JUST ENOUGH SOFTWARE ARCHITECTURE

A RISK-DRIVEN APPROACH

GEORGE FAIRBANKS

FOREWORD BY DAVID GARLAN

Praise for *Just Enough Software Architecture: A Risk-Driven Approach*

If you're going to read only one book on software architecture, start with this one. *Just Enough Software Architecture* covers the essential concepts of software architecture that everyone — programmers, developers, testers, architects, and managers — needs to know, and it provides pragmatic advice that can be put into practice within hours of reading.

—Michael Keeling, professional software engineer

This book reflects the author's rare mix of deep knowledge of software architecture concepts and extensive industry experience as a developer. If you're an architect, you will want the developers in your organization to read this book. If you're a developer, do read it. The book is about architecture in real (not ideal) software projects. It describes a context that you'll recognize and then it shows you how to improve your design practice in that context.

—Paulo Merson, practicing software architect and Visiting Scientist at the Software Engineering Institute

Fairbanks' focus on "just enough" architecture should appeal to any developers trying to work out how to make the architecting process tractable. This focus is made accessible through detailed examples and advice that illustrate how an understanding of risk can be used to manage architecture development and scope. At the same time, Fairbanks provides detail on the more academic aspects of software architecture, which should help developers who are interested in understanding the broader theory and practice to apply these concepts to their projects.

—Dr. Bradley Schmerl, Senior Systems Scientist, School of Computer Science, Carnegie Mellon University

The Risk-Driven Model approach described in George Fairbanks' *Just Enough Software Architecture* has been applied to the eXtensible Information Modeler (XIM) project here at the NASA Johnson Space Center (JSC) with much success. It is a must for all members of the project, from project management to individual developers. In fact, it is a must for every developer's tool belt. The Code Model section and the anti-patterns alone are worth the cost of the book!

—Christopher Dean, Chief Architect, XIM, Engineering Science Contract Group – NASA Johnson Space Center

Just Enough Software Architecture will coach you in the strategic and tactical application of the tools and strategies of software architecture to your software projects. Whether you are a developer or an architect, this book is a solid foundation and reference for your architectural endeavors.

—Nicholas Sherman, Program Manager, Microsoft

Fairbanks synthesizes the latest thinking on process, lifecycle, architecture, modeling, and quality of service into a coherent framework that is of immediate applicability to IT applications. Fairbanks' writing is exceptionally clear and precise while remaining engaging and highly readable. *Just Enough Software Architecture* is an important contribution to the IT application architecture literature and may well become a standard reference work for enterprise application architects.

—Dr. Ian Maung, Senior VP of Enterprise Architecture at Citigroup and former Director of Enterprise Architecture at Covance

This book directly tackles some key needs of software practitioners who seek that blend of tools to help them create more effective systems, more effectively. George reaches frequently into his own experience, combining important ideas from academia to provide a conceptual model, selected best practices from industry to broaden coverage, and pragmatic guidance to make software architectures that are ultimately more useful and realistic. His simple risk-based approach frames much of the book and helps calibrate what "just-enough" should be. This book is an important addition to any software architecture bookshelf.

—Desmond D'Souza, Author of MAp and Catalysis, Kinetium, Inc.

This book shows how software architecture helps you build software instead of distracting from the job; the book lets you identify and address only those critical architectural concerns that would otherwise prevent you from writing code.

—Dr. Kevin Bierhoff, professional software engineer

System and software developers questioning *why* and *where* about software architecture will appreciate the clear arguments and enlightening analogies this book presents; developers struggling with *when* and *how* to do architecture will discover just-enough guidance, along with concepts and ideas that clarify, empower, and liberate. All in all, this book is easy to read, concise, yet rich with references — a well-architected and finely-designed book!

—Dr. Shang-Wen Cheng, flight software engineer

Just Enough Software Architecture

A Risk-Driven Approach

George Fairbanks

Published by Marshall & Brainerd
2445 7th Street
Boulder, CO 80304
(303) 834-7760

Library of Congress Control Number: 2010910450

ISBN 978-0-9846181-0-1

First printing, August 2010

Foreword

In the 1990s software architecture emerged as an explicit subfield of software engineering when it became widely recognized that getting the architecture right was a key enabler for creating a software-based system that met its requirements. What followed was a dizzying array of proposals for notations, tools, techniques, and processes to support architectural design and to integrate it into existing software development practices.

