Creates and manages Views</li> <li>Together with View, handles selection</li> <li>Passes locations and keystrokes to the Model</li> <li>May pass keystrokes to the Views (e.g., for character echoing)</li> </ul>

**Figure 19: MVC APIs**  
*From [Citation Needed]*

Conway's Law, of supporting the DynaBook metaphor [Kay1972] or the Tools and Materials metaphor [Rie1995]. For a deeper understanding of MVC, you can read a bit about its history from the perspective of its creator [Ree2003]. For those who want a reminder of the APIs, see Figure 19.

#### 7.1.4 MVC-U: Not the end of the story.

While MVC does a good job of helping end users (and programmers, too, of course) interpret the data in the program, and start to make sense out of how an interesting algorithm might interact with those data, it doesn't capture anything about the algorithms themselves, of what they do, or of how they work. That may or may not be important. In "classical object-oriented programming," the algorithms are trivial relative to the structure of the data, and we turn our attention to the data. But sometimes it matters.

## A Short History of Computer Science

We can characterize many object-oriented systems as having relatively simple external behaviors that operate on richer and more interesting data structures inside the software. These simple or “atomic” exchanges between people and software have dominated object orientation since its earliest support of graphical user interfaces. To appreciate the broader landscape of software architecture a bit more, we whimsically look back in history.

Computers were a wonderful invention. They allowed people to describe a repetitive task once, such as tabulating the tax on each of 10,000 payroll records, so that untiring machine could apply that task to each of those records. The job of the computer was to compute; data were the subject of those computations. Early programs were collections of these mundane algorithms linked together into increasingly complex computations.

The focus was on the structure of the computations, because that's where the business intelligence lay. All the algorithms for a given “job”, which was usually a batch of computations thrown together, were loaded into memory. Then, one by one, data records were fed to the program through the card reader or consumed from a disk file. The data throughput was massive: a card reader could read 1000 cards a minute, and each card bore 960 bits. Each data record had a short lifetime. However, the organization of the data was uninteresting except for its layout on the punch card or paper tape: one had to know which data appeared in which columns, but little more than that. The representation of data inside the computer program followed the organization of the procedures. The data were passed from procedure to procedure as arguments. Copies of (or references to) the data lived on the activation records of the procedures as local variables. Users never interacted *with* the program; they interacted with the 026 cardpunch machines to prepare the data *for* the program, and they interacted with the 132-column printout produced *by* the program.

Fast forward about thirty years through many interesting intermediate developments such as block worlds and SHRDLU and da-

tabases, and we come to the era of three interesting machines at Xerox PARC: the Dove, the Dorado, and the Dolphin. Though graphical screens had been around for a while, these machines took the interactive interface to new limits. There was no card reader; you interacted with the machine through the GUI and a strange device created by Doug Englebart called the *mouse*. Each mouse click caused the machine to do something—instantaneously. Each function (or method) was small, designed to respond in human time scales to human interactions. There was no massive data input: the mouse and keyboard were typically 8-bit devices that could operate roughly at human typing speed, about 1000 times slower than a card reader. Now, the data lived *in* the program instead of living on the cards. The world had been turned inside out. These new machines would be the birthplace of a new style of programming called object-oriented programming.

We present this little history to illustrate a key principles that has dominated object-oriented programming for years: The methods are small in relation to the data, and the overall method structuring is relatively unimportant with respect to the overall data structuring. This balance between a dominating data structure and a repressed method structure characterizes many interactive programs. Consider the graphical shapes editor again. We don't think of the operations on shapes as being algorithms. Such operations are usually encapsulated in a single shape and are tightly tied to the shape-ness of the shape: re-color, move, and resize. The same is true for the primitive operations of a text editor: inserting and deleting characters.

### **Atomic Event Architectures**

Consider a shape-editing program (again). You might need a degree in geometry to understand the polytetrahedra and the data structures used to represent them, but the operations on them are simple: move this shape to here, recolor that shape to be this color. Even the most advanced operations, such as graphical rotation, are still atomic algorithms without much rich structure.

Concern	Atomic Event Architecture	DCI Architecture
User goal	Direct manipulation of a domain object to ask it to do something	A sequence of tasks toward a goal in some context
Requirements Formalism	According to need: state machine, custom formalism: it depends	Use Cases
Technology	Good old-fashioned object-oriented programming	Multi-paradigm design (procedures and objects); DCI
Design focus	Form of the data	Form of the algorithm
Scope	Single primary object or a small number of statically connected objects	Multiple objects that dynamically associate according to the request
Interaction Style	Noun-verb: select an object first to define a context, and then select the operation	Sometimes verb-noun: select the Use Case scenario first, then the objects
Example	Text editor: delete character Bank system: print account balance	Text editor: spell-check Bank system: money transfer

**Figure 20: Two Agile architecture styles**

In good object-oriented programs of the 1980s, methods were very small—in Smalltalk programs, they were commonly two or three lines long. We were encouraged to think of a method on an object the same way that we think of an arithmetic operator like `+` on an integer: simple, atomic, and precise. In the parts of your program that have that nature the algorithms aren’t terribly important (at least to the system form, to the architecture) and we shouldn’t fret about how they mirror the user mental model. We discuss that style of architecture, a so-called atomic event architecture, here in this chapter.

## DCI architectures

On the other hand, algorithms are, or should be, first-class citizens in most programs. Consider a text editor, which you may right now think of as a simple program with primitive operations such as inserting or deleting a character. But how about spell checking? Or global search-and-replace? These are *algorithms*. We could make them into methods, but the question is: methods of *what*? I can spell-check a file, or a buffer, or perhaps just the current selected text. This isn't just a matter of adding a spell-check method to some object: Spell checking is an important concept in its own right. For such cases—and they are many—we need the other half of MVC: the Data, Context and Interaction (DCI) architecture. We talk about that in CHAPTER 8.

## 7.2 The form and architecture of atomic event systems

Atomic event systems have very little concern with the user model of how he or she organizes sequences of work. In CHAPTER 6, we discussed Use Case as a way to capture what the system does *when it is important to understand a sequence of work towards a goal in a context*. We'll talk more about those kinds of systems, called task-oriented systems, in CHAPTER 8. But what about the rest—those “it depends” systems from Section 6.8?

In fact, most interactive systems today are dominated by atomic event semantics. Every keystroke that you enter in a text editor, forms editor, or web page is of that nature. Most mouse-click operations are of that nature. What characterizes these interactions is that they are difficult to describe in terms of any algorithm that is of significance to the user. Because it is implemented on a Von Neumann machine, such functionality is of course implemented as an algorithm in the end, and we can of course find its code in the program. But we won't find it in the end user's head. What we instead find in the end user's head is a data model—a model of the data or form of their world, where each piece of data is “smart” and knows how to

do its domain duty in a way that appears trivial and atomic to the user. The computer not only augments the end user's information processing power in a domain (if it even does that), but instead helps the end user master and manage the data. Deleting a shape in a shape-drawing program is an example.

### 7.2.1 Domain Objects

O.K. let's see what the architecture of an atomic event system looks like. Let's start with the end user mental model. What is the user thinking? The user's worldview is dominated by the data structure at this point. In our shape editor, it's the shapes. In a text editor, it's the text. These are our good friends the domain objects from CHAPTER 5, and they will always be with us (the objects at the bottom of Figure 20). What else?

### 7.2.2 Roles, Interfaces and the Model

Consider the shapes editor again. What domain object should support the **delete** function? Maybe it is an operation on a shape that removes it from the screen. But what if several shapes are highlighted, and the user presses delete? Now it is an operation on a collection. Collections of objects live in the domain model; groups of graphical objects are a common structure in shape programs. We could replicate the operation and add a lot of context-sensitive complexity—but, instead, we return to the user mental model. To the user, the deleted “things” are both special cases of *selections*. When one deletes something, what one deletes is the current *selection*. Now, a **selection** isn't a class in the Shape hierarchy; it's a *role*. A Shape can play that role of **selection**. So can a **collection** of shapes.

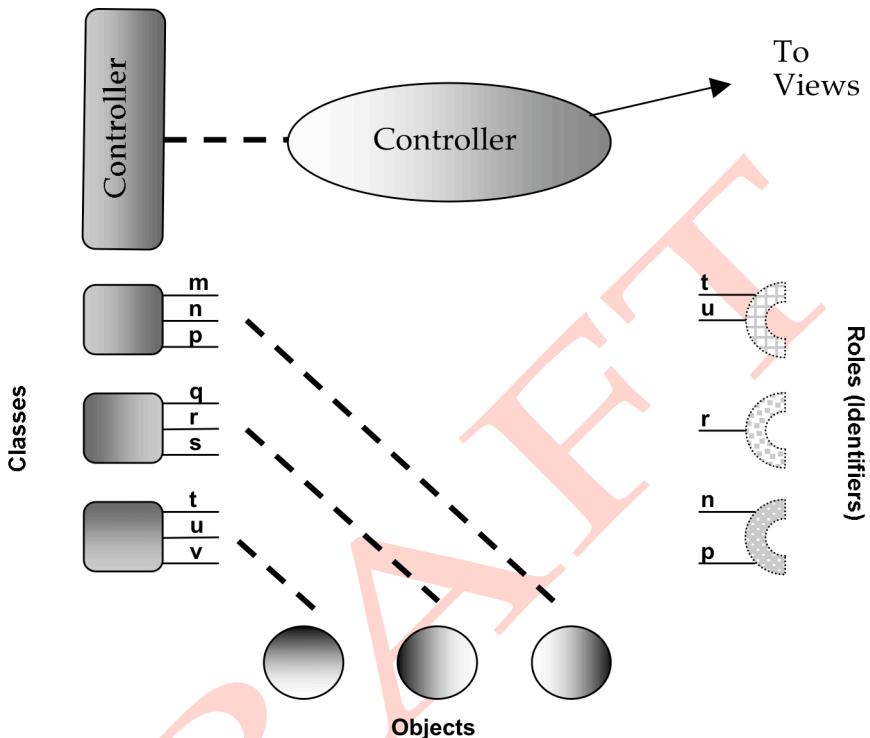
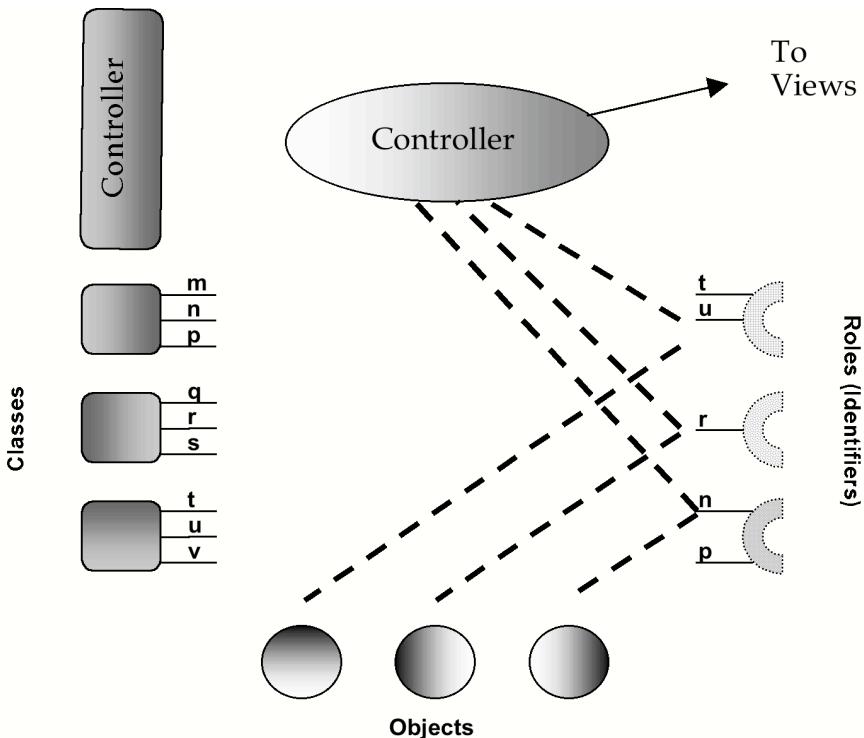


Figure 21: Basic Atomic Event Architecture

We've already captured the selection role during our analysis of what-the-system-does (Section ???). These roles appear as part of the atomic system architecture as ordinary Java interfaces, C# interfaces, or C++ abstract base classes (on the right of Figure 21). Each interface is designed as a partial wrapper onto an existing domain object. In other words, the class for the corresponding object should implement the corresponding interface. The mapping of interfaces to domain classes is many to many.

Roles are a crucial part of the end user mental model. When users think about what they're operating on or playing with, they think objects. When they think about what those objects are going to do or

how they are going to do it, they usually think in terms of the roles those objects play. This will become more important when we use the DCI architecture to start supporting a user mental model that operates on several roles together, but it's true even for simple operations. When I ask the ATM to print the balance for the account currently on the screen, I momentarily forget exactly what kind of account it is. I think of it in its role as a printable account, not as a savings account or checking account. That class distinction is irrelevant to the action of printing.



**Figure 22: Linking Up the Domain Objects with the Controller**

The Controller declares identifiers in terms of these interface types. It will either instantiate the appropriate domain object on response to a user command, or will otherwise bind its role identifier to a suitable domain object (one that implements that role) when an atomic command is invoked. The Controller then carries out its duties in terms of the interfaces on the roles, and the actual work gets done down in the domain objects (**Figure 22**).

Within the MVC framework, these roles become interfaces to the Model objects. The roles and interfaces capture the user notion of what-the-system-does, and the domain objects (and classes) capture the user's mental model of what-the-system-is. We are simply inter-

preting MVC as an architectural framework that stores and organizes the results of our domain design and Use Case analysis.

### *Example*

Let's look at our simple financial system as an example. Our account holder wants to print the balance on the account currently displayed on the screen. How did it get there? Right now, we don't care. The analysts know that there are Use Case scenarios that bring us to this point, and that we want to give the user the ability to print the balance on that account.

The term "account" here refers an instance of one of any number of account types. When in a situation where it can be printed, it plays a specific role that is characterized in part by its ability to print its balance. What do we call that role? We could very simply call it **AccountWithPrintableBalance**. The role obviously supports the responsibility of printing the balance but it could in theory support other responsibilities. And an account in this state is likely to take on other roles at the same time, such as having the ability to disburse or receive funds.

The design is simple. The logic in the Controller will be written in terms of identifiers that represent roles; in most programming languages, these will reduce to declarations in terms of interfaces (or abstract base classes in C++). The APIs of those interfaces express the business functions that map from the end user's mental model of the task being accomplished.

At some time during the program execution those identifiers will become bound or re-bound to objects. Those objects are instantiated from classes whose member functions implement the method declarations in the interfaces. When implementing a new Use Case scenario, it is up to the designer to create and fill in such methods.

The code for the role could be as simple as this:

```
class AccountWithPrintableBalance {
public:
    virtual void printBalance(void) = 0;
};
```

As regards **Figure 22**, this is one of the methodless roles on the right of the picture. The Controller is likely to have a method within it that responds to a print command, and within the scopes accessible to that method we'll find an identifier declared in terms of this methodless role. The logic in the controller binds this identifier to the account object of interest, and uses it to invoke the `printBalance` method.

We'll get to the domain classes soon—Section 7.3.

### 7.2.3 Reflection: Use Cases, atomic event architectures, and algorithms

In general, analysis starts with User Stories, becomes a bit more disciplined in Use Cases, which can be broken down into scenarios, and carried into design as algorithms. There is an important shift between Use Case scenarios and algorithms. Scenarios are about *what*; algorithms are about *how*. In general, design isn't so simple that we can simply write down Use Case scenarios in the code. For example, we always write code sequentially, and if there are two steps in an algorithm, we can say which will precede the other. Such steps may correspond to end user activities and the end user may not care about the order in which they are executed.

As a specific example, to deposit to your checking account the system must know both the amount you are depositing and the account to which it is being deposited. The requirements may, in general, not say anything about which of these two is done first. However, because the code implements concrete decisions, it will choose an ordering constraint on the scenario that is not present in the requirements. The algorithm may also implement non-functional re-

quirements as part of the algorithm that are stated separately as rules in the requirements. Use Cases have a place both for such essential system behaviors and for such rules.

However, in atomic event systems, the individual behaviors are usually simple enough that there is very little sequencing. We will return to this issue in Section 8.4.1.

#### 7.2.4 A special case: One-to-many mapping of roles to objects

Sometimes an end user envisions a role whose responsibilities actually belong together as a role, but which for technical, historical or other reasons, map onto multiple objects. For example, an end user may view an audio headset as a single device playing a single role, whereas the hardware actually has a driver object each for the microphone and the earphone. In this case:

1. Keep the AudioHeadset role, implementing it as an interface or abstract base class;
2. Create a new AudioHeadsetAggregate class. This class will coordinate the Microphone and Earphone objects through a HAS-A relationship
3. Create an instance of AudioHeadsetAggregate and bind it to an identifier declared in terms of AudioHeadset

It's a quick and dirty solution, but it works. The opposite case (which is more obvious, perhaps: the user thinks of the microphone and earphone as being separate roles with separate muting and volume controls, though those functions are controlled by methods on a single driver object) can also be easily handled: both an Earphone and Microphone role can be bound to the same driver object.

---

### 7.3 *Updating the Domain Logic: Method elaboration, factoring, and re-factoring*

Now we have enough context to update the architecture!

For there to be domain objects, there must be domain classes (unless you are programming in an unclassful language such as `self`). The actual workhorses of an atomic event system, the domain classes live behind the abstract base classes produced in our what-the-system-is activities of CHAPTER 5.

The central guiding Lean principle at work in the domain architecture is poka yoke: enabling constraints. As a programmer is rushing to get a feature to market they are likely to be less attentive to the gross system structure and form than to the correctness of the business logic. The domain architecture encodes soberly considered form that can guide development in the excitement of new feature development. It is like I am building a house. The architect and carpenters leave space for the stove and oven in the kitchen, but the owners will choose and install the stove only later. When the gas man or electrician arrives to install the stove, they don't have to decide where to put it: it fits in one place. They can install it without disrupting the structural integrity of the carpenter's work, or the aesthetics or kitchen workflow vision of the architect. Those parts of the domain architecture that the project imports from the market or from other projects reflect another Lean principle: leveraging standards—the real-world analogies to the stove being standard stove widths and standard fittings for electrical and gas connections.