And yet, despite the existence of this body of material, principled attention to software architecture has in many cases not found its way into common practice. Part of the reason for this has been something of a polarization of opinions about the role that architecture should play. On one side is a school of thought that advocates architecture-focused design, in which architecture plays a pivotal and essential role throughout the software development process. People in this camp have tended to focus on detailed and complete architectural designs, well-defined architecture milestones, and explicit standards for architecture documentation. On the other side is a school of thought that deemphasizes architecture, arguing that it will emerge naturally as a by-product of good design, or that it is not needed at all since the architecture is obvious for that class of system. People in this camp have tended to focus on minimizing architectural design as an activity separate from implementation, and on reducing or eliminating architectural documentation.

Clearly, neither of these camps has it right for all systems. Indeed, the central question that must be asked is "How much explicit architectural design should one carry out for a given system?"

In this book, George Fairbanks proposes an answer: "Just Enough Architecture." One's first reaction to this might be "Well, duh!" because who would want too much or too little. But of course there is more to it than that, and it is precisely the detailing of principles for figuring out what "just enough" means that is the thrust of this

book. As such, it provides a refreshing and non-dogmatic way to approach software architecture — one with enormous practical value.

Fairbanks argues that the core criterion for determining how much architecture is enough is risk reduction. Where there is little risk in a design, little architecture is needed. But when hard system design issues arise, architecture is a tool with tremendous potential. In this way the book adopts a true engineering perspective on architecture, in the sense that it directly promotes the consideration of the costs and benefits in selecting a technique. Specifically, focusing on risk reduction aligns engineering benefits with costs by ensuring that architectural design is used in situations where it is likely to have the most payoff.

Naturally, there are a lot of secondary questions to answer. Which risks are best addressed with software architecture? How do you apply architectural design principles to resolve a design problem? What do you write down about your architectural commitments so that others know what they are? How can you help ensure that architectural commitments are respected by downstream implementers?

This book answers all of these questions, and many more — making it a uniquely practical and approachable contribution to the field of software architecture. For anyone who must create innovative software systems, for anyone who is faced with tough decisions about design tradeoffs, for anyone who must find an appropriate balance between agility and discipline — in short, for almost any software engineer — this is essential reading.

David Garlan
Professor, School of Computer Science
Director of Professional Software Engineering Programs
Carnegie Mellon University
May 2010

Preface

This is the book I wish I'd had when I started developing software. At the time, there were books on languages and books on object-oriented programming, but few books on design. Knowing the features of the C++ language does not mean you can design a good object-oriented system, nor does knowing the Unified Modeling Language (UML) imply you can design a good system architecture.

This book is different from other books about software architecture. Here is what sets it apart:

It teaches risk-driven architecting. There is no need for meticulous designs when risks are small, nor any excuse for sloppy designs when risks threaten your success. Many high-profile agile software proponents suggest that some up-front design can be helpful, and this book describes a way to do just enough architecture. It avoids the "one size fits all" process tar pit with advice on how to tune your architecture and design efforts based on the risks you face. The rigor of most techniques can be adjusted, from quick-and-dirty to meticulous.

It democratizes architecture. You may have software architects at your organization — indeed, you may be one of them. Every architect I have met wishes that all developers understood architecture. They complain that developers do not understand why constraints exist and how seemingly small changes can affect a system's properties. This book seeks to make architecture relevant to all software developers, not just architects.

It cultivates declarative knowledge. There is a difference between being able to hit a tennis ball and knowing why you are able to hit it, what psychologists refer to as *procedural knowledge* versus *declarative knowledge*. If you are already an expert at designing and building systems then you will have employed many of the techniques

found here, but this book will make you more aware of what you have been doing and provide names for the concepts. That declarative knowledge will improve your ability to mentor other developers.

It emphasizes the engineering. People who design and build software systems have to do many things, including dealing with schedules, resource commitments, and stakeholder needs. Many books on software architecture already cover software development processes and organizational structures. This book, in contrast, focuses on the technical parts of software development and deals with what developers do to ensure a system works — the *engineering*. It shows you how to build models and analyze architectures so that you can make principled design tradeoffs. It describes the techniques software designers use to reason about medium- to large-sized problems and points out where you can learn specialized techniques in more detail. Consequently, throughout this book, software engineers are referred to as *developers*, not differentiating architects from programmers.