Do I sometimes need to trim or cut the cabinets a bit, or add gas or electrical adaptors when installing a stove? Of course. Architecture isn't meant to be cast in stone, but to be an overall guiding light that makes it difficult to create errors of form. A software architecture sometimes needs minor adaptation to accommodate unforeseen needs of a Use Case. The larger such accommodations are, the more the need for an “all hands on deck” session to assess the extent of the adjustment on all parts of the system.

### 7.3.1 Creating new classes and filling in existing function placeholders

We fill in class member functions or create domain classes as needed, as new Use Case scenarios call on us to implement their logic. These classes follow many of the common practices you know from good object-oriented design. The designer can organize these classes into class hierarchies to express knowledge about the organization of domain entities or simply to gain the common benefits of inheritance, such as code reuse. One can argue whether the derived classes—or for that matter, whether any of the classes—should formally be thought of as part of the architecture. After all, architecture is about form, and classes are about structure. However, this is a rather unhelpful argument to have or to resolve, and it's one of those wonderful decisions about which you can let the manager flip a coin. Of course, in the end everything matters; what you call things is a matter of what elicits the best communication in your culture. We can't tell you that.

In the atomic event architecture, user actions are simple and usually correspond to short, simple (hence the word “atomic”) operations. Such operations correspond closely to ordinary object methods. One of the easiest, Agile ways to add atomic event functionality is to embed the user-focused feature code in the domain objects. This is a particularly relevant technique for small-team projects, such as one might find working on a real-time device controller.

Filling in the domain member functions is straightforward with domain experts' insight or programmers' familiarity with the business. Requirements from the Use Cases can guide the implementer, but the implementer should not focus too closely on any single Use Case, keeping the broad domain needs in mind. On the other hand, later re-factoring can evolve today's version of the method (written for this Use Case) into a more general method. Because domain design drove the structure of the class API, such re-factoring doesn't erode the architectural integrity of the interface.

## Domain Analysis



```

class SavingsAccount:
    public AccountWithPrintableBalance
{
public:
    SavingsAccount(void);
    virtual Currency availableBalance(void);
    virtual void decreaseBalance(Currency);
    virtual void increaseBalance(Currency);
    . . .
    void printBalance(void);
    . . .
};

```

Added  
from  
Use Case →

**Figure 23: Atomic Member Functions**

### *Example*

Continuing our example from Section 7.2.2, let's look at what the class declaration would look like. Consider an account `SavingsAccount` that we want to code so it can play the role of `AccountWithPrintableBalance`. The code could look as simple as this:

```

class SavingsAccount:
    public AccountWithPrintableBalance
{
public:
    void printBalance(void);
    . . .
};

. . .

void SavingsAccount::printBalance(void)
{
    // do the actual printing on the printing device
}

```

```
    . . .
}
```

Up until this time, `SavingsAccount::printBalance` may have been just a stub that fired an assertion or threw an exception—an architectural placeholder that anticipated the arrival of this Use Case scenario. But now the time has come to code it up—just in time, in line with Lean principles. We will do the same for all other account types that must support the **AccountWithPrintableBalance** role.

### 7.3.2 Back to the future: this is just good old-fashioned OO

Such an architectural style evolves to contain “smart” objects: objects that can do more than lie around like dumb data. They can serve user requests. Take care to honor good cohesion within objects and de-coupling between objects; the “smart-ness” of an object shouldn’t be an excuse to give it dominion over a host of other objects that it controls or coordinates.

We will later contrast this approach with the DCI approach, which teases all the what-the-system-does functionality out of the dumb domain objects, leaving them dumb. That localizes the more rapidly changing functional logic elsewhere, where its maintenance doesn’t become entangled with the domain code. In fact, the atomic event architecture has a liability in that it fattens the interfaces of the domain classes and makes them less “domain-y”—they become hybrid homes for what-the-system-does-and-is alike.

### 7.3.3 Tools [CRC Cards]

### 7.3.4 Factoring

Factoring is a simple technique that helps grow the architecture over time. You may notice after a while that method-ful role after method-ful role contains the same logic, and that the logic is inde-

---

pendent of the type of the class into which it is injected. That logic can be factored into a base class.

### 7.3.5 A Caution about Re-Factoring

Re-factoring is a time-honored and effective technique to keep code clean locally. Try to develop a habit of leaving the code cleaner than when you found it; you'll thank yourself later. Robert Martin's book *Clean Code* relates good tips not only about re-factoring, but about other coding conventions that reflect key lean principles [Mar2009].

Re-factoring is relatively ineffective at fixing architectural problems that span architectural units. Re-factoring should leave code functionality provably invariant, or at least arguably invariant, while improving its structure or expressiveness. It's easy to make such arguments within class scope, particularly if you have a good re-factoring browser that avoids accidentally causing an identifier reference to become bound to a declaration in a different scope than intended. But it is almost impossible to reason about functional invariance when you start moving definitions and declarations across class boundaries.

Will you have to bite the bullet and make such adjustments to your code? They're hard to avoid completely. But a good up-front architecture can reduce them. To argue that you should start with a casually or briefly considered design and then let re-factoring bring you to good code structure recalls an old adage from my grandfather: Hope is not a plan.

## 7.4 Documentation?

## 7.5 Why all these artifacts?

If you're a good old-fashioned object-oriented programmer who has learned to live lean with just objects and classes, all these additional artifacts may look superfluous (and it's going to get even just a little bit worse in CHAPTER 8). Why do we introduce roles?

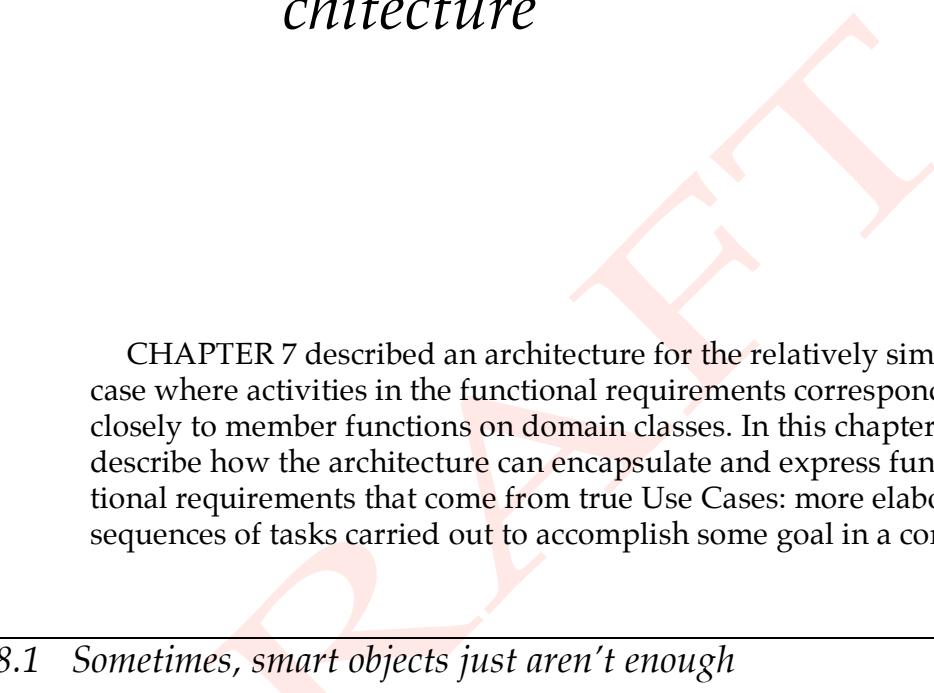
One reason is that roles usually figure important in the end user's mental model, as described in even some of the trivial examples in this section. Another reason is that as we start to handle more exotic end-user operations when we discuss task-oriented styles, the roles become an important locus of form that is crucial from a user's perspective: an encapsulation of what the system *does* in a particular Use Case. For the sake of uniformity, most of the time we use roles and interfaces even for the simple case.

Another reason for roles is that we want to explicitly express the form (which is what architecture is all about) separate from the implementation. It can become difficult or impossible to reason about the fundamental structure of the system if the code obscures or obfuscates it. Being able to tease out abstract base classes and roles separate from their derived classes allows us to reason about form. This is important for anyone trying to understand the big picture in a large system—and that implies most of us. It's a key Lean principle: seeing the whole.

Software interfaces are a time-honored architectural practice to help decouple parts of the system from each other as well. Such decoupling is a central admonition of the GOF book [Ga+2005, p. 18] but it has appeared earlier in dozens of texts.

Because we often want to separate the behavioral code from the basic platform code, it is sometimes a good idea to use the DCI architecture even when dealing with requirements that look like atomic events. It's your choice. We talk about DCI in the next chapter. Right now.

# *Coding it Up: the DCI Architecture*



CHAPTER 7 described an architecture for the relatively simple case where activities in the functional requirements correspond closely to member functions on domain classes. In this chapter we describe how the architecture can encapsulate and express functional requirements that come from true Use Cases: more elaborate sequences of tasks carried out to accomplish some goal in a context.

---

## *8.1 Sometimes, smart objects just aren't enough*

At the beginning of CHAPTER 7 we described the kinds of programming structures well-suited to a Model-View-Controller architecture, such as the common primitive operations of a graphical shapes editor. In operations such as moving or re-coloring a shape, the algorithmic structure is trivial relative to the program data structure. The Controller can catch the menu selection or mouse button push that indicates a certain command given the context where the gesture occurred, and it can directly dispatch a request to the right model (domain) object to field that request. Clever designers that we are, we will have enhanced the API of the domain object so that it supports such requests in addition to the “dumb data” operations we expect all domain objects to support.

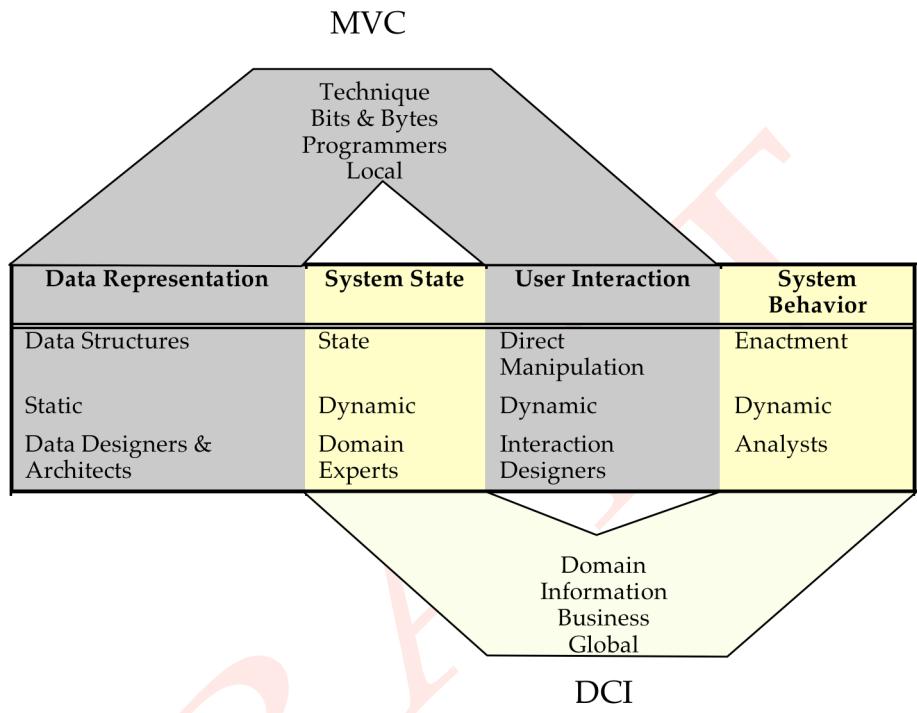
But computers help us do more than just store data; sometimes, they can tackle complex tasks that actually make the computer seem pretty smart. In most of these cases the end user is thinking of some goal they want to attain through a short sequence of tasks. (It's still a *short* sequence: if the sequence starts verging into coffee-break-duration territory, we don't have an Agile interactive program on our hands any more, but a batch program in disguise). *The sequence of tasks achieve some goal in a context.* Now, we are firmly in Use Case land: it is exactly these kinds of tasks that Use Cases capture.

---

## 8.2 Overview of DCI

Trygve Reenskaug's DCI system offers an exciting alternative to good old-fashioned object-oriented programming that encapsulates the what-the-system-does perspective on a per-scenario basis. DCI stands for Data model (the underlying object model that we developed in CHAPTER 5), Context model (a model of what objects play what roles for a given use case scenario) and Interaction (the task ordering within an individual scenario). The goal of DCI is to separate the code that represents the system state from the code that represents system behavior. This separation is related to but different than MVC's split between data representation and user interaction (**Figure 24**).

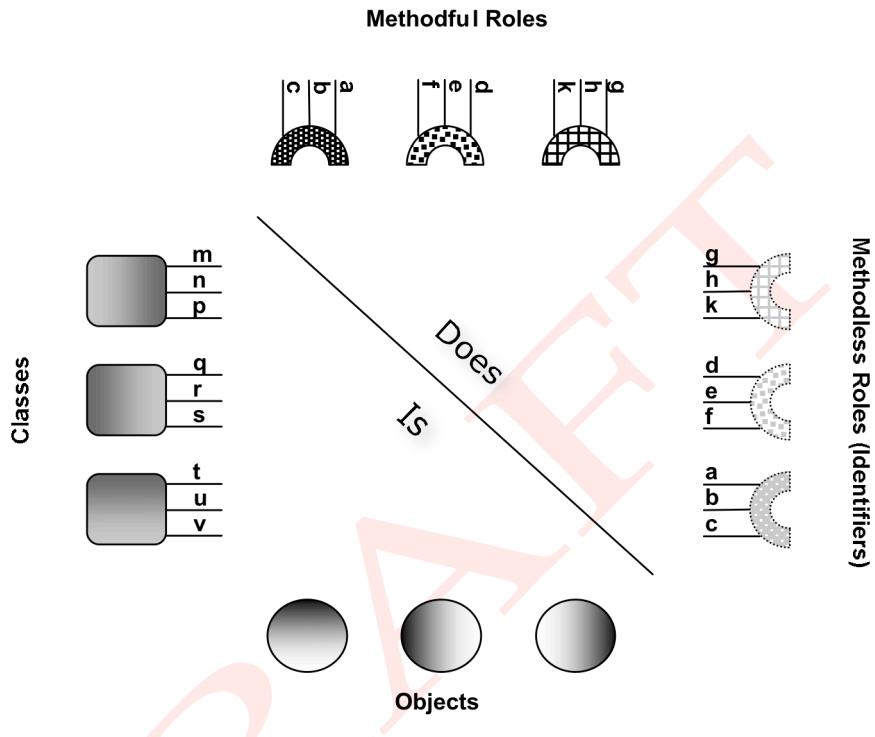
MVC and DCI are designed to work together as a way for the programmer to reason about the end user's mental models and to capture those models in the code.



**Figure 24: Relationship between MVC and DCI**

### 8.2.1 Some parts of the user mental model we've forgotten

Think of it this way: Every piece of software has several structures inspired by real-world, business, and customer concerns and perspectives. The domain model we built in CHAPTER 5 is one example of such a model, one that captures the underlying essence of the business structure. The roles we developed in Section 6.6 reflect the end user model of how the program works. Both of these are rather static structures.



**Figure 25: Basic DCI Architecture**

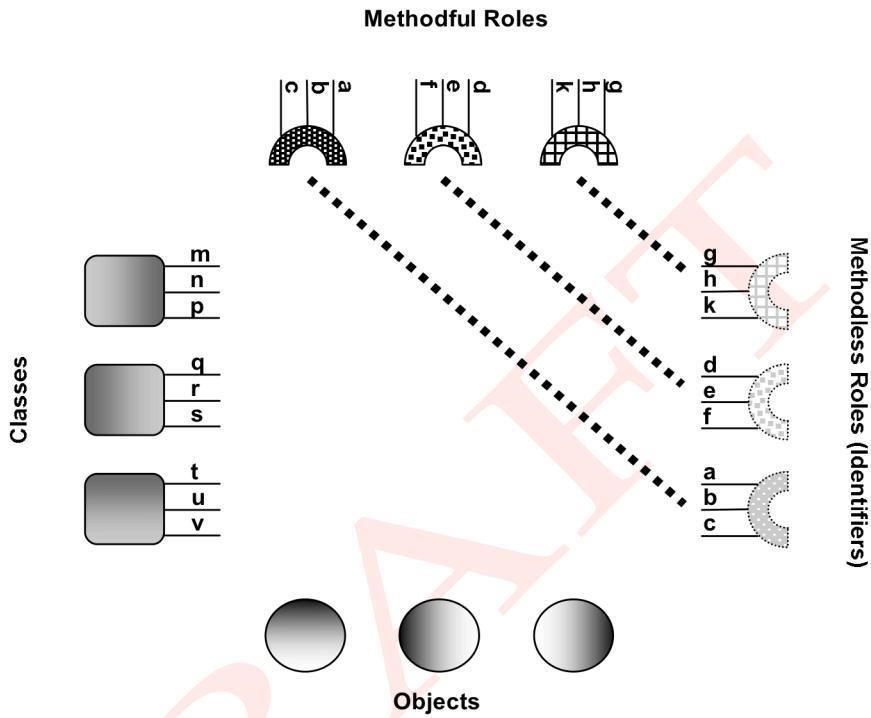
In the days of procedural programming we used procedures as a natural mechanism to program enactment—what the program does. Object-orientation has all but outlawed them. This unnecessary taboo has quashed the expressiveness of object-oriented designers for thirty years. Curiously, the techniques that have recently evolved around object orientation have slowly brought us back to where we can again capture algorithms in an object-oriented framework.

You'll remember our simple picture from earlier in the book (Figure 4 on page 29, for example) that splits the design world into what the system is and what the system does. **Figure 24** is a re-take of that picture. In the what-the-system-is part, we have our familiar

friends the classes and the objects (and especially the objects) from CHAPTER 5. On the other side of the line, we find artifacts called *methodless roles* and *method-ful roles*. The methodless roles are also old friends: we created those in Section 6.6 as a way to capture the end user's cognitive model of the action. Those roles came directly from Use Case actors. In Java and C#, they become interfaces; in C++, we cheat a bit and use abstract base classes. Together these roles identify the *functional architecture* of the system. They document and codify the contracts between system parts by which they interact to do the end user's bidding. They are form, not structure: the form of enactment.

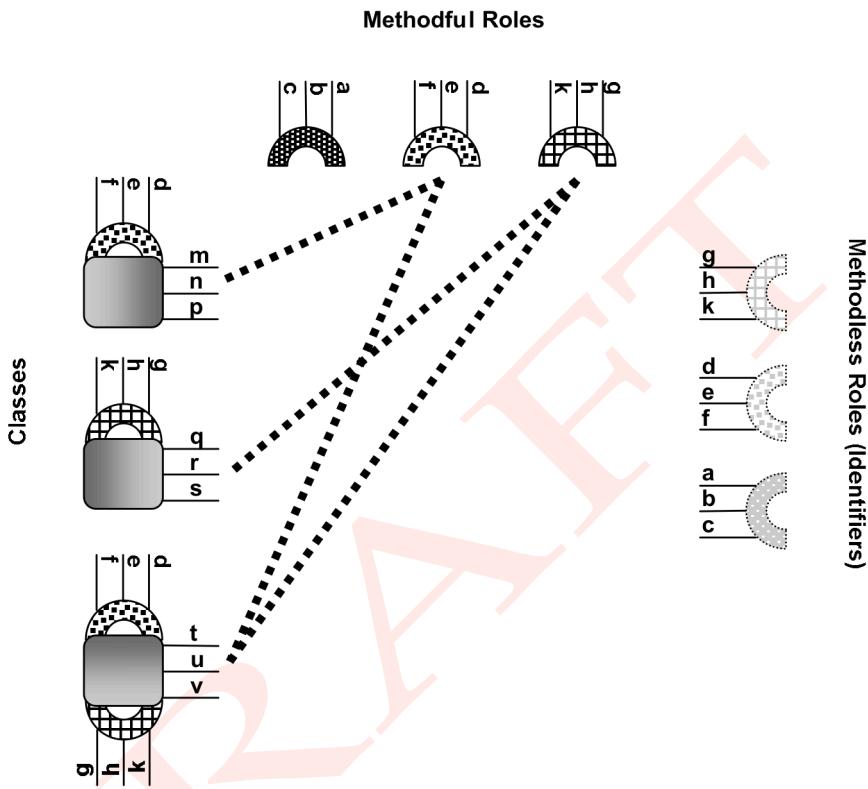
### 8.2.2 Enter method-ful roles

The method-ful roles are new animals here. From the outside, they look just like the methodless roles. But unlike the methodless roles, their methods are filled in with code. These roles carry the real knowledge of what the system *does*. The Use Cases live in these method-ful roles. In classic object orientation, the system behavior lies distributed throughout the domain objects, where it is fragmented and polymorphic and a lot of other things that sound good but which make any given Use Case scenario devilishly hard to understand. These method-ful roles bring the fragments of the algorithm together in one place that maps onto the end user world model. We call this model the *volitive* world model: the end user's model not of the form of the domain or of the data, but of the form of the action.



**Figure 26: Association of Identifiers with Algorithms**

We still want artifacts that represent the pure form of the architecture, and that capture the interface of the roles apart from the implementations of their algorithms. We desire these methodless roles because these interfaces should change less often than the methods themselves. Agile is about managing change, and the interfaces have a different rate of change than the methods. We might have architects oversee the interfaces, and application programmers take care of the methods. In the ideal case, user interaction code (often in the Controller) will usually talk to a method-ful role through an identifier declared as a methodless role: an interface (**Figure 26**).



**Figure 27: Using Traits to Inject Algorithms into Domain Classes**

Because method-ful roles have implemented methods and member functions, they can't be Java or C# interfaces. We're not in Kansas any more. Our programming languages are missing a feature to express these concepts. We'll get to that in a second.

These roles capture what the objects *do*. Well, *what* objects? Ultimately, it's the domain objects. We create domain objects without any interfaces to support the what-the-system-does architecture, because domain objects are there to capture the domain structure only. We need to somehow combine the domain objects' code with the

code that runs the scenarios. But to do that, we will *inject* the scenario code into the classes from which those objects were created.

Where does that code come from? We can count on there being a single, closed, maintainable copy of that code in the method-ful role. We can inject that code into each class whose objects must take on the corresponding role at some point during their lifetime (**Figure 27**).

### 8.2.3 Tricks with Traits

How do we do that? We can use a programming approach called *traits* [Sch2003]. A trait is a holder of stateless methods. Traits use language tricks (different tricks in different languages and environments) to effectively compose classes together. We treat one class as though it is a role, while the other class holds its identity as a domain class, and we inject the logic of the former into the latter. We can make a method-ful role a trait of a class to inject its functionality into that class. This leaves method-ful roles generic, apart from the class into which they are injected—and the word “generic” rings true here in several implementations of traits. Methods of the role can invoke methods of the domain class into which they are injected. We’ll show you details in the next section.

Now we can finally get back to objects. The objects come and go in the system as initialization and business scenarios create them. Each object can take on all the roles for all the scenarios we have designed it to support. Of course, multiple objects will often (in fact, usually) work together to support any single given scenario. When there is work to be done, where do we go to get it done? How does that object know of the other objects with which it is supposed to collaborate during an enactment?

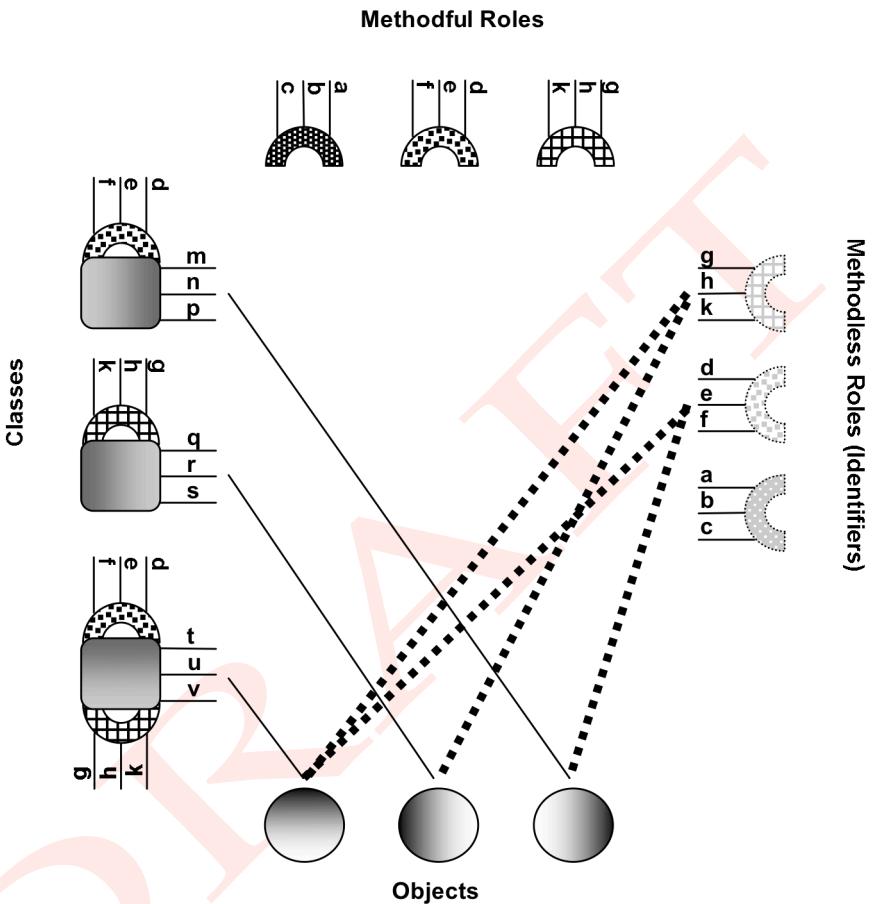


Figure 28: DCI Object Instantiation and Identifier Bindings

#### 8.2.4 Context Classes: One per Use Case Scenario

For every Use Case scenario, we define a Context class. A Context knows the roles that are involved in any given use case, and its implementation holds identifiers declared in terms of the methodless

roles (interfaces or abstract base classes) for those roles (**Figure 28**). For any given invocation of a Use Case scenario, those roles must be bound to objects. Those bindings can be externally retrieved through public methods of the respective Context object.

When a use case scenario starts up, we instantiate its Context object. The Context object can use whatever means available to it to bind its role identifiers to whatever objects exist in the system. Of course, it uses contextual knowledge to ensure that a given role is bound only to an object that supports that role's interface. Programming languages with static type checking can help avoid violations of this assumption and can produce compile-time errors if the designer tries to create the wrong associations. And then the Context object just invokes the first method of the first role. If one object in the scenario needs to communicate with another, it retrieves the identity of that object from the Context object, through the public methods that map the Use Case roles to the objects playing them in the current enactment.

This may sound a bit contrived, but let's look at some of the design tradeoffs here:

1. Polymorphism: It's gone. Actually, we have moved it to the Context object, which is now choosing an object to satisfy a given set of method invocations instead of leaving the method choice to calling time. So we remove much of the uncertainty of the where-will-this-polymorphic-dispatch-end-up problem. There is still a degree of uncertainty because of the dynamic binding of Context role identifiers. However, the binding is under explicit control of the business logic in Context, so we can reason about the dispatching in business terms. Besides, the scenario logic is all in the method-ful roles—for any given role and method, there is just one implementation. The polymorphism plays out only in the method-ful roles' invocation of methods on `self`, methods that are deferred to the domain object. Trygve Reenskaug, the inventor of the DCI approach, says: "We solve the problem by suspending polymorphism for the methods that are essential for the integrity of the problem."

2. There is a nice degree of compression going on here. All objects that play a given role shall process the same interaction messages with the same methods. That's Lean.
3. Code readability: Algorithms now read like algorithms. You can look at the code of a method-ful role and reason about it, analyze it, maybe even stub it off and unit test it.
4. The rapidly evolving what-the-system-does code can evolve independently of the (hopefully more stable) what-the-system-is code. The domain structure no longer defines nor constrains the run-time structure. Early in system development you'll be filling in methods of both kinds of classes together, but the domain-driven design will help the domain class methods become stable over time.
5. We now have a place to reason about system state and behavior, not just object state and behavior.
6. Though not simplistic, DCI is simple. It is as simple as it can be, if our goal is to capture the end user's mental models in the code—end user engagement is crucial to success in Agile. That's usually a good thing, because it helps contain evolution well and helps us understand and communicate requirements better. In addition, it was the whole goal of object-orientation in the first place. In fact, having a usability focus in design, combined with object orientation, together with Agile development, and the DCI paradigm—one quickly realizes that they are all just different facets of the same thing.
7. We make the code a little more failure-proof by keeping the domain interfaces separate and minimal. They are no longer polluted with the domain operations as was the case in the atomic event architecture. The domain classes serve as simpler, clearer constraints and guidelines on design than in ordinary object-oriented programming.

What we have done here is to tease out a different kind of commonality than we talked about in CHAPTER 5. The roles and Con-

text objects define a recurring commonality of *behavior* or *algorithm* that is independent of the objects that carry out those algorithms, or the classes of the objects that carry out the algorithm. Those commonalities relate closely to the end-user mental model (volitive model) of system behavior.

Context objects sometimes can play roles that come from the end user mental model! Consider the public interface of a Context object: it provides the API by which we start off individual Use Cases. Let's say that we group related Use Cases ("related" in the sense that they work with the same role combinations and that they use similar strategies to map roles to objects) together in one Context object. That object now represents a collection of related responsibilities. That's an object in the end user mental model, or more precisely, the role played by some object. We'll explore this concept further in Section 8.5.

In the following sections, we illustrate how to use DCI with the simple banking funds transfer example from Section 6.9.

---

### 8.3 DCI by Example

#### 8.3.1 The Inputs to the Design

We kick off a DCI design when the business decides that the software needs to offer a new service. These services are most often expressed in terms of scenarios. DCI is optimized for designs where the form of the scenario is as important as the form of the underlying data model. We think of a banking transfer (primarily) as an algorithm that operates on two or more financial instruments (which are the secondary focus—we are concerned about their details only with respect to their roles in the transfer).

DCI starts with two major inputs that together capture the end user mental model of their world. The first input is the domain model, which we reduced to code in Section 5.4. In addition to the code, we have the domain documentation that was also developed

in CHAPTER 5, and may also have some domain-level patterns that describe high-level system form. For example, a banking system may include financial instruments, transactions, audit trails, and other actuarial artifacts that capture the current state of holdings and investments.

The second input is the actor model, which we developed in Section 6.6. The actor model conveys the user's understanding of system dynamics. For example, a funds transfer in a bank would involve financial instruments as actors called source account and destination account.

In addition to these two inputs, we also need the Use Case scenario that we are implementing. The Use Case will become an algorithm that we reduce to code for the new business service.

### 8.3.2 Use Case Scenarios to Algorithms

At the end of Section 6.6 we discussed the path that goes from User Stories to Use Cases to algorithms. Use Cases are overkill for atomic event architectures, but they capture important scenarios and variations for more complex requirements. It's important to realize that a Use Case is not an algorithm, and that the algorithm in the code reflects sequencing decisions, implementation of business rules, and other details that aren't explicit in any given Use Case scenario.

Let's start with the Use Case for transferring money between two accounts in **Figure 29**, which we introduced in Section 6.9. This is a classic Use Case that captures the users' (Account Holders') intent, the responsibilities they must fulfill to carry out that intent, and the system responsibilities that support the account holders in achieving their goal. This Use Case builds on two other Use Cases: one for selecting the accounts for the transfer (Select Accounts), and one where the transfer actually takes place (Move Money and Do Accounting).

**Use Case Name:** Transfer Money

**User Intent:** To transfer money between his or her own accounts

**Motivation:** The Account Holder has an upcoming payment that must be made from an account that has insufficient funds

**Preconditions:** The Account Holder has identified himself or herself to the system

**Basic Scenario:**

1. Account Holder requests an account transfer
2. System displays valid accounts
3. Account Holder Selects Accounts
4. System displays selected account balances
5. Account Holder chooses the amount to transfer
6. System moves money and does accounting

**Variations:**

- 1a. Account Holder has only one account: tell the Account Holder that this cannot be done
- 2a. The accounts do not exist or are invalid...

**Postconditions:**

- ✓ No money is “lost” in the transaction
- ✓ The periodic statements reflect the exact intent of the transaction (a transfer is a transfer — not a pair of a withdrawal and a deposit)

**Figure 29: Transfer Money Use Case  
(Repeat of Figure 15)**

We will examine the Move Money and Do Accounting Use Case in more detail below. How will that Use Case know which accounts to use? The Transfer Money Use Case must be stateful: that is, it must remember the decision made in step 3 for use in step 6. Here are three possible approaches to passing this information:

1. The method-ful role for the Transfer Money Use Case remembers the accounts and uses them when it sets up the Context object for the Move Money and Do Accounting Use Case. We talk more about this approach in Section 8.3.5.
2. The Controller takes on this responsibility somewhere over in MVC. However, the Controller is architecturally quite a bit distant from this Use Case logic, and that implies bad coupling and a coordinated update.
3. The Transfer Money Use Case stores the information in the Model. If the Model contains a transaction object that scopes the account selection and transfer, this could work out well. However, it may require some ingenuity on the part of the developer if this transaction time is overly long. It also greatly complicates things if multiple applications simultaneously use the same Model object.

In **Figure 30**, we recall the Move Money and Do Accounting Use Case from Section 6.9. The Use Case captures the business interactions between the Account Holder and the system. We'll use it as our coding example, showing how to capture those business interactions in code. Before coding, we have to translate the Use Case *scenario* into an *algorithm* a computer can execute.

We describe the algorithm in terms of the actors (which become roles, as discussed in the next section) from the previous step. We will start thinking of the actors not as the original ones in the real world, but as software artifacts in the program. This is where the transition happens that makes object-oriented programming what it is: a way to capture the end user conceptual model in the program form. We'll formalize this transition in the next step when we choose a concrete expression for roles that captures the actor semantics, but we try to look ahead a bit here already.

**Use Case Name:** Move Money and Do Accounting

**User Intent:** To transfer money between his or her own accounts

**Motivation:** The Account Holder wants to move money between funds to avoid restrictions on certain kinds of transactions on the source account, or to take advantage of the convenience or financial gain resulting from the money resting in the destination account.

**Preconditions:** The Account Holder has identified the Source Account, Destination Account, and the amount to be transferred

**Basic Scenario:**

1. System verifies funds available
2. System updates the accounts
3. System updates statement information

**Variations:**

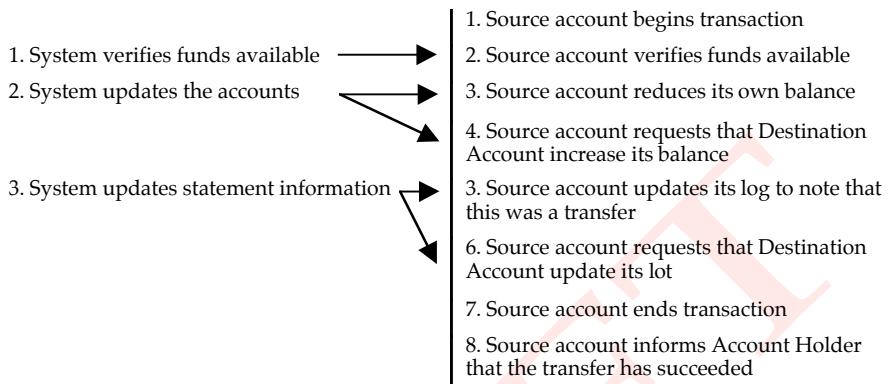
- 1a. Funds not available in the Source Account: system displays an error message

**Postconditions:**

- ✓ No money is “lost” in the transaction
- ✓ The periodic statements reflect the exact intent of the transaction (a transfer is a transfer — not a pair of a withdrawal and a deposit)

**Figure 30: Move Money and Do Accounting Use Case  
(Repeat of Figure 16)**

The Use Case in **Figure 30** is a view from the perspective a single stakeholder: the Account Holder. Yet the code must run for all stakeholders. While there is no concept of a transaction in the mental model of the Account Holder, there is such a concept in the mental model of the Actuary. Such complications make the code a bit more complicated than the scenario.