It provides practical advice. This book offers a practical treatment of architecture. Software architecture is a kind of software design, but design decisions influence the architecture and vice versa. What the best developers do is drill down into obstacles in detail, understand them, then pop back up to relate the nature of those obstacles to the architecture as a whole. The approach in this book embraces this drill-down/pop-up behavior by describing models that have various levels of abstraction, from architecture to data structure design.

About me

My career has been a quest to learn how to build software systems. That quest has led me to interleave academics with industrial software development. I have the complete collector's set of computer science degrees: a BS, an MS, and a PhD (the PhD is from Carnegie Mellon University, in software engineering). For my thesis, I worked on software frameworks because they are a problem that many developers face. I developed a new kind of specification, called a design fragment, to describe how to use frameworks, and I built an Eclipse-based tool that can validate their correct usage. I was enormously fortunate to be advised by David Garlan and Bill Scherlis, and to have Jonathan Aldrich and Ralph Johnson on my committee.

I appreciate academic rigor, but my roots are in industry. I have been a software developer on projects including the Nortel DMS-100 central office telephone switch, statistical analysis for a driving simulator, an IT application at Time Warner Telecommunications, plug-ins for the Eclipse IDE, and every last stitch of code for my own web startup company. I tinker with Linux boxes as an amateur system administrator and have a closet lit by blinking lights and warmed by power supplies. I have sup-

ported agile techniques since their early days — in 1996 I successfully encouraged my department to switch from a six-month to a two-week development cycle, and in 1998 I started doing test-first development.

Who is this book for?

The primary audience for this book is practicing software developers. Readers should already know basic software development ideas — things like object-oriented software development, the UML, use cases, and design patterns. Some experience with how real software development proceeds will be exceedingly helpful, because many of this book's basic arguments are predicated on common experiences. If you have seen developers build too much documentation or do too little thinking before coding, you will know how software development can go wrong and therefore be looking for remedies like those offered in this book. This book is also suitable as a textbook in an advanced undergraduate or graduate level course.

Here is what to expect depending on what kind of reader you are:

Greenhorn developers or students. If you already have learned the basic mechanics of software development, such as programming languages and data structure design, and, ideally, have taken a general software engineering class, this book will introduce you to specific models of software that will help you form a *conceptual model* of software architecture. This model will help you make sense of the chaos of large systems without drawing a lot of diagrams and documentation. It may give you your first taste of ideas such as *quality attributes* and *architectural styles*. You will learn how to take your understanding of small programs and ramp it up to full industrial scale and quality. It can accelerate your progress toward becoming an effective, experienced developer.

Experienced developers. If you are good at developing systems then you will invariably be asked to mentor others. However, you may find that you have a somewhat idiosyncratic perspective on architecture, perhaps using unique diagram notations or terminology. This book will help you improve your ability to mentor others, understand why you are able to succeed where others struggle, and teach you about standard models, notations, and names.

Software architects. The role of software architect can be a difficult one when others in your organization do not understand what you do and why you do it. Not only will this book teach you techniques for building systems, it will also give you ways to explain what you are doing and how you are doing it. Perhaps you will even hand this book to co-workers so that you can better work as teammates.

Academics. This book makes several contributions to the field of software architecture. It introduces the *risk-driven model* of software architecture, which is a way of deciding how much architecture and design work to do on a project. It describes three approaches to architecture: *architecture-indifferent design*, *architecture-focused design*, and *architecture hoisting*. It integrates two perspectives on software architecture: the functional perspective and the quality-attribute perspective, yielding a single conceptual model. And it introduces the idea of an *architecturally-evident coding style* that makes your architecture evident from reading the source code.

Acknowledgments

This book would not have been possible without the generous assistance of many people. Several worked closely with me on one or more chapters and deserve special recognition for their help: Kevin Bierhoff, Alan Birchenough, David Garlan, Greg Hartman, Ian Maung, Paulo Merson, Bradley Schmerl, and Morgan Stanfield.

Others suffered through bad early drafts, caught huge numbers of problems, and provided needed guidance: Len Bass, Grady Booch, Christopher Dean, Michael Donohue, Daniel Dvorak, Anthony Earl, Hans Gyllstrom, Tim Halloran, Ralph Hoop, Michael Keeling, Ken LaToza, Thomas LaToza, Louis Marbel, Andy Myers, Carl Paradis, Paul Rayner, Patrick Riley, Aamod Sane, Nicholas Sherman, Olaf Zimmermann, and Guido Zgraggen. Thank you.