**Figure 31: Scenario-to-Algorithm Mapping**

The actual algorithm may look like this; it is one of several possible viable implementations:

1. Source account begins transaction
2. Source account verifies funds available (notice that this must be done inside the transaction to avoid an intervening withdrawal!)
3. Source account reduces its own balance
4. Source account requests that Destination Account increase its balance
5. Source Account updates its log to note that this was a transfer (and not, for example, simply a withdrawal)
6. Source account requests that Destination Account update its log
7. Source account ends transaction
8. Source account informs Account Holder that the transfer has succeeded

The algorithm reflects a mapping (**Figure 31**) from the Use Case into steps that the computer can execute in a deterministic way (just because it's ordinary procedural code) and which meets business needs that come from other requirements (such as the need to not lose any money in the process, so we use transactions). [TODO: The end user interaction seems a bit out-of-place here—move it to a higher Use Case?]

### 8.3.3 Methodless Roles: The Framework for Identifiers

The scenario refers to a number of actors—concepts that live in the mind of the end user. We want to transfer those to concepts that can live in the mind of the programmer and in the code itself. What are called actors in the requirements domain are called roles in the coding domain.

The translation from actors to roles is quite straightforward. In Java or C#, we code them as interfaces. In C++, we code them as abstract base classes. In Objective-C or Smalltalk, they are just classes. Because we are interested in being lean, and for parallelism with abstract base classes in the what-the-system-is architecture, these roles have no associated code for algorithms or data structure: they are pure protocol. The implementation will come later when we weave algorithms into method-ful roles. This separation gives us the flexibility to define a role architecture up front, building on deep domain knowledge that incorporates foresight from previously built systems, without investing too deeply in implementation.

```

class MoneySink {
public:
    virtual void increaseBalance(Currency amount) = 0;
    virtual void updateLog(string, MyTime, Currency) = 0;
    virtual void transferFrom(Currency amount,
        MoneySource *source) = 0;
};

```

The `MoneySource` role member functions are virtual. In theory, we could bind all the member function invocations at run time by putting all the type information into the templates that implement the traits. However, that would complicate the C++ code and make it less readable. Instead, we use the common C++ approach that abstract base classes define the protocol / interface to a group of classes whose objects will play that role.

### 8.3.4 Partitioning the algorithms across Method-ful Roles

Now it's time to capture the Use Case requirements as algorithms in the method-ful roles. This is the heart of DCI's key benefits to the system stakeholders: that the original requirements are clear from the code itself.

#### *Traits as a Building Block*

Before going into the next section on building method-ful roles, we take a quick diversion here to describe traits in a bit more detail. Our goal in DCI is to separate the Use Case scenario knowledge in one place from the domain knowledge in another, to separate system state from system behavior. Both the behavior knowledge and the domain knowledge have the outward appearance of being collections of behaviors. We think of any such collection as being an object when representing the form of the domain, and as being a role when representing the end user's model of system behavior.

What we want to do in DCI is to compose a role and its algorithms with an object and its support for domain logic. However, few programming languages support roles with methods and few

programming languages let us program objects. Instead, the main building blocks are classes. To a first approximation, what we want to do is to glue two classes together. More specifically, the class representing the roles is a collection of stateless, generic methods. They must be able to work with a somewhat anonymously typed notion of `this` or `self`, because the class with which the role is composed determines the type of the object. The class representing the domain logic is, well, just a class. How do we compose these two? That's what traits do for us.

Schärli did his original implementation of traits in Smalltalk. Each class gets an additional field that implicates the traits with which it is composed, together with other properties of traits that we don't need for DCI. Furthermore, in DCI, we block the possibility of trait overrides in the class. This traits field is used during method lookup if the desired method selector isn't found in the methods of the class itself: the method dictionaries of the injected roles (classes) are checked if the class search fails. We will return to the Smalltalk implementation in more detail in Section 8.7.5 below.

In C++, it's a bit more straightforward. Consider a trait `T` that contains algorithms `t1` and `t2`:

```
template <class derived> class T
{
public:
    virtual void derivedClassFunction(int, int) = 0;
    void t1(void) {
        derivedClassFunction(1, 2);
        . . .
    }
    void t2(void) {
        . . .
    }
};
```

This trait represents the role `T`, a role characterized by its methods `t1` and `t2`. Note that `T` presumes on the class into which it will be injected to support the method `void derivedClassFunction(int, int)`. Of course, a role doesn't have to depend on the

presence of a function in its target class, but it's a common situation that we want to support.

We include the type parameter derived for the common case that the trait wants to communicate business logic in terms of the type of the actual object involved in the Use Case. Making it a parameter leaves the trait generic so that it is possible to decouple its maintenance from that of any domain class in particular. [TODO: make sure that some later example demonstrates this.]

We want to inject this role into classes of all objects that play this role at some time or another so the class gives those objects the appearance of supporting  $t_1$  and  $t_2$  in their public interface. Let's assume that one of those objects is an object of class  $D$ . We inject the role when we declare  $D$ :

```
class D: public T<D>
{
public:
    void derivedClassFunction(int a, int b) {
        . . .
    }
    . . .
};
```

Now all instances of  $D$  will have the appearance of supporting methods  $t_1$  and  $t_2$  in their public interface. Of course, we can inject additional roles into  $D$  using multiple inheritance. It's safe to do this because traits contain no data. Of course, one must be mindful of name collisions and resolve them manually (the C++ compiler will tell you if there is any ambiguity and if you need to do so). It should also be obvious that the trait  $T$  can be injected into other classes as well, while remaining the single, closed definition of the algorithms  $t_1$  and  $t_2$ .

Some languages support traits even more directly. Scala [Od+2008] is one such language; we will discuss its implementation of traits and DCI in Section 8.7.1.

With traits and role injection in hand, let's go on to defining method-ful roles!

## Coding it up

Here we apply the approach of Section 8.2 to this example. This is a boring algorithm to the extent that most of the processing takes place inside one role: the Source Account (`TransferMoneySource`). The Destination Account (`TransferMoneySink`) contains a little logic to receive the funds.

Let's write the C++ code. First we define some macros as a convenient way to look up the current object bindings for the roles involved in the Use Case. We need to look up two roles: the object currently playing the current role (`self`), and the recipient of the transfer. We define macros as follows:

```
#define SELF static_cast<const ConcreteDerived*>(this)

#define RECIPIENT ((MoneySink*) \
    (static_cast<TransferMoneyContext*> \
    (Context::currentContext_)->destinationAccount()))
```

The `self()` macro evaluates to a pointer to *whatever object is playing the current role*. It is used by code within the role to invoke member functions of the role self, or `this`. As we shall see shortly, this allows the code in the role to “down-call” to methods of the derived class into which the trait is injected.

Using macros makes it possible to use role names directly in the code. So we can say something like:

```
RECIPIENT->increaseBalance(amount)
```

and that will find whatever object is currently playing the role of the recipient, and will apply the `increaseBalance` method to it. We could also do this with inline member functions `self()` and `recipient()`, but the function syntax is slightly distracting. Use the member functions if that is your taste.

Next, we create a template that implements the trait for the role.

```
template <class ConcreteDerived>
class TransferMoneySource: public MoneySource
{
```

Now we come to the interesting parts: the role behaviors. We can start with a single, simple behavior that implements the transfer of money from the current role (TransferMoneySource) to the role TransferMoneySink.

```
public:

// Role behaviors
void transferTo(Currency amount) {
    // This code is reviewable and
    // meaningfully testable with stubs!
    beginTransaction();
    if (SELF->availableBalance() < amount) {
        endTransaction();
        throw InsufficientFunds();
    } else {
        SELF->decreaseBalance(amount);
        RECIPIENT->increaseBalance (amount);
        SELF->updateLog( "Transfer Out", DateTime(),
                           amount);
        RECIPIENT->updateLog("Transfer In",
                             DateTime(), amount);
    }
    gui->displayScreen(SUCCESS_DEPOSIT_SCREEN);
    endTransaction();
}
```

The code also accesses the local member function `recipient()`, which returns the object playing the role of the TransferMoneySink. We'll talk about that member function later.

Here is the analogous code for the other trait, representing the TransferMoneySink role.

```

template <class ConcreteDerived>
class TransferMoneySink: public MoneySink
{
public:
    void transferFrom(Currency amount) {
        SELF->increaseBalance(amount);
        SELF->updateLog("Transfer in",
                         DateTime(), amount);
    }
};

```

The classes `TransferMoneySource` and `TransferMoneySink` together capture the algorithm for transferring money from a source account to a destination (sink) account—*independent* of whether it is a savings or investment account. (We'll look at savings and investments accounts below.) The algorithm is a top-down composition of two sub-algorithms: an outer `transferTo` algorithm that invokes in an inner `transferFrom` algorithm. We associate those algorithms with the roles to which they are most tightly coupled. Other than that, they behave just as procedures in a simple procedural design.

Each of these method-ful roles captures its part of the algorithm of transferring money from one identified account to another. Yes, we use procedural decomposition here—or is it just a method invocation? The point is that the right logic is in the right place. The transfer logic doesn't belong to any single account class.

### 8.3.5 The Context framework

We need a `Context` class for each Use Case scenario. The `Context` is the locus of its own set of end-user mental models. Consider our end user Marion who wants to do a funds transfer. Though Marion has the concepts `MoneySource` and `MoneySink` in mind when enacting this Use Case, another part of Marion's brain has reasoned about the transfer in terms of an amount, in terms of Marion's `InvestmentAccount` and in terms of Marion's `SavingsAccount`. The `Context` is, in large part, the mapping between these two perspectives.

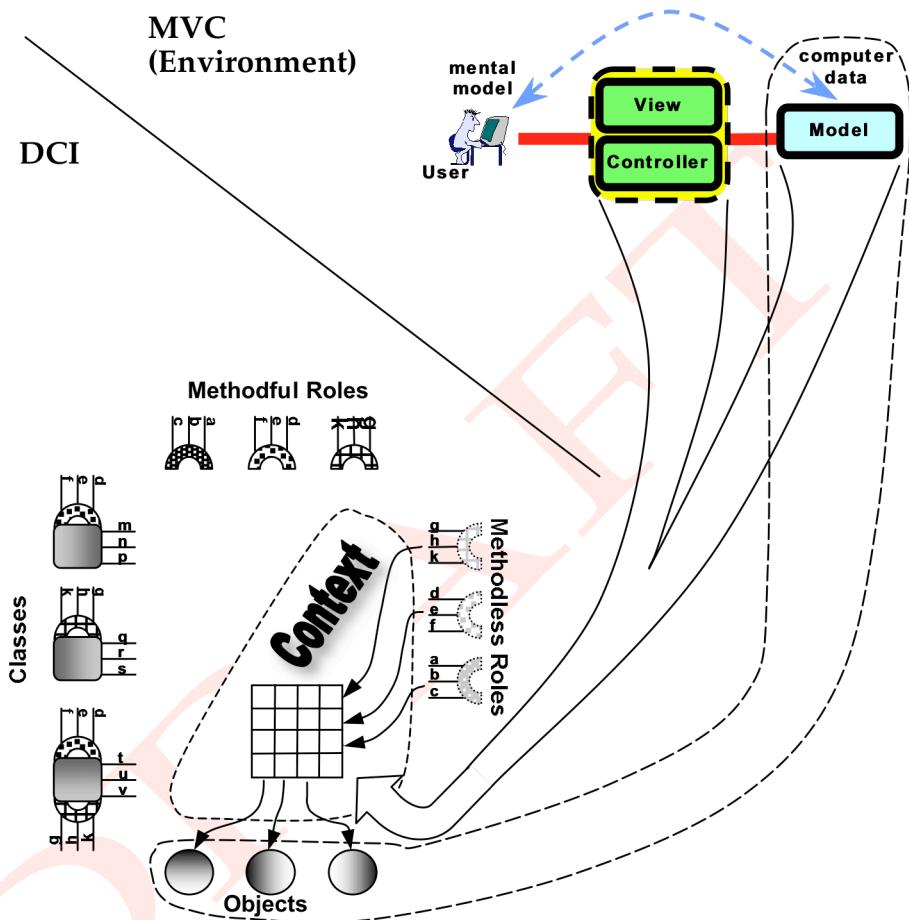


Figure 32: The place of the Context Object

The job of the Context object for a given Use Case is:

1. To look up the actual objects that should participate in this particular Use Case invocation. This is like a “database lookup,” using knowledge at hand to find the actual domain objects that represent the data of interest in this “transaction;”

2. To associate these objects with the roles they play in the current Use Case of this type;
3. To publish those interface bindings for use by the method-ful roles that participate in this Use Case.

Think of the associations between the roles and objects as being like a simple table that the Context object builds inside of itself (**Figure 32**). A fresh Context object, and a fresh set of associations, come together for each Use Case enactment. The table is put together from a combination of data in the *environment* (mainly in the data and the Controller of the MVC code) and the Context object's knowledge of how the Use Case pieces (algorithms, roles and objects) fit together.

Since this scenario affects a money transfer, let's call the class TransferMoneyContext. The TransferMoneyContext class doesn't encapsulate the scenario, but encapsulates knowledge of what actors to bring to the stage for a given scene of the play. It might look like this:

```
class TransferMoneyContext
{
public:
    TransferMoneyContext(void);
    void doit(void);
    MoneySource *sourceAccount(void) const;
    MoneySink *destinationAccount(void) const;
    Currency amount(void) const;
private:
    void lookupBindings(void);
    MoneySource *sourceAccount_;
    MoneySink *destinationAccount_;
    Currency amount_;
};
```

The private data sourceAccount\_ and destinationAccount\_ are declared in terms of the methodless roles MoneySource and MoneySink, respectively. The Context object holds a reference to an object that captures the transfer amount as well, which was likely es-

tablished during a previous Use Case. During any single Use Case enactment, these members hold the binding of methodless roles to the domain objects into which the method-full roles have been injected. Each Use Case is like a performance of a play, and we cast (in the theatrical sense) an actor to play each of the given roles. Here, the roles are represented by the member data of `TransferMoneyContext`, and the domain objects represent the actors. Each actor has memorized his or her script or scripts: that is, we have injected the method-ful roles into each domain object according to the roles it may be called to play. It soon will be time to call them on stage.

Lights, camera—and when we instantiate the `Context` object, we get action.

The `TransferMoneyContext` object is constructed from a call within what DCI calls an *environment*. An environment is an object that starts up a system operation. In most applications, the environment will usually be a Model-View-Controller instance responding to a gesture from an end user. The actual knowledge may be lodged either in the Controller or in the data of MVC's domain model.

When we create an instance of the `TransferMoneyContext` class, it acquires knowledge of the three main objects it will be working with: the object representing the source account, the one representing the destination account, and the one representing the amount of the transfer. If the previous Use Cases selected the source account, destination account, and transfer amount, and those Use Cases were run through or by the Controller, then either the Controller or the system data can remember those selections. The system data are part of the model information in the MVC framework (**Figure 32**). The Controller can retrieve those data from the model and supply them to a new `TransferMoneyContext` class constructor when it is created:

```
TransferMoneyContext(Currency amount,
                      MoneySource *src,
                      MoneySink *destination);
```

Alternatively, the TransferMoneyContext object can do a “database lookup” in the system domain objects, relying on identifiers declared globally or elsewhere in the environment (e.g., from the Controller). We assume the latter for the time being, but we’ll consider other options later.

The TransferMoneyContext needs very little code to set up and execute the money transfer Use Case:

```

TransferMoneyContext::TransferMoneyContext(void)
{
    lookupBindings();
}

TransferMoneyContext::TransferMoneyContext(Currency
amount, MoneySource *source, MoneySink *destination):
    Context()
{
    // Copy the rest of the stuff
    sourceAccount_ = source;
    destinationAccount_ = destination;
    amount_ = amount;
}

TransferMoneyContext::doit(void)
{
    sourceAccount()->transferTo(amount());
}

void
TransferMoneyContext::lookupBindings(void)
{
    sourceAccount_ = databaseLookup(); // maybe an
                                    // investment
    destinationAccount_ = . . . .;    // maybe Savings
    amount_ = . . . .;              // chosen amount
}

```

Notice that the `doit` member function retrieves the identity of the accounts from its own local member functions. Remember that the `SavingsAccount` is also participating in the Use Case, playing the role of the `MoneySink`. How does it get access to the context?

## *Making Contexts Work*

The notion of Contexts is so fundamental to the end user's concept of what's going on at the business level that it deserves the same consideration at the design level as the concepts of `this` and `self` do at the programming level. General-purpose programming languages provide features only to reason about and express the behavior of local objects, but not the behavior of the system as a system (remember **Figure 24** on page 195). It would be nice if programming languages automatically provided a pointer to a Context or environment to each method in the same way that they provide `self` or `this`. Maybe DCI will someday lead language designers down that path, but not today (see the history section at the end of the chapter).

One down side of DCI is that programmers must do some housekeeping to keep things working. Most of this housekeeping can be captured in a few simple practices and rules. Remember these rules when creating a Context object:

1. Create a new Context object for each distinct Use Case. With cleverness and experience, you can start to build class hierarchies of Context objects that reflect some of Ivar Jacobsson's original vision of a type theory for Use Cases.
2. Each Context object should have a default constructor (one with no arguments) that the environment (e.g., a domain data object or a MVC controller) can conveniently instantiate and turn loose to do what it needs to do. You might add specialized constructors that take arguments such as references to participants in the Use Case.
3. Each Context object should have a separate `doIt` (or `enact` or `run` other suitably named) method that runs the Use Case scenario. Alternatively, you can adopt a convention that the Context constructor itself will trigger the Use Case scenario implicitly.
4. Its interface should publish pointers (identifiers) for all roles involved in the corresponding Use Case. The Context object is the oracle for the mapping from the methodless role identifiers

to the objects bearing the method-ful role logic. Every role involved in the Use Case should be able to find any other role involved in the Use Case—and the Context Object is the source.

5. The identifiers for these roles should be typed in terms of the methodless role declarations rather than the method-ful role declarations. This ensures that the APIs between method-ful roles don't depend directly on each other. This is particularly important in languages like C++ with strong compile-time type systems because it limits the compile-time coupling between method-ful roles.

Our goal is to be able to code the methods of method-ful roles so they directly reflect the algorithm we derive from the Use Case. We don't want to clutter the code with explicit logic to map roles to objects. Ideally, we refer to an object that plays a given role knowing only the name of the role it plays. We assume that the system has taken care of the rest. These role names are exactly the bindings made available in the interface of the Context object. For example, in the money transfer example, the `TransferMoneyContext` member functions `sourceAccount` and `destinationAccount` name the roles with which we are concerned. So if we make the Context object available, it is only one step away to access the role handles. Here, we propose four ways to pass a suitably typed Context reference to the places it is needed in the code. The fourth and last way is the recommended way, but your programming culture or application may suggest that one of the other three is better for you.

1. *The simple case where role bindings are already arguments to the methods in the method-ful roles.* This is how our simple funds transfer example works. The Context object itself passes the individual methodless role identifiers as arguments to the method-ful roles that need them:

```

TransferMoneyContext::doit(void)
{
    sourceAccount()->transferTo(amount(),
                                destinationAccount());
}

```

This gives the method-ful roles (such as `sourceAccount()` here) access to all other roles with which they communicate to achieve the overall Use Case scenario.

2. *Pass it to the method-ful role interface of each domain object when the Context object maps the methodless roles to the domain object at the beginning of the Use Case.* This approach simulates the approach used by the original BabyUML implementation of DCI (see Section 8.9.2). Consider the original code from our simple funds transfer example above:

```

void
TransferMoneyContext::lookupBindings(void)
{
    sourceAccount_ = databaseLookup();
    destinationAccount_ = . . .
    amount_ = . . .
}

```

We can add APIs to role interfaces that take a Context object as an argument: We must augment the interfaces of the methodless roles to support the `setContext` invocations invoked from the Context objects:

```

class MoneySource {
public:
    virtual void transferTo(double amount) = 0;
    virtual void decreaseBalance(double amount) = 0;
    virtual void payBills(void) = 0;

    // Context Stuff
    virtual void setContext(PayBillsContext*) = 0;
    virtual void setContext(TransferMoneyContext*)= 0;
};

```

```

class MoneySink {
public:
    virtual void increaseBalance(Currency amount) = 0;
    virtual void updateLog(string, Time, Currency)= 0;

    // Context stuff:
    virtual void setContext(PayBillsContext*) = 0;
    virtual void setContext(TransferMoneyContext*)= 0;
};

```

These APIs cache the information locally for later use, but we will implement the APIs in the methodful roles that implement the MoneySource and MoneySink interfaces (i.e., in TransferMoneySink, TransferMoneySource, etc.) Now we can invoke these setContext APIs from within the Context objects. This is the Context object's way of broadcasting, to all the objects involved in a given Use Case, the handles to the other methodless roles involved in the Use Case. Each object can cache away the role handle information it feels it needs for the Use Case. Here is what the TransferMoneyContext code might look like:

```

void
TransferMoneyContext::setContexts(void) {
    sourceAccount()->setContext(this);
    destinationAccount()->setContext(this);
    amount()->setContext(this);
}

TransferMoneyContext::TransferMoneyContext(void)
{
    lookupBindings();
    setContexts();
}

```

```
void
TransferMoneyContext::lookupBindings(void)
{
    sourceAccount_ = databaseLookup();
    destinationAccount_ = . . .
    amount_ = . . .
}
```

We use the simple helper methods that live in the protected interface of the methodful role class to act as local handles to the other roles in the Use Case scenario:

```
MoneySource* template<class ConcreteDerived>
class MoneySink::recipient(void) {
    return
        TransferMoneyContext_->destinationAccount();
}
```

If we also have a Pay Bills Use Case and a Context object for it, then the MoneySink role should also support access to the object playing the role of the Creditors:

```
std::list<Creditor*> creditors(void) const {
    return payBillsContext_->creditors();
}
```

For the Pay Bills scenario, we also need a PayBillsContext object analogous to the TransferMoneyContext trait:

```
void
PayBillsContext::setContexts(void)
{
    sourceAccount()->setContext(this);
}

PayBillsContext::PayBillsContext(void)
{
    lookupBindings();
    setContexts();
}
```

```

void
PayBillsContext::lookupBindings(void)
{
    sourceAccount_ = . . . . // like a database select
    creditors_ = . . . . // another database lookup
}

```

In this C++ implementation, the `setContext` member function is overloaded within each method-ful role. The Context type is the basis for the overloading: there is a separate `setContext` for every type of Use Case scenario in which the method-ful role participates. This of course has the strong liability of needing to add APIs to many method-ful roles every time a Use Case is added. The implementations of these methods in the derived classes cache away just those method-less role identifiers from the Context interface that it needs for the Use Case. Alternatively, it can cache a pointer to the Context object, which is guaranteed to persist for the duration of the Use Case.

Also in the trait, we add code to remember the Context when it identifies itself to all the objects involved in its collaboration. Here we show two `setContext` setters, one for each of two different Context objects. Because only one Context runs at a time, we can save a bit of space by sharing the two Context pointers in a single union:

```

public:
    // Context stuff
    void setContext(TransferMoneyContext *c) {
        TransferMoneyContext_ = c;
    }
    void setContext(PayBillsContext *c) {
        payBillsContext_ = c;
    }
public:
    TransferMoneySource(void): TransferMoneyContext_(0) {
    }

```

```

private:
    union {
        TransferMoneyContext *TransferMoneyContext_;
        PayBillsContext     *payBillsContext_;
    };

```

With the Context information safely cached inside the trait, the trait can now map any role name to the object currently playing that role in the existing collaboration. In C++ each trait provides a member function named after the role, and each such member function returns a pointer to the object playing that role in the existing Use Case. These functions are of protected access so they are accessible only to the role (trait) itself and potentially to the domain object which we anticipate might have knowledge of this role in unusual circumstances (though that wouldn't be recommended practice):

```

protected:
    MoneySink *recipient(void) {
        return TransferMoneyContext_->destinationAccount();
    }

```

Let's say that we have a `SavingsAccount` with `PayBillsContext` injected into it. We augment it only to inject the `TransferMoneySink` role into it. We do this by inheriting the parameterized class:

```

class SavingsAccount:
    public Account,
    public TransferMoneySink<SavingsAccount> {
public:
    typedef double Currency;
    SavingsAccount(void);
    virtual Currency availableBalance(void);
    virtual void decreaseBalance(Currency);
    virtual void increaseBalance(Currency);
    virtual void updateLog(string, MyTime, Currency);

```

```

private:
    // Model data
    Currency availableBalance_;
};


```

CheckingAccount is similar:

```

class CheckingAccount:
    public Account,
        public TransferMoneySink<CheckingAccount> {
public:
    CheckingAccount(void);
    virtual Currency availableBalance(void);
    virtual void decreaseBalance(Currency);
    virtual void increaseBalance(Currency);
    virtual void updateLog(string, MyTime, Currency);

private:
    // The data for the model
    Currency availableBalance_;
};


```

InvestmentAccount may again be similar, and so forth.

For Context objects used in this style, we can add guidelines to those started back on page 221:

6. Inside the constructor of each Context object, invoke the `setContext` method of each role that is involved in the Use Case scenario, passing in the Context object itself as an argument. This allows each object to cache references to its collaborators in the scenario.
7. Add an API that can be used to bind to the objects attached to each of the roles involved in the Context's Use Case scenario.
3. *Pass the Context object as an argument to all methods of method-ful roles.* Think of the Context as being a super object that contains,

in a very strong sense, all objects that will be involved in a given Use Case scenario. In the same sense that an object method passes the `self` or `this` identifier to the methods it calls, so we can envision each and every of the methods in a method-ful role receiving a reference to their shared Context. No programming language provides such an argument automatically as it does with `this` and `self`, so we can provide it explicitly.

This approach is clumsy to the extent that each method is cluttered with an additional argument. The Smalltalk implementation of DCI in BabyUML avoided this clumsiness by changing the Smalltalk compiler to look up contexts automatically, but here we try to avoid any solution that could not easily be ported from one installation to another. Though the approach is clumsy, it avoids the administrative boilerplate of the previous approach above.

4. (Recommended) *Let each method-ful role access a global Context object through macros whose syntax distinguishes such access as role access.* Roles interact with each other to realize a Use Case scenario. The source code can be written in terms of role references rather than object references to better match the end user mental model of the role interaction. Macros make it possible to use a syntax that hides the “active” nature of a role name invocation: that it actually resolves to an object pointer according to the role-to-object mapping in the Context object.

At any given time, the code is executing only within a single context. The code within one context may create another and, as described earlier, Context objects can stack. However, the fact that there is only one Context executing at a time makes it possible to maintain a single, “global” Context object pointer. Furthermore, the fact that the type of that context can be inferred by knowing what function is executing (since that Context is what started it off), we can safely use static casting to re-

store full Context type information to a generic pointer that stands in for all Context types. The pointer access can be buried inside of a macro that also does the necessary down-casting to the appropriate derived Context class. That derived class provides interfaces to retrieve the object pointers for the roles that they represent.

So let's again look at our banking example, which has the role Transfer Money Source for financial transfers. It participates in a Use Case together with another role which we'll call the RECIPIENT. Together with the code for the TransferMoneySource trait, we include the macro:

```
#define RECIPIENT \
    ((static_cast<TransferMoneyContext*> \
        (Context::currentContext_)->destinationAccount()))
```

If you want a higher degree of paranoia in the code, you can change the `static_cast` to a `dynamic_cast` and check for a null result.

Now we can define the `transferTo` method for the Transfer Money Source role in terms of the roles, like RECIPIENT, with which it interacts:

```
if (SELF->availableBalance() < amount) {
    endTransaction();
    throw InsufficientFunds();
} else {
    SELF->decreaseBalance(amount);
    RECIPIENT->increaseBalance(amount);
    SELF->updateLog(
        "Transfer Out",
        DateTime(),
        amount);
```

```

RECIPIENT->updateLog(
    "Transfer In",
    DateTime(),
    amount);
}

```

Note the use of the special role SELF, which designates the object for which the role is currently executing. Its macro is simply:

```
#define SELF \
    static_cast<const ConcreteDerived*>(this)
```

The ConcreteDerived parameter will be bound to the appropriate template argument in the template for the Transfer-MoneySource role.

Your innovativeness may discover other approaches that are even better for your own situation. Be Agile: inspect and adapt.

### *Nested Contexts in Method-ful Roles*

It's common practice to compose "re-usable" Use Case scenarios into higher-level scenarios. Ivar Jacobsson's original vision of Use Cases provided for an includes relation between a high level Use Case and a "smaller" Use Case on which it depended to complete interactions with the end user.

Let's look at our own little money funds software to develop an example. We already have a Use Case for transferring money (**Figure 30** on page 208). Let's say that we also want a Use Case for paying bills automatically from our account. Our new Use Case can use the money transfer use case to make the actual transfers, taking care of the logging and transactions and other "details" important to that scenario. We can see the new Use Case in **Figure 33**. Note the invocation of Move Money and Do Accounting—the fact that it is underlined is a cue that it is another Use Case.

**Use Case Name:** Pay Bills from Selected Account

**User Intent:** To use money from one of my accounts to pay off the current amount due from all of my creditors

**Motivation:** To be able to let the bank automatically pay my bills monthly, or for me to be able to pay all bills on demand with a minimum of effort

**Preconditions:** The Account Holder has identified the Source Account, Destination Account, and the amount to be transferred

**Basic Scenario:**

1. User selects source account
2. Move Money and Do Accounting

**Variations:**

**Postconditions:**

- ✓ All postconditions of Transfer Money hold

**Figure 33: Pay Bills Use Case**

Note that there are two ways of thinking about paying bills. One approach gives the responsibility of identifying creditors to the bank; another approach gives that responsibility to me. In the former approach, any creditor who can legitimize their claim against me is allowed by the bank to queue for payment. In the latter case, I both have to remember (with the help of my software) to whom I owe money and add to or subtract from such a list based on my discretion. These two modes of bill paying lead to two different designs because they distribute responsibility differently. It is the same responsibilities distributed differently across objects. What makes the difference? The difference comes from how the responsibilities aggregate together into different roles. These are two different designs—and two different Use Cases. Here, we choose to trust the

bank to screen my creditors, and I'm happy to pay who ever the bank believes is owed money by me.

The Pay Bills Use Case is germane to the role of a `MoneySource` and, in this case, belongs specifically to a `TransferMoneySource`. We can frame out the code (really only pseudo-code at this point) like this:

```
void payBills(void) {
    // Assume that we can round up the creditors
    for ( ; iter != creditors.end(); iter++ ) {
        try {
            // transfer the funds here
        } catch (InsufficientFunds) {
            throw;
        }
    }
}
```

If we wish, we later can come back and change the algorithm to sort the creditors by amount owed, by how long it has been since we last paid them, or whatever else we choose to do. For now, let's keep it simple.

We must solve two problems here. The first is to retrieve a stable set of creditors that will remain constant across the iterations of the `for` loop. Remember that DCI is designed to support the rapidly changing relationships between objects that one finds in a real-world system, and that we don't want new creditors coming into the picture after we've decided to pay the bills (no more than creditors want to potentially be dropped from the list of remunerated candidates should conditions change). We retrieve the list of creditors from the context and store it locally to solve that problem:

```
...
list<Creditor*> creditors = CREDITORS;
list<Creditor*>::iterator i = creditors.begin();
for ( ; i != creditors.end(); i++ ) {
    ...
}
```

The second problem is: How do we “invoke” the Move Money and Do Accounting Use Case scenario from within the new code for paying bills? It is usually the environment—a Controller or a domain object—that sets up the Context object and invokes its `doIt` operation. Here, the Use Case is kicked off not by a direct user interaction, but indirectly by another Use Case (which itself was probably started by an interaction on the GUI). That means that the enclosing Use Case for paying bills must set up the context for the funds transfer Use Case. It is as if Pay Bills reaches outside the program and presses the Transfer Money button on the screen.

As it stands, the `TransferMoneyContext` object makes that difficult to do because it is in charge of mapping the Use Case scenario roles to their objects, using context from the Controller or from hints left in the model objects. We fix this by adding a new constructor for the `TransferMoneyContext` object that allows the bill payment Use Case code to override the usual database select operation or other lookup used by the Context object to associate roles with objects:

```

TransferMoneyContext::TransferMoneyContext(
    Currency amount,
    MoneySource *source,
    MoneySink *destination)
{
    // Copy the rest of the stuff
    sourceAccount_ = source;
    destinationAccount_ = destination;
    amount_ = amount;
    setContexts(); // if using style 3 from p. 207
}

// We need this function only if using
// style 3 from page 207:

void
TransferMoneyContext::setContexts(void) {
    sourceAccount()->setContext(this);
    destinationAccount()->setContext(this);
}

```

The new, complete codified Use Case scenario looks like this:

```
template <class ConcreteDerived>
class TransferMoneySource: public MoneySource {
    . . .
    // Role behaviors
    void payBills(void) {
        // While object contexts are changing, we don't
        // want to have an open iterator on an external
        // object. Make a local copy.
        std::list<Creditor*> creditors = CREDITORS;
        std::list<Creditor*>::iterator iter =
            creditors.begin();
        for (; iter != creditors.end(); iter++) {
            try {
                // Note that here we invoke another Use Case
                TransferMoneyContext transferTheFunds(
                    (*iter)->amountOwed(),
                    SELF,
                    (*iter)->account());
                transferTheFunds.doit();
            } catch (InsufficientFunds) {
                throw;
            }
        }
    }
};
```

When the `TransferMoneyContext` object comes into being, it becomes the current context, replacing the global context pointer `Context::currentContext`. That allows the methods of the methodful roles inside the `Transfer Money Context` Use Case to resolve to the correct objects. Notice that the `Context` object lifetime is limited to the scope of the `try` block: there is no reason to keep it around longer. At the end of the scope its destructor is called, and it restores `Context::currentContext` to its previous value. Therefore, `Contexts` implement a simple stack that the `Context` class can represent as a linked list:

```

class Context {
public:
    Context(void) {
        parentContext_ = currentContext_;
        currentContext_ = this;
    }
    virtual ~Context() {
        currentContext_ = parentContext_;
    }
public:
    static Context *currentContext_;
private:
    Context *parentContext_;
};

```

### 8.3.6 Variants and Tricks on DCI

[Hiding domain class interfaces (because they should be called only by trusted logic, and such logic should exist only in roles?) Is this right? Can it be done in C++?]

[Mixing in roles only when necessary, perhaps at the point of object instantiation, instead of mixing in all roles into the domain class? Scala has anonymous classes: in C++ we have to create these classes. Is it worth it?]

[Context layering? Letting a Context act like a domain object—like SavingsAccoung?]

---

## 8.4 Updating the Domain Logic

In the atomic event architectural style, we have two coding tasks after the interfaces and abstract base classes are framed out: to write the functional business logic, and to flesh out the domain logic in the Model of MVC. The same is true here. Nevertheless, in the atomic event style these two kinds of code co-exist in the domain classes. Here, we have already separated out the functional business logic into the role classes. Let's step back and again compare the DCI ap-

proach to domain class evolution with what we already presented in CHAPTER 7.

#### 8.4.1 Comparing and Contrasting DCI with the Atomic Event Style

There are two basic ways to add what-the-system-does functionality to an object-oriented system: to add short, atomic operations directly to the domain objects, or to use the DCI architecture. DCI applies only when we have Use Cases that describe a sequence of tasks that are directed to some end-user goal. If the “Use Case” is a simple, atomic action, then Use Cases are the methodological equivalent of shooting a fly with an elephant gun.

Most object-oriented architectures over the years have been created as though they were atomic event architectures, which meant that they failed to separate the volatile what-the-system-does logic from the more stable domain model. We described some examples in Section 7.2. Those implementations usually correspond to highly visual or physical end-user interactions.

It has been a long time now that objects have established a beach-head in the traditionally algorithmic areas of business, finance, and numeric computation. All of these areas, perhaps former strongholds of FORTRAN, RPG, and COBOL, have traditionally featured algorithms. These algorithms have multiple steps, each one of which corresponds to some user intention. In an Agile world, which is usually interactive, these intentions are taken in by the system and come back to the user in discrete acknowledgments of completion. We want those sequences captured in the code—at least as well as we used to in FORTRAN, RPG, and COBOL. This notion of relating to increments of user intent is the “Rebecca Wirfs-Brock school” of Use Cases. [CITE]

Any given system will usually have a combination of these two architectural styles. We recommend the following approach as rules of thumb—but *only* as rules of thumb.