I would be remiss if I did not acknowledge all the people who have mentored me over the years, starting with my parents who provided more support than I can describe. My professional mentors have included Desmond D'Souza and the gang from Icon Computing; my thesis advisors, David Garlan and Bill Scherlis; and the faculty and students at Carnegie Mellon.

The wonderful cover illustration was conceived and drawn by my friend Lisa Haney (http://LisaHaney.com). Alan Apt has been a source of guidance and support through the book writing process.

The preparation of this book was done primarily with open source tools, including the Linux operating system, the LyX document processor, the Memoir LaTeX style, the LaTeX document preparation system, and the Inkscape drawing editor. Most diagrams were created using Microsoft Visio and Pavel Hruby's Visio UML template.

Contents

Chapter 1

Introduction

Decade after decade, software systems have seen orders-of-magnitude increases in their size and complexity. This is remarkable — and more than a little scary for those of us who build software. In contrast, imagine how hard basketball would be if it scaled up the same way, with 5 people on the floor one decade, then 50, then 500. Because of this growth, today's software systems are arguably the largest and most complex things ever built.

Software developers are always battling the ever-stronger foes of complexity and scale, but even though their opponent grows in strength, developers have staved off defeat and even reveled in victory. How have they done this?

One answer is that the increases in software size and complexity have been matched by advances in software engineering. Assembly language programming gave way to higher-level languages and structured programming. Procedures have, in many domains, given way to objects. And software reuse, which used to mean just subroutines, is now also done with extensive libraries and frameworks.

It is no coincidence that the battle between developers and software complexity always seems to be at a stalemate. Since developers cannot grow bigger brains, they have instead improved their weapons. An improved weapon gives developers two options: to more easily conquer yesterday's problems, or to combat tomorrow's. We are no smarter than developers of the previous generation, but our improved weapons allow us to build software of greater size and complexity.

Software developers wield some tangible weapons, such as Integrated Development Environments (IDEs) and programming languages, but intangible weapons arguably make a bigger impact. Returning to our basketball metaphor, consider a coach

and a rookie (a novice sports player) watching the same game. The coach sees more than the rookie — not because the coach's eyes are more acute, but because he has an intangible weapon. He has built up a set of mental abstractions that allow him to convert his perceptions of raw phenomena, such as a ball being passed, into a condensed and integrated understanding of what is happening, such as the success of an offensive strategy. The coach watches the same game that the rookie does, but he understands it better. Alan Kay has observed that your "point of view is worth 80 IQ points" (Kay, 1989).

Software is similar in that there are lots of low-level details. If developers have built up a set of mental abstractions (i.e., a conceptual model), they can convert those details into a condensed understanding: where before they saw just code, perhaps they now see a thread-safe locking policy or an event-driven system.

1.1 Partitioning, knowledge, and abstractions

To be successful at combating the scale and complexity of software in the next decade, developers will need improved weapons. Those weapons can be categorized, perhaps with a bit of shoehorning, into three categories: partitioning, knowledge, and abstraction. Developers *partition* a problem so that its parts are smaller and more tractable, they apply *knowledge* of similar problems, and they use *abstractions* to help them reason. Partitioning, knowledge, and abstraction are effective because they enable our fixed-sized minds to comprehend an ever-growing problem.

- **Partitioning**. Partitioning is effective as a strategy to combat complexity and scale when two conditions are true: first, the divided parts must be sufficiently small that a person can now solve them; second, it must be possible to reason about how the parts assemble[1] into a whole. Parts that are encapsulated are easier to reason about, because you need to track fewer details when composing the parts into a solution. You can forget, at least temporarily, about the details inside the other parts. This allows the developer to more easily reason about how the parts will interact with each other.

- **Knowledge**. Software developers use knowledge of prior problems to help them solve current ones. This knowledge can be implicit know-how or explicitly written down. It can be specific, as in which components work well with others, or general, as in techniques for optimizing a database table layout. It comes in many forms, including books, lectures, pattern descriptions, source code, design documents, or sketches on a whiteboard.

[1] Mary Shaw has remarked that when dividing and conquering, the dividing is the easy part.

- **Abstraction**. Abstraction can effectively combat complexity and scale because it shrinks problems, and smaller problems are easier to reason about. If you are driving from New York to Los Angeles, you can simplify the navigation problem by considering only highways. By hiding details (excluding the option of driving across fields or parking lots), you have shrunken the number of options to consider, making the problem easier to reason about.