- If a system or subsystem has a critical mass of true scenarios and Use Cases that reflect sequences of conscious user intents, then design that entire subsystem using the DCI architecture. Make even the atomic operations part of the role interfaces and keep the domain interfaces clean. Sometimes this will mean duplicating the role interface in the domain object (because that's the right way to elicit the domain behavior), and it's likely that the method-ful role will only forward its request to the domain object. *Example: ....*
- If a system or subsystem is dominated by atomic operations, and has only a few true Use Cases, then use plain old object-oriented programming, where all of the system work is done by methods on the domain objects. This may slightly erode the cohesion of the domain objects (because the actions on them require a bit of help from other objects) but the overall architecture will be simpler and easier to understand. *Example: ....*
- If neither the Use Case semantics nor the atomic semantics dominate, use a hybrid. Atomic operations can go direct to the domain objects, while method-ful roles can be used to package up the Use Cases. *Example: ....*

The first point is that there is no need to choose one style over the other. Choosing a style is a matter of meeting programmer and designer expectations, to avoid surprises. There is no technical reason that the two cannot co-exist. The second point, which bears additional consideration, is that the above decisions should reflect organizational and business concerns. A large, complex organization sometimes works more smoothly if the code providing end-user functionality can more fully be separated from the code that captures domain logic.

## 8.4.2 The DCI approach: special considerations for domain logic

We have created the method-ful roles, carefully translated from the end-user Use Case scenarios. In an ideal world, we would do this translation without regard to the established APIs in the domain classes or, alternatively, we would take the domain class APIs as givens. Being more pragmatic, we realize that the domain classes and method-ful roles have insights to offer to each other. Just as in the atomic event architectural style, the interface between the method-ful roles and the domain classes evolves in three major ways: domain class method elaboration, factoring, and re-factoring.

In Section 7.3 we described how to update domain logic for the atomic event architecture case. Unlike that atomic event approach, the DCI approach leaves the domain classes untouched and pure. What does “pure” mean? It means *dumb*. The domain classes represent system data and the most basic functionality necessary to retrieve and modify it. These classes have no direct knowledge of user tasks or actions: they sustain the relatively staid state changes local to an individual logic without worrying too much about the coordinated object dynamics at the system level.

This has distinct advantages for the architect and long-term benefits to the end user. The Model of MVC-U stays cleanly a model, rather than a mixture of several user mental models. Instead of mixing the what-the-system-does logic in with the what-the-system-is logic as we did in the atomic event architectural style, here we keep it separate. Separation of validly separate concerns is always a good architectural practice.

When we start work on a new Use Case scenario, the logic in the method-ful roles usually depends on services from the domain objects. The classes for these objects may or may not exist yet, even though the APIs exist as coded interfaces in the architecture. Remember, the domain class API is simply an abstract base class, and there may or may not be a class behind it to support any given interface. The time has now come to create those classes.

Just as in the non-DCI case, filling in the domain member functions is straightforward. The DCI approach to actually writing domain class member functions is the same as for writing the general domain member functions to support the atomic event architectural style. See Section 7.3.

Member functions are one problem; bringing in the role traits is another. In C++, we use the Curiously Recurring Template Idiom [Cop1996] to mix in the role logic:

```

class SavingsAccount:
    public Account,
    public TransferMoneySink<SavingsAccount> {
public:
    Currency availableBalance(void);

    // These functions can be virtual if there are
    // specializations of AavingsAccount
    void decreaseBalance(Currency);
    void updateLog(string, MyTime, Currency);
    void increaseBalance(Currency);
};

class InvestmentAccount:
    public Account,
    public TransferMoneySource<InvestmentAccount> {
public:
    typedef double Currency;
    Currency availableBalance(void);
    void decreaseBalance(Currency);
    void updateLog(string, MyTime, Currency);
};

```

It's as simple as that.

---

## 8.5 *Context Objects in the User Mental Model: Solution to an Age-old problem*

In the old days we used to say that classes should reflect real-world concepts, but the question arises: How real is real? Consider a `SavingsAccount`, for example, as discussed throughout this chapter? Is it real? The fact is that if you go into a bank, you can't walk into the safe and find a bag of money on a shelf somewhere with your account number on it.

Your `SavingsAccount` isn't real, but it is part of your conceptual world model. That leaves us with a bit of a conundrum if we want to split the world into the domain part that reflects the actual form of the domain, and the behavior part that corresponds to your anticipated use of the domain model. What, exactly, *is* a `SavingsAccount`?

Before completely answering that question, it's useful to look at what the domain classes really are in a banking system. Banks are complex accounting entities with money flowing in dozens of different directions all the time, all with the goal of optimizing the bank's margins. Your `SavingsAccount` money isn't in a bag somewhere, but is more likely lent out to other clients so they can buy houses, or it's invested in pork bellies in Chicago or in gold in New York. It's complicated, and it's really hard to track down exactly where your money went and when it went there.

Difficult—but not impossible. If banks are anything, they are both accurate and precise regarding these transactions. Every transfer takes place as a transaction that's guaranteed not to lose any money along the way, even if a server or a network link goes down. These transactions are supported by database technology. As a further hedge against failure, there are transaction logs that can be used to recreate the sequences of events that led to your money being invested in those pork bellies. And there are audit trails over these transaction logs that have to reconcile with each other, further reducing the margin of error to the point of being almost negligible.

This has several implications. Your `SavingsAccount` balance at any given time isn't sitting in an object somewhere, nor is it even spinning on a disk as a value. It is dynamically calculated by span-

ning the audit trails, adding up any deposits (either that you made to the account, or that the bank made in terms of an interest accrual) and subtracting any withdrawals (including your own disbursements as well as any fees charged by the bank). As such, the SavingsAccount isn't a piece of "smart data," but is a collection of algorithms! It is a collection of related algorithms that implement a collection of related Use Cases that work in terms of roles such as SourceAccounts and DestinationAccounts and ATMs, such that those roles are bound to the transactions and audit trails related to *your* account.

We've seen this concept before. It's called a Context.

---

## 8.6 Why all these artifacts?

You're probably thinking: "We got by with just classes all of these years (and sometimes with a little thinking about objects even at design time). Why, all of a sudden, do we need all of this complexity?" With the more complete DCI model in hand, we can give a more complete argumentation than in our response to the same question back in Section 7.4.

The simple answer (if you like simple answers) is that all of this is essential complexity. Too much object-oriented design over the past twenty years is rooted in abstraction: the conscious discarding of information to help emphasize information of current interest. The problem with abstraction is that the decision about what to throw away is made at one point in time and its repercussions are often removed until later. In software development, it is usually made during analysis and design before the realities of implementation hit. That's too early to understand what should be kept around and what should be discarded. A better position is to ground the design in long-term domain invariants, and to keep those around. That's what CHAPTER 5 was about.

A related reason, more in line with program behavior in the same sense that CHAPTER 5 is about program form, is that this is the stuff

floating around in the head of the end user. To the degree the software captures the end user's world model, it's a good thing:

- Directly, it is about communicating with users in terms of their mental models of the world instead of computer-ese—along the lines of the Agile value of users and interactions over processes and tools.
- It makes it easier for us as programmers to communicate with users and to meet their needs—along the lines of the Agile value of customer collaboration.
- It makes it easier for us to reason about forms that are important to the user, such as the form of the sequencing of events toward a goal in a context—along the lines of the Agile value of working software.
- It catches change where it happens and encapsulates it, rather than spreading it across the architecture (as would be the case if we distributed parts of the algorithm across existing classes)—along the lines of the Agile value of embracing change.

No, we didn't just come along and add this list as an afterthought or as an opportunistic way to shoehorn these ideas into buzzword-dom. The techniques have come out of a conscious effort to strive towards the values that underlie Agile.

But let's get out of the clouds and get into the nitty-gritty.

### *Why not use classes instead of “method-ful roles”?*

Well, in most contemporary programming languages we actually do use classes as a way to implement method-ful roles. Scala properly has traits, which are in effect just method-ful roles. We foresee the need for a new programming language feature, which is a kind of generic collection of stateless algorithms that can be mixed into existing classes without having to resort to a “trick” like traits. Traits are a convenient and adequately expressive way of expressing these

semantics in the mean time. It's unlikely that such a language feature would make or break a project; it would only relieve temporary confusion or perhaps help new project members come on board a bit more quickly.

*Why not put the entire algorithm inside of the class with which it is most closely coupled?*

Even if one could identify this class, it wouldn't be the only one. The problem ultimately is another problem of essential complexity: the class structure is not the behavior structure, and there is a many-to-many mapping between these Use Cases and the classes of the objects that they orchestrate. The question then becomes one of how the hosting class effects the algorithm for objects of another class that want to play to the same Use Case scenario. That would mean duplicating the code manually, and that would be error-prone for all the reasons that we hate code duplication. Therefore, we "re-use" it, semi-automatically, using traits or another suitable method.

*Then why not localize the algorithm to a class and tie it to domain objects as needed?*

That's essentially what we do, and essentially what traits are.

*Why not put the algorithm into a procedure, and combine the procedural paradigm with the object paradigm in a single program?*

This is the old C++ approach and in fact is the approach advocated in *Multi-paradigm design for C++* [Cop1998]. The main problem with this approach is that it doesn't express relationships between related algorithms, or between related algorithm fragments, that exist in the end user's head. It obfuscates the role structure: these algorithms belong to roles, rather than existing as stand-alone procedures.

Does this mean that procedural design is completely dead? Of course not. There are still algorithms that are just algorithms (Freud's "sometimes a cigar is just a cigar"). We can encapsulate algorithms in the private implementation of classes using procedural decomposition. There's nothing wrong with that. However, those concerns are rarely architectural concerns; we leave those to the cleverness of designers and architects as they come to low-level design and the craftsmanship of implementation.

### *So, what do DCI and lean architecture give me?*

We claim that this style of development:

- Modularizes Use Case scenarios and algorithms in the software so you can reason about, unit test, and formally analyze the code with respect to functional requirements.
- Maps the architecture onto a domain model that reduces the time and expense of determining where new functionality should be added.
- Produces an architecture that can support a GUI or command set with few surprises for the user—because it is based on the end user mental model.
- Gives you all the flexibility of MVC in separating the data model of the program from the interface.
- Leads to a design that encourages the capture and expression of essential complexity, which means that when you need to express essential complexity, you don't need to go in with dynamite and jack hammers to make changes: the right structure is already there.
- Allows you to introduce functionality just in time, whether the code is in the traditional platform code or in the application code.

## *And remember...*

This is an Agile book, and you are (at least sometimes) an Agile programmer. We encourage you to think, to plan a bit, and to not just take our word for how to do things. We provide you examples to stimulate your thinking; these *concepts* in hand, you can tune the code to your own local needs. There are a million ways to map roles to objects, to inject methods into classes, and to make the context available to the method-ful roles. Here, we have focused on a C++ implementation: the idioms and approaches you use in Java, C# and Objective C will be different. Use your ingenuity. And read on for a few more hints.

---

## 8.7 *Beyond C++: DCI in Other Languages*

We've illustrated DCI first in C++ here for two reasons. First, the DCI implementation is statically typed and is based on templates; we wanted to show that DCI works even under these restrictions. The second reason is the relative breadth-of-use of C++ in real-time and complex problems that have the most need for such expressiveness.

However, DCI is hardly a C++ phenomenon. It is broadly based on the notion of injecting collections of methods into a class. C++ and most other classful languages tend to use classes as their building blocks for this injection, so it reduces to class composition—Schärli's original notion of traits. Other languages cannot only support DCI well but in many instances can do so better than C++ because their type systems are varyingly more dynamic.

### 8.7.1 Scala

Scala is one such language. In September of 2008, Bill Venners [Od+2008] demonstrated the first application of DCI in that language. Scala is interesting because it has traits as full first-class language citizens; it looks almost like it was made for DCI.

What's more, Scala has one significant advantage over C++. C++ forces us to do the role injection (class composition using templates) in the declaration of the domain class at compile time. That suggests that the addition of any role to a domain will require the recompilation of all code in that domain. (As is true in almost all languages, the domain code must actually be changed only if the new injection creates name collisions.) In Scala, the injection is done at the point of object instantiation:

```
    . . .
trait TransferMoneySource extends MoneySource {
  this: Account =>
  // This code is reviewable and testable!
  def transferTo(amount: Long, recipient: MoneySink) {
    beginTransaction()
    if (availableBalance < amount) {
      . . .
    }
  . . .
  val source =
    new SavingsAccount with TransferMoneySource
  val sink = new CheckingAccount with TransferMoneySink
  . . .
```

Bill Venners has contributed a Scala variant of our running Accounts example in APPENDIX A.

### 8.7.2 Python

Serge Beaumont and David Byers have contributed a rendition of the Accounts example in Python that dates from October 2008. The Python code is quite faithful to the DCI paradigm. It is a fascinating implementation that binds role methods to the domain objects in a fully dynamic way: the role methods are injected into the domain objects only for the duration of the Use Case scenario for which they are needed! Contrast this with the C++ implementation that pre-loads the role methods into the classes at compile time and the Scala implementation that loads the role methods into the object at instant-

tiation time. You can find the Python implementation in APPENDIX B.

### 8.7.3 C#

Christian Horsdal Gammelgaard has provided an early version of DCI in C#, and it is shown in APPENDIX C. The code features the way that C# handles traits using extension methods. Extension methods are methods that can statically be bound to a class at run time. They take explicit object pointer arguments rather than using a transparent `self` argument to give the illusion of being inside of the object. With that minor concession the illusion is good enough: collections of methods can be added to a class at run time to give the illusion of class composition.

### 8.7.4 ... and even Java

To date, the Java solutions are less satisfying than those offered by the older C++ language and the newer Python and Scala solutions. These modern languages seem to offer a good point of compromise—and perhaps a near-ideal solution—regarding expressiveness and performance. Ruby (Steen Lehmann; see APPENDIX D), has an analogous solution. The C++ solution shines in expressiveness and performance but suffers accidental coupling that has high configuration management costs. These Java solutions struggle to balance expressiveness and performance and are both more “foreign” to their host language than in the other examples presented here.

The Java example in APPENDIX E comes from Agata Przybyszewska. The TraitBase code in this Java implementation exposes the method twizzling that goes on in the Java reflection API to catch method calls on a domain object proxy and route them either to the domain object itself or to a method-ful role class that implements the what-the-system-does behavior. It incurs a string-to-method mapping on every method invocation and is reminiscent of the symbolic programming emulation of [Cop1992]. This example is a quickly constructed concept proof, and any adept Java programmer can

quickly identify low-hanging fruit optimizations such as adding a per-class hash map to associate method names with methods. However, a straightforward implementation that remains strictly within Java semantics (with the slight extension of a few annotations) won't succumb to most JIT strategies that might restore more familiar levels of method invocation overhead.

Another longstanding approach with many similarities to DCI can be found in the Qi4J framework [Qi42006] from Rickard Öberg and Michael Hunger. A Qi4J programmer decorates Java code with annotations that the framework can use to affect the right role-to-object bindings at run time. It is less an implementation in Java per se than it is an interim measure with less run-time overhead than the solution in APPENDIX E but with a bit more syntactic saccharin. There is nothing in Qi4J which at present can succinctly express either the dynamics of role-to-object bindings, or, as is a liability in most implementations other than Reenskaug's original Squeak implementation, anything that lets one express interactions in terms of roles without regard to an instance. One can find example code in APPENDIX F.

### 8.7.5 The Original Stars and Circles Example in Smalltalk

Here, we look at the original stars and circles example that Trygve Reenskaug wrote as the first application of his DCI-based IDE, BabyUML [CITE].

#### *Deriving the Method-ful Roles*

The stars example has two principle Use Cases. One of them controls the coming and going of planets and stars. The other controls the drawing of an arrow that grows to reach a target star or planet from a source star or planet, five times in succession.

```
Environment>>animateArrows
    [self currentState == #ARROWS]
    whileTrue:
        [self removeAllArrows.
        ThisContext reset.
        Shape1 play1.
        (Delay forMilliseconds: 1500) wait].
```

Each of the celestial bodies in the arrow animation sequence is also a role:

```
Shape1>>play1
    self displayLarge: '1'.
    Arrow12 play12.

Arrow12>>play12
    Environment addMorphBack: self.
    self from: Shape1 to: Shape2.
    self grow.

Shape2 play2.
    Shape2>>play2
    self displayLarge: '2'.
    Arrow23 play23.

Arrow23>>play23
    Environment addMorphBack: self.
    self from: Shape2 to: Shape3.
    self grow.
    Shape3 play3.

    . . .
```

and so forth. The program has a separate use case for the coming and going of stars and planets:

```
Shapes4Env>>animateShapes
| newShape suggestedShapeCenter |
[self currentState == #SHAPES]
    whileTrue: [data shapesCount >= 50
        ifTrue: [data deleteShape].
        data shapesCount <= 50
        ifTrue: [newShape :=
            (Collection randomForPicking next * 10)
            rounded odd
                ifTrue: [Shapes4Star new initialize]
                ifFalse: [Shapes4Circle new initialize].
            data addShape: newShape.
            [suggestedShapeCenter :=
                (self bounds left to: self bounds right)
                atRandom
                    @ (self bounds top to:
                        self bounds bottom) atRandom.
            data allShapes
                noneSatisfy:
                    [:someShape |
                        (someShape fullBounds extendBy:
                            someShape extent)
                        containsPoint:
                            suggestedShapeCenter]]]
        whileFalse.
        newShape center: suggestedShapeCenter.
        self addMorphBack: newShape.
        newShape flash]]
```

This code is a readable algorithm.

---

## 8.8 Documentation?

A good architecture reduces the need for additional explicit documentation. In DCI, the code is the documentation of the algorithm! Abstract base classes for the domain classes are themselves a

treasure of information about the partitioning of the system into domains and about the general facilities of each domain. A major theme of Agile communities is code-as-documentation, and it's a great common-sense approach that makes life easier for everyone who interacts with the code.

Programming language designers spend a lot of effort making languages expressive: in fact, such expressiveness is exactly what distinguishes programming languages. All common programming languages are equally powerful. The evidence for any programming language offering any significant productivity improvement is slim. Nevertheless, they express things differently. Smalltalk emphasizes minimalism; C++, richness of features and static typing; Java, a degree of cultural compatibility with C++ but with some Smalltalk semantics; and so forth. Given all this expressiveness, your coding language is a pretty good language for describing what the code does.

Given that, here are some good tips for supporting your documentation efforts—or, more precisely, for going beyond brute-force textual descriptions of the program design.

- *Choose the right language, and let the code do the talking.* If you're using Use Cases and DCI, what language best expresses your scenarios? If you're using an atomic event style, what language best expresses the semantics of the system events? Experiment with generally available languages to see which one best fits your need. Create your own domain-specific languages only with extreme caution; the cost in creating and sustaining the environment and infrastructure for a language is high, and it's difficult enough to express yourself in an existing language, let alone create the language in which you then must learn to express yourself.
- *Keep it short.* While the abstract base classes for the domain design should cover the scope of the business, don't load up the interfaces with gratuitous hooks and embellishments.
- *Choose good identifier names.* This may seem like a strange item to include in this list, but don't underestimate its value. Lov-

ingly name each identifier with the same care as for a first-born child. You'll be using these names a lot as you pair program or otherwise discuss the code with your colleagues. Many identifiers at the architectural level will rise to the level of the vernacular that you share with your customers. Strive to make them feel included by using their names for things, rather than yours.

- *Document architectural intent with pre- and post-conditions.* Just delimiting the range of values of a scalar argument speaks volumes about a piece of code. Such documentation evolves along with the code (unlike external text documents which are too often forgotten), can boost visibility of the alignment of the business structure with the code, and can give programmers constraints that focus and guide their work. This is Lean's notion of failure-proof (poka yoke) in the small.
- *Use block comments.* Let the code speak for itself on a line-by-line basis, but use explicit comments only as the introduction to a major opus of code (such as a class or package) or where the code is particularly tricky. If the block is longer than 3 lines, a lot fewer people will read it, and the chances of it being out-of-date will increase faster over time than if it were shorter.

In general, some of the best work you can do to support the architects in their goal to reduce discovery costs is to keep the code clean. You'll find your own practices and ceremonies for doing this, but take a cue from the experts in Uncle Bob's *Clean Code* book [Mar2009].

## 8.9 *History and Such*

### 8.9.1 DCI and Aspect-Oriented Programming

DCI arose as a conscious attempt to capture the end user's model of the algorithm in the code rather than as a solution to an engineering problem. Aspect-Oriented Programming, by contrast, can be viewed as an engineering solution to an architectural problem. Most implementations of AOP (most notably AspectJ [Ram2003]) make it almost impossible to reason about the functionality of aspectualized code without global knowledge of the program.

Jim Coplien first presented DCI as an alternative to AOP at the European Science Foundation Workshop on Correlation Theory in Vielsalm, Belgium in August 2008. DCI brings together logic that is "tangled" into domain classes the same way that particle states are entangled in quantum computing.

### 8.9.2 Context in the original DCI

In the original DCI implementation, Trygve Reenskaug made the Context object available to all callers by giving the illusion that each method received a handle to the currently active Context object as a parameter. The implementation is based on the fact that in Smalltalk, the stack itself is an object, and it can be traversed to find the controlling Context object on the activation record of the method that kicks off the Use Case in the environment. That method places the Context object on the stack before invoking a method on the top-level method-ful role. That invocation (as well as all subsequent invocations of methods on method-ful roles) translated into a stack search for the Context object, and an invocation of a database lookup on that Context object to retrieve the object playing the method-ful role in question. That object, by design, had already been injected with the methods of the method-ful role, which then executed with `self` bound to the selected object.

### **8.9.3 Other approaches**

Capabilities in the IBM System 38?

DRAFT



# *Scala Implementation of the DCI Account Example*

```
import java.util.Date

trait Account {
    private var balance: Long = 0
    def availableBalance: Long = balance
    def decreaseBalance(amount: Long) {
        if (amount < 0)
            throw new InsufficientFundsException
        balance -= amount
    }
    def increaseBalance(amount: Long) {
        balance += amount
    }
    def updateLog(msg: String, date: Date,
                 amount: Long) {
        println("Account: " + toString + ", " + msg + ", "
               + date.toString + ", " + amount)
    }
}

trait MoneySource {
    def transferTo(amount: Long, recipient: MoneySink)
}
```

```
trait MoneySink {
    def increaseBalance(amount: Long)
    def updateLog(msg: String, date: Date, amount: Long)
}

trait TransferMoneySink extends MoneySink {
    this: Account =>
    def transferFrom(amount: Long, src: MoneySource) {
        increaseBalance(amount)
        updateLog("Transfer in", new Date, amount)
    }
}

class InsufficientFundsException
    extends RuntimeException

trait TransferMoneySource extends MoneySource {

    this: Account =>
    // This code is reviewable and testable!

    def transferTo(amount: Long, recipient: MoneySink) {

        beginTransaction()

        if (availableBalance < amount) {

            endTransaction()
            throw new InsufficientFundsException
        }
        else {
            decreaseBalance(amount)
            recipient.increaseBalance(amount)
            updateLog("Transfer Out", new Date, amount)
            recipient.updateLog("Transfer In",
                new Date, amount)
        }
        gui.displayScreen(SUCCESS_DEPOSIT_SCREEN)
        endTransaction()
    }
}
```

```
class SavingsAccount extends Account {  
    override def toString = "Savings"  
}  
  
class CheckingAccount extends Account {  
    override def toString = "Checking"  
}  
  
object App extends Application {  
    val source =  
        new SavingsAccount with TransferMoneySource  
    val sink = new CheckingAccount with TransferMoneySink  
    source.increaseBalance(100000)  
    source.transferTo(200, sink)  
    println(source.availableBalance + ", " +  
           sink.availableBalance)  
}
```



# *Account example in Python*

```
"""
DCI proof of concept
```

This is David's version with changes by Serge:

- moved context out of the domain object onto the stack. Put in as contrast and for exploration, not necessarily because it's better on all fronts.
- To show the use of "context on the stack" I removed the sink parameter. In this Collaboration it's okay because the source and sink are a given.
- Context is a separate object to the Collaboration (again for exploration of alternatives). Made a class for it, but a Dictionary is also possible. Any advantages to using lookup keys?
- added Role method in MoneySink to ensure only access though Role interface
- changed amount to be a parameter instead of a constructor parameter (exploration).
- Added if \_\_name\_\_... idiom.

Author: David Byers, Serge Beaumont  
7 October 2008

```
"""

import new
```

```
class Role(object):
    """A Role is a special class that never gets
    instantiated directly. Instead, when the user wants
    to create a new role instance, we create a new class
    that has the role and another object's class
    as its superclasses, then create an instance of that
    class, and link the new object's dict to the original
    object's dict."""

    def __new__(cls, ob):
        members = dict(__ob__ = ob)
        if hasattr(ob.__class__, '__slots__'):
            members['__setattr__'] = Role.__setattr__
            members['__getattr__'] = Role.__getattr__
            members['__delattr__'] = Role.__delattr__

        c = new.classobj("%s as %s.%s" %
                         (ob.__class__.__name__,
                          cls.__module__, cls.__name__),
                         (cls, ob.__class__), members)
        i = object.__new__(c)
        if hasattr(ob, '__dict__'):
            i.__dict__ = ob.__dict__

        return i

    def __init__(self, ob):
        """Do not call the superclass __init__. If we
        did, then we would call the __init__ function in
        the real class hierarchy too (i.e. Account, in
        this example)"""
        pass

    def __getattr__(self, attr):
        """Proxy to object"""
        return getattr(self.__ob__, attr)

    def __setattr__(self, attr, val):
        """Proxy to object"""
        setattr(self.__ob__, attr, val)
```

```
def __delattr__(self, attr):
    """Proxy to object"""
    delattr(self.__ob__, attr)

class MoneySource(Role):
    def transfer_to(self, ctx, amount):
        if self.balance >= amount:
            self.decreaseBalance(amount)
            ctx.sink.receive(ctx, amount)

class MoneySink(Role):
    def receive(self, ctx, amount):
        self.increaseBalance(amount)

class Account(object):
    def __init__(self, amount):
        print "Creating a new account with balance of " +
              str(amount)
        self.balance = amount
        super(Account, self).__init__()

    def decreaseBalance(self, amount):
        print "Withdraw " + str(amount) + " from " +
              str(self)
        self.balance -= amount

    def increaseBalance(self, amount):
        print "Deposit " + str(amount) + " in " +
              str(self)
        self.balance += amount

class Context(object):
    """Holds Context state."""
    pass

class TransferMoney(object):
    def __init__(self, source, sink):
        self.context = Context()
        self.context.source = MoneySource(source)
        self.context.sink = MoneySink(sink)
```

```
def __call__(self, amount):
    self.context.source.transfer_to(
        self.context, amount)

if __name__ == '__main__':
    src = Account(1000)
    dst = Account(0)

    t = TransferMoney(src, dst)
    t(100)

    print src.balance
    print dst.balance
```

## *Account example in C#*

Christian Horsdal Gammelgaard provides the following code in C#, using the extension method facility as a way to demonstrate injection of role functionality into a domain class.

```
using System;

namespace DCI
{
    // Methodless roles
    public interface TransferMoneySink
    {
    }

    // Methodfull roles
    public interface TransferMoneySource
    {
    }
}
```

```
public static class TransferMoneySourceTraits
{
    public static void TransferFrom(
        this TransferMoneySource self,
        TransferMoneySink recipient, double amount)
    {
        // Is this cheating?
        // -It means that this methodful role can only
        // be mixed into Account object (and subtypes)
        // But in the other implementations the object
        // having a methodful role mixed in also has to
        // have expected (duck) type, dont they?
        Account self_ = self as Account;
        Account recipient_ = recipient as Account;

        // Self-contained readable and testable
        // algorithm

        if (self_ != null && recipient_ != null)
        {
            self_.DecreaseBalance(amount);
            self_.Log("Withdrawing " + amount);
            recipient_.IncreaseBalance(amount);
            recipient_.Log("Depositing " + amount);
        }
    }
}
```

```
// Context object
public class TransferMoneyContext
{
    // Properties for accessing the concrete objects
    // relevant in this context through their
    // methodless roles
    public TransferMoneySource Source {
        get; private set;
    }

    public TransferMoneySink Sink {
        get;
        private set;
    }

    public double Amount {
        get; private set;
    }

    public TransferMoneyContext()
    {
        // logic for retrieving source and sink accounts
    }

    public TransferMoneyContext(
        TransferMoneySource source,
        TransferMoneySink sink,
        double amount)
    {
        Source = source;
        Sink = sink;
        Amount = amount;
    }

    public void Doit()
    {
        Source.TransferFrom(Sink, Amount);

        // Alternatively the context could be passed
        // to the source and sink object.
    }
}
```

```
/////////// Model //////////  
  
// Abstract domain object  
public abstract class Account  
{  
    public abstract void DecreaseBalance(double  
amount);  
    public abstract void IncreaseBalance(double  
amount);  
    public abstract void Log(string message);  
}
```

```
// Concrete domain object
public class SavingsAccount :
    Account,
    TransferMoneySource,
    TransferMoneySink
{
    private double balance;

    public SavingsAccount()
    {
        balance = 10000;
    }

    public override void DecreaseBalance(double amount)
    {
        balance -= amount;
    }

    public override void IncreaseBalance(double amount)
    {
        balance += amount;
    }

    public override void Log(string message)
    {
        Console.WriteLine(message);
    }

    public override string ToString()
    {
        return "Balance " + balance;
    }
}
```

```
////////// Controller //////////  
  
// Test controller  
  
public class App  
{  
    public static void Main(string[] args)  
    {  
        SavingsAccount src = new SavingsAccount();  
        SavingsAccount snk = new SavingsAccount();  
  
        Console.WriteLine("Before:");  
        Console.WriteLine("Src:" + src);  
        Console.WriteLine("Snk:" + snk);  
  
        Console.WriteLine("Run transfer:");  
  
        new TransferMoneyContext(src, snk, 1000).Doit();  
  
        Console.WriteLine("After:");  
        Console.WriteLine("Src:" + src);  
        Console.WriteLine("Snk:" + snk);  
  
        Console.ReadLine();  
    }  
}
```

## *Account example in Ruby*

This rendition comes from Steen Lenmann.

```
class Account
  attr_reader :balance

  def initialize
    @balance = 0
  end

  def decreaseBalance(amount)
    raise "Insufficient funds" if amount < 0
    @balance -= amount
  end

  def increaseBalance(amount)
    @balance += amount
  end

  def update_log(msg, date, amount)
    puts "Account: #{inspect}, #{msg}, #{date.to_s},
#{amount}"
  end

  def self.find(account_id)
    @@store ||= Hash.new
    return @@store[account_id] if @@store.has_key? account_id
  end
```

```
if :savings == account_id
  account = SavingsAccount.new
  account.increaseBalance(100000)
elsif :checking == account_id
  account = CheckingAccount.new
end
@@store[account_id] = account
account
end
end

module TransferMoneySource
def transfer_to(amount, recipient)
  raise "Insufficient funds" if balance < amount
  decreaseBalance amount
  recipient.increaseBalance amount
  update_log "Transfer Out", Time.now, amount
  recipient.update_log "Transfer In", Time.now, amount
end
end

class TransferMoneyContext
attr_reader :source_account, :destination_account,
:amount

def initialize(amount, source_account_id,
  destination_account_id)
  @source_account = Account.find(source_account_id)
  @source_account.extend TransferMoneySource

  @destination_account = Account.find(
    destination_account_id)
  @amount = amount
end

def doit
  @source_account.transfer_to
  @amount, @destination_account
end
end
```

```
class SavingsAccount < Account
end

class CheckingAccount < Account
end

context = TransferMoneyContext.new(200, :savings,
                                  :checking)
context.doit
context = TransferMoneyContext.new(100, :checking,
                                  :savings)
context.doit

puts "#{context.source_account.balance},
      #{context.destination_account.balance}"
```





```
// TraitBaseInterface.java

package dk.mathmagicians.traithelp;

interface TraitBaseInterface<T> {
    public void setSelf(T self);
}

// Trait.java

package dk.mathmagicians.traithelp;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

/** keeps the annotations in code at runtime */
@Retention(RetentionPolicy.RUNTIME)
public @interface Trait {

}
```

```
// TraitBase.java

package dk.mathmagicians.traithelp;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;
import java.util.ArrayList;
import java.util.List;

/** Base class that all traits must implement. */
public abstract class TraitBase<T>
    implements TraitBaseInterface<T> {

    protected T self;

    public void setSelf(T self) {
        this.self = self;
    }

    /**
     * This is where the magic happens ... basically we are
     * creating a dynamic proxy, and making sure, that the
     * correct method gets invoked. Note - this
     * example can only add one trait at a time.
     *
     * @param <S>
     *          the type of the traittype
     * @param <S>
     * @param theObjectToTraitify
     * @param traitType
     * @return
     * @throws IllegalArgumentException
     *          the trait type is required to be marked by the
     *          marker interface {@code Trait}
     */
    @SuppressWarnings("unchecked")
    public static <T, S extends TraitBase<? super T>, U> U addTrait(
        T theObjectToTraitify, Class<? extends S> traitType,
        Class<? extends U> traitInterfaceType)
        throws IllegalArgumentException {
        // Check whether the traitType has the Trait annotation
        if (theObjectToTraitify == null || traitType == null)
```

```
    || traitInterfaceType == null) {
        throw new IllegalArgumentException(
            "Object and trait cant be null");
    }
    if (traitType == null ||
        traitType.getAnnotation(Trait.class) == null) {
        throw new IllegalArgumentException(
            " Did you remember to use the annotation Trait?");
    }

    try {
        S trait = traitType.newInstance();
        // Create a proxy object
        Class<?>[] types = { traitInterfaceType };
        // OK to suppress warnings - proxy is guaranteed to
        // implement all of its interface types
        return (U) Proxy.newProxyInstance(
            theObjectToTraitify.getClass()
                .getClassLoader(),
            types,
            new TraitifyingInvocationHandler(
                theObjectToTraitify, trait
            )
        );
    } catch (InstantiationException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
        return null;
    } catch (IllegalAccessException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
        return null;
    }
}

/**
 * Quick & dirty implementation - to show the idea.
 * Care need to be taken (but is not) with doubled method
 * names, and varargs.
 *
 * @author Agata Przybyszewska
 *
 * @param <S>
 */

```

```

private static class TraitifyingInvocationHandler<T,
                                         S extends TraitBase<T>>
    implements InvocationHandler {

    private List<Method> traitMethods =
        new ArrayList<Method>();
    private S theTrait;
    private T theObjectToTraitify;

    @SuppressWarnings("unchecked")
    public TraitifyingInvocationHandler(T theObjectToTraitify,
                                         S theTrait) {
        this.theObjectToTraitify = theObjectToTraitify;
        this.theTrait = theTrait;
        for (Method method: theTrait.getClass().getMethods()) {
            if (method.getAnnotation(Trait.class) != null) {
                traitMethods.add(method);
            }
        }
        // set self on the trait
        theTrait.setSelf(theObjectToTraitify);
    }

    @Override
    public Object invoke(Object proxy, Method method,
                         Object[] args)
        throws Throwable {
        if (method.getAnnotation(Trait.class) != null) {
            // we need to call the method from the
            // traitType class
            for (Method traitMethod : traitMethods) {
                boolean matching = traitMethod.getName().equals(
                    method.getName());
                for (int i = 0;
                     i < traitMethod.getParameterTypes().length
                     && matching;
                     i++) {
                    matching &=
                        traitMethod.getParameterTypes()[i]
                            .equals(method.getParameterTypes()[i]);
                }
            }
        }
    }
}

```

```
// invoke the method on the trait!
if (matching) {
    return traitMethod.invoke(theTrait,
        args);
}
return null;
} else {
    // invoke the regular method on
    // the proxy object
    return method.invoke(theObjectToTratify,
        args);
}
}

}

// Account.java
package dk.mathmagicians.money;

public class Account {
    private long balance =0;

    public void increaseBalance(long amount){
        balance += amount;
    }

    public void decreaseBalance(long amount){
        balance -= amount;
    }

    public long getBalance() {
        return balance;
    }
}
```

```
    @Override
    public String toString() {
        return getClass().getName()+"=[balance ="+ balance+"]";
    }
}

// TransferMoneySource.java

package dk.mathmagicians.money;

import dk.mathmagicians.traithelp.Trait;

public interface TransferMoneySource<T extends Account> {
    @Trait
    public abstract void transfer(long amount, MoneySink to);
    @Trait
    public abstract void increaseBalance(long amount);
}

// ClientWithTraits.java

package dk.mathmagicians.money;

import java.lang.reflect.Proxy;
import dk.mathmagicians.traithelp.TraitBase;
```

```
public class ClientWithTraits {

    public static void main(String[] args) {
        System.out.println("* Welcome to the ClientWithTraits *");
        Account mySavingsAccount = new Account();
        mySavingsAccount.increaseBalance(10000);
        Account myInvestmentAccount = new Account();

        TransferMoneySource savingsAccountWithTrait =
            TraitBase.addTrait(mySavingsAccount,
                TransferMoneySourceTrait.class,
                TransferMoneySource.class);
        TransferMoneySink investmentAccountWithTrait =
            TraitBase.addTrait(myInvestmentAccount,
                TransferMoneySinkTrait.class,
                TransferMoneySink.class);

        savingsAccountWithTrait.transfer(100,
            investmentAccountWithTrait);
        System.out.println("Savings account is: "
            +savingsAccountWithTrait +
            ", investment account is: " +
            investmentAccountWithTrait);
    }
}

// TransferMoneySinkTrait.java

package dk.mathmagicians.money;

import dk.mathmagicians.traithelp.Trait;
import dk.mathmagicians.traithelp.TraitBase;
```

```
@Trait
public class TransferMoneySinkTreat extends TraitBase<Account>
implements TransferMoneySink {

    @Override
    @Trait
    public void increaseBalance(long amount) {
        self.increaseBalance(amount);
    }

}

// MoneySink.java

package dk.mathmagicians.money;

import dk.mathmagicians.traithelp.Trait;

@Trait
public interface MoneySink {
    @Trait
    public void increaseBalance(long amount);
}

// TransferMoneySource.java

package dk.mathmagicians.money;

import dk.mathmagicians.traithelp.Trait;
import dk.mathmagicians.traithelp.TraitBase;
```

```
@Trait
public class TransferMoneySourceTrait<T extends Account>
    extends TraitBase<T>
    implements TransferMoneySource<T> {

    /* (non-Javadoc)
     * @see dk.mathmagicians.money.TransferMoneySource#transfer(
     *      long, dk.mathmagicians.money.MoneySink)
     */
    @Trait
    public void transfer( long amount, MoneySink to ){
        self.decreaseBalance(amount);
        to.increaseBalance(amount);
        System.out.println("Money transfer completed:"+self);
    }

    /* (non-Javadoc)
     * @see
     *
dk.mathmagicians.money.TransferMoneySource#increaseBalance(long)
    */
    @Override
    @Trait
    public void increaseBalance(long amount) {
        self.increaseBalance(amount);
    }
}

// TransferMoneySink.java

package dk.mathmagicians.money;

import dk.mathmagicians.traithelp.Trait;
import dk.mathmagicians.traithelp.TraitBase;

@Trait
public interface TransferMoneySink<T extends Account>
    extends MoneySink {
```

```
// MoneySource.java

package dk.mathmagicians.money;

import dk.mathmagicians.traithelp.Trait;

@Trait
public interface MoneySource {

}
```

Qi4J [Qi42006] is a Java framework that supports class composition to achieve a DCI-like architecture. The implementation relies heavily on annotations. You can read more about the annotations and the framework in the reference.

```
@Concerns({PurchaseLimitConcern.class, InventoryConcern.class})
@Mixins( PropertiesMixin.class )
public interface OrderComposite
    extends Order, HasLineItems, Composite
{
}

public abstract class InventoryConcern
    implements Invoice
{
    @Service InventoryService inventory;
    @ConcernFor Invoice next;

    public void addLineItem( LineItem item )
    {
        String productCode = item.getProductCode();
        int quantity = item.getQuantity();
        inventory.remove( productCode, quantity );
        next.addLineItem( item );
    }
}
```

```
public void removeLineItem( LineItem item )
{
    String productCode = item.getProductCode();
    int quantity = item.getQuantity();
    inventory.add( productCode, quantity );
    next.removeLineItem( item );
}
}

@Concerns({PurchaseLimitConcern.class, InventoryConcern.class})
@Mixins( PropertiesMixin.class )
public interface OrderComposite
    extends Order, HasLineItems, EntityComposite
{
}

@SideEffects( MailNotifySideEffect.class )
@Concerns({PurchaseLimitConcern.class, InventoryConcern.class})
@Mixins( PropertiesMixin.class )
public interface OrderComposite
    extends Order, HasLineItem, EntityComposite
{
}

public abstract class MailNotifySideEffect
    implements Order
{
    @Service MailService mailer;
    @ThisCompositeAs HasLineItems hasItems;
    @ThisCompositeAs Order order;
```

```
public void confirmOrder()
{
    List<LineItem> items = hasItems.getLineItems();

    StringBuilder builder = new StringBuilder();
    builder.append( "An Order has been made.\n");
    builder.append( "\n\n");
    builder.append( "Total amount: " );
    builder.append( order.getOrderAmount() );
    builder.append( "\n\nItems ordered:\n" );
    for( LineItem item : items )
    {
        builder.append( item.getName() );
        builder.append( " : " );
        builder.append( item.getQuantity() );
        builder.append( "\n" );
    }
    mailer.send( "sales@mycompany.com",
                builder.toString() );
}
```



---

## *DCI, Aspects, and Multi-Methods*

DRAFT



---

# Bibliography

- [Autopo2009] “Autopoiesis.” Wikipedia.  
<http://en.wikipedia.org/wiki/Autopoiesis>. 6 March 2009, accessed 29 March 2009.
- [Bal2000] Ballard, Glenn. Positive vs. Negative Iteration in Design. From URL  
<http://www.leanconstruction.org/pdf/05.pdf>, accessed 18 July 2008.
- [Bec1991] Beck, Kent. Think like an object. In UNIX Review, September 1991, ff. 41.
- [Bec1999] Beck, Kent. Extreme Programming Explained: Embrace Change. Reading, MA: Addison-Wesley, 1999.
- [Bec+2001] Beck, Kent, et al. The Agile Manifesto.  
<http://www.agilemanifesto.org>, February 2001, accessed 15 November 2008.
- [Bec2002] Beck, Kent. Test-driven development by example. Addison-Wesley, 2002.
- [Bec2005] Beck, Kent. Extreme Programming Explained, 2<sup>nd</sup> edition. Pearson Publications, 2005.
- [BeyHol1998] Beyer, Hugh, and Karen Holtzblatt. Contextual Design. San Francisco: Morgan Kauffman, 1998.

- [Bjø2003] Bjørnvig, Gertrud. Patterns for the role of Use Cases. Proceedings of EuroPLoP '03, p. 890.
- [Bra1995] Brandt, Stewart. How buildings learn: what happens to them after they're built. New York: Penguin, 1995.
- [Bra1999] Brand, Stewart. The clock of the long now. Basic Books, 1999.
- [Bu+1996] [BuHeSc2007a] Pattern-oriented software architecture volume 41 a system of patterns. Wiley, 1996.
- [BuHeSc2007a] Pattern-oriented software architecture volume 4: a pattern language for distributed computing. Wiley, 2007.
- [BuHeSc2007b] Pattern-oriented software architecture volume 4: on patterns and pattern languages. Wiley, 2007.
- [Bye2008a] Byers, David. Personal E-mail correspondence, 7 October 2008.
- [Coc2007] Cockburn, Alistair. Agile Software Development: The Cooperative Game, 2nd ed. Reading, MA: Addison-Wesley 2007.
- [Con1986] Conway, Melvin E. How do committees invent? *Dta-mation* 14(4), April, 1968.
- [Cop1992] Coplien, James O. Advanced C++ Programming Styles and Idioms. Reading MA: Addison-Wesley, 1992.
- [Cop1996] James O. Coplien. A Curiously Recurring Template Pattern. In Stanley B. Lippman, editor, C++ Gems, 135—144. Cambridge University Press, New York, New York, 1996.
- [Cop1998] Coplien, James O. Multi-paradigm design for C++. Reading, MA: Addison-Wesley, 1998.

- [CopHar2004] Coplien, James, and Neil Harrison. *Organizational Patterns of Agile Software Development*. Upper Saddle River, NJ: Prentice-Hall / Pearson, July 2004.
- [CoSu2009] Coplien, James and Jeff Sutherland. Seven +/- 3 Software Scrum Sensibilities. Scrum Gathering, Orlando, Florida, 16 March 2009. [TODO: URL]
- [Cro1984] Cross, Nigel, ed. *Developments in Design Methodology*. Chichester, UK: Wiley, 1984.
- [Dav1999] Davidson, E. J. Joint application design (JAD) in practice. *Journal of Systems & Software*, 45(3), 1999, 215–223.
- [Dic2007] —. *New Oxford American Dictionary*, 2007.
- [EiCz2000] Eisenecker, Ulrich, and Krysztof Czarnecki. *Generative Programming: Methods, Techniques and Applications*. Addison-Wesley, 2000.
- [Eva2003] Evans, Eric. *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley: 2003.
- [Fuh2008] Fuhrer, Phillip. Personal E-mail conversation of 17 October 2008.
- [Fra+2003] Fraser, Steven, Kent Beck, Bill Caputo, Tim Mackinnon, James Newkirk and Charlie Pool. “Test Driven Development (TDD).” In M. Marchesi and G. Succi, eds., *XP 2003*, LNCS 2675, pp. 459—462, 2003. © Springer-Verlag, Berlin and Heidelberg, 2003.
- [Ga+2005] Gamma, Eric, et al. *Design Patterns: Elements of Reusable object-oriented software*. Reading, Ma: Addison-Wesley, ©2005.

- [Gab1998] Gabriel, R. Patterns of Software: Tales from the Software Community. New York: Oxford University Press, 1998.
- [Gla2006] Glass, Robert L. The Standish Report: Does It Really Describe a Software Crisis? CACM 49(8), August 2006, pp. 15—16.
- [Ha2007] Hanmer, Robert S. Patterns for fault tolerant software. Wiley, 2007.
- [Jan1971] Janis, Irving L. Groupthink. Psychology Today, November 1971, 43—46, 74—76.
- [JanSal2008] Janzen and Saledian, Does test-driven development really improve software design quality? IEEE Software 25(2), March / April 2008, pp. 77—84.
- [Kay1972] Kay, Alan. A Personal Computer for Children of All Ages. Xerox Palo Alto Research Center, 1972.
- [Ker2001] Kerth, Norman L. Project Retrospectives: A Handbook for Team Reviews. Dorset House Publishing Company, 2001.
- [KiJa2004] Kircher, Michael, and Prashant Jain. Pattern-oriented software architecture volume 3: patterns for resource management. Wiley, 2004.
- [Lau1993] Laurel, Brenda. Computers as Theatre. Reading, MA: Addison-Wesley, 1993.
- [Lie1996] Lieberherr, Karl J. Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns. PWS Publishing Company, Boston, 1996. ISBN 0-534-94602-X, available at <http://www.ccs.neu.edu/research/demeter>.

- [Lik2004] Liker, Jeffrey K. *The Toyota Way*. McGraw-Hill, 2004.
- [Lis1986] Liskov, Barbara. Data abstraction and hierarchy. *SIGPLAN Notices* 23(5), May 1986.
- [MaBiNo2004] Martin, Angela, Robert Biddle and James Noble. The XP Customer Role in Practice: Three case studies. *Proceedings of the Second Annual Agile Development Conference*, 2004.
- [Mar2004] Martin, Angela. Exploring the XP Customer Role, Part II. *Proceedings of the 5<sup>th</sup> annual conference on Extreme Programming and Agile Processes in Software Engineering*, Jutta Eckstein and Hubert Baumeister, eds.
- [Mar2009] Martin, Robert C., et al. *Clean Code: A Handbook of Agile Software Craftsmanship*. Reading, MA: Pearson, ©2009.
- [Mey1994] Meyer, Bertrand. *Object-oriented software construction* (second edition). Prentice-Hall, 1994.
- [Moo2001] Moore, Thomas. *Original Self*. Perennial, 2001.
- [NaRa1968] Naur. Peter., and B. Randell, eds. *Proceedings of the NATO Conference on Software Engineering*. Nato Science Committee, October 1968.
- [Nei1980] Neighbors, J. M. "Software Construction Using Components." Tech Report 160. Department of Information and Computer Sciences, University of California. Irvine, CA. 1980.
- [Nei1989] Neighbors, J. M. "Draco A Method for Engineering Reusable Software Systems." In Biggerstaff, T. J. and A. J. Perlis, eds., *Software Reusability*, Vol. 1: Concepts and Models. ACM Frontier Series. Reading, AM: Addison-Wesley, 1989, Ch. 12, pp. 295—319.

- [NoWe2000] Noble, James, and Charles Weir. Small memory software: patterns for systems with limited memory. Addison-Wesley, 2000.
- [Od+2008] Odersky, Martin, Lex Spoon and Bill Venners. Programming in Scala: A comprehensive step-by-step guide. Artima, 2008,  
[http://www.artima.com/shop/programming\\_in\\_scala](http://www.artima.com/shop/programming_in_scala), accessed 4 October 2008.
- [Øst2008] Østergaard, Jens. Personal E-mail exchange, 8 October, 2008.
- [Par1978] Parnas, David. Designing Software for Ease of Extension and Contraction. Proceedings of the 3<sup>rd</sup> International Conference on Software Engineering. Atlanta, GA., May 1978, pp. 264—277.
- [Qi42006] —. Qi4J in 10 Minutes. <http://www.qi4j.org/163.html>, accessed 2 October 2008.
- [Ram2003] Laddad, Ramnivas. AspectJ in Action. Manning Publications, 2003.
- [Ras2000] Raskin, Jeff. The humane interface: New directions for designing interactive systems. Reading, MA: Addison-Wesley, 2000.
- [Ree1996] Reenskaug, Trygve, P. Wold and O. A. Lehne. Working with Objects: The OOram Software Engineering Method. Greenwich: Manning Publications, 1995.
- [Ree2003] Reenskaug, Trygve. The Model-View-Controller (MVC): Its Past and Present. From  
[http://heim.ifi.uio.no/trygver/2003/javazone-jao/MVC\\_pattern.pdf](http://heim.ifi.uio.no/trygver/2003/javazone-jao/MVC_pattern.pdf), accessed 10 October 2008.  
August 2003.

- [Rie1995] Dirk Riehle, Heinz Züllighoven. "A Pattern Language for Tool Construction, Integration Based on the Tools, Materials Metaphor." In *Pattern Languages of Program Design*. Edited by James O. Coplien, Douglas C. Schmidt. Addison-Wesley, 1995. Chapter 2, pages 9—42.
- [Ryb1989] Rybczynski, Witold. *The Most Beautiful House in the World*. New York: Penguin, 1989.
- [Sch2003] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz and Andrew Black, "Traits: Composable Units of Behavior," *Proceedings of European Conference on Object-Oriented Programming (ECOOP'03)*, LNCS, vol. 2743, Springer Verlag, July 2003, pp. 248—274.
- [ScStRo2000] Pattern-oriented software architecture volume 2: patterns for concurrent and distributed objects. Wiley, 2000.
- [SinAbr2007a] Siniaalto, Maria, and Pekka Abrahamsson. Comparative study on the effect of test-driven development on program design and test coverage. ESEM 2007.
- [SinAbr2007b] Siniaalto, Maria, and Pekka Abrahamsson. Does test-driven development improve the program code? Alarming results from a comparative case study. *Proceedings of Cee-Set 2007*, 10 – 12 October 2007, Poznan, Poland.
- [SnBo2007] Snowden and Boone. A Leader's Framework for Decision Making. *Harvard Business Review*, Nov. 2007.
- [StLiUn1989] Stein, Lynn Andrea, Henry Lieberman and David Ungar. A Shared View of Sharing: The Treaty of Orlando. Addendum to the OOPSLA '87 Conference Proceedings, ACM Press, 1989, 43—44.

- [StMeCo1974] Stevens, W. P., Myers, G. J., and Constantine, L. L. Structured Design. IBM Systems Journal 13(2), 1974, pp. 115—139.
- [Sut2003] Sutherland, Jeff, SCRUM: Another Way to think about scaling a project. 11 March 2003, accessed 28 November 2007.  
[http://jeffsutherland.org/scrum/2003\\_03\\_01\\_archive.html](http://jeffsutherland.org/scrum/2003_03_01_archive.html).
- [Sut2007] Sutherland, Jeff. Origins of Scrum. July 2007, accessed 20 July 2008,  
<http://jeffsutherland.com/scrum/2007/07/origins-of-scrum.html>
- [Sut2008a] Sutherland, Jeff. The First Scrum: Was it Scrum or Lean? <http://jeffsutherland.com/scrum/>, August 10, 2008, accessed 8 October, 2008.
- [TakNon1986] Takeuchi, Hirotaka, and Ikujiro Nonaka. The New New Product Development Game. Harvard Business Review, Reprint 86116, January-February 1986.
- [Wo+1991] Womack, James P., Daniel T. Jones, and Daniel Roos. The Machine that Changed the World: The Story of Lean Production. New York: Harper Perennial, 1991.
- [YoCo1975] Yourdon, E., and Constantine, L. L. Structured Design. Englewood Cliffs, NJ: Prentice-Hall, 1979; Yourdon Press, 1975.
- [Øre2008] —. Panel on Domain-Driven Design. Øredev 2008, Malmö, Sweden, 19 November 2008.

---

## *Leftovers*

As is true of a house, architecture just *is*: what the system does is what we make of it, tying parts of the architecture together with the activities of business life. A house is not a garden party, but a good architecture can make a garden party more enjoyable, convenient, and even beautiful. Such is the dance between architecture and function in software.