You should not expect any *silver bullets*, as Fred Brooks called them, that will suddenly eliminate the difficulties of software development (Brooks, 1995). Instead, you should look for weapons that help you partition systems better, provide knowledge, and enable abstraction to reveal the essence of the problem.

Software architecture is one such weapon and it can help you address the complexity and scale of software systems. It helps you *partition* software, it provides *knowledge* that helps you design better software, and it provides *abstractions* that help you reason about software. It is a tool in the hands of a skilled developer. It helps software developers to routinely build systems that previously required virtuosos (Shaw and Garlan, 1996), but it does not eliminate the need for skilled software developers. Instead of removing the need for ingenuity, it allows developers to apply their ingenuity to build bigger and more complex systems.

1.2 Three examples of software architecture

That is what software architecture *does* for you, but what *is* it? Roughly speaking, architecture is the macroscopic design of a software system. Chapter 2 of this book discusses a more careful definition, but perhaps it is best to understand software architecture using some concrete examples first.

It can be hard to "see the forest for the trees," which in this case means finding the architecture amidst the design details. But by comparing multiple similar systems with different architectures you should be able to notice what is different and therefore identify their architectures. What follows is a description of three systems with the same functionality, yet different architectures, based on the experiences at Rackspace.

Rackspace is a real company that manages hosted email servers. Customers call up for help when they experience problems. To help a customer, Rackspace must search the log files that record what has happened during the customer's email processing. Because the volume of emails they handle kept increasing, Rackspace built three generations of systems to handle the customer queries (Hoff, 2008b; Hood, 2008).

Version 1: Local log files. The first version of the program was simple. There were already dozens of email servers generating log files. Rackspace wrote a script that

would use ssh to connect to each machine and execute a grep query on the mail log file. Engineers could control the search results by adjusting the grep query.

This version initially worked well, but over time the number of searches increased and the overhead of running those searches on the email servers became noticeable. Also, it required an engineer, rather than a support tech, to perform the search.

Version 2: Central database. The second version addressed the drawbacks of the first one by moving the log data off of the email servers and by making it searchable by support techs. Every few minutes, each email server would send its recent log data to a central machine where it was loaded into a relational database. Support techs had access to the database data via a web-based interface.

Rackspace was now handling hundreds of email servers, so the volume of log data had increased correspondingly. Rackspace's challenge became how to get the log data into the database as quickly and efficiently as possible. The company settled on bulk record insertion into merge tables, which enabled loading of the data in two or three minutes. Only three days worth of logs were kept so that the database size would not hinder performance.

Over time, this system also encountered problems. The database server was a single machine and, because of the constant loading of data and the query volume, it was pushed to its limit with heavy CPU and disk loads. Wildcard searches were prohibited because of the extra load they put on the server. As the amount of log data grew, searches became slower. The server experienced seemingly random failures that became increasingly frequent. Any log data that was dropped was gone forever because it was not backed up. These problems led to a loss of confidence in the system.

Version 3: Indexing cluster. The third version addressed the drawbacks of the second by saving log data into a distributed file system and by parallelizing the indexing of log data. Instead of running on a single powerful machine, it used ten commodity machines. Log data from the email servers streamed into the Hadoop Distributed File System, which kept three copies of everything on different disks. In 2008, when Rackspace wrote a report on its experiences, it had over six terabytes of data spanning thirty disk drives, which represented six months of search indexes.

Indexing was performed using Hadoop, which divides the input data, indexes (or "maps") in jobs, then combines (or "reduces") the partial results into a complete index. Jobs ran every ten minutes and took about five minutes to complete, so index results were about fifteen minutes stale. Rackspace was able to index over 140 gigabytes of log data per day and had executed over 150,000 jobs since starting the system.

As in the second system, support techs had access via a web interface that was much like a web search-engine interface. Query results were provided within seconds.

When engineers thought up new questions about the data, they could write a new kind of job and have their answer within a few hours.

1.3 Reflections

The first thing to notice from looking at these three systems is that they all have roughly the same functionality (querying email logs to diagnose problems) yet they have different architectures. *Their architecture was a separate choice from their functionality.* This means that when you build a system, you can choose an architecture that best suits your needs, then build the functionality on that architectural skeleton. What else can these systems reveal about software architecture?

Quality attributes. Despite having the same functionality, the three systems differ in their modifiability, scalability, and latency. For example, in the first and second version, ad hoc queries could be created in a matter of seconds, either by changing the grep expression used for the search or by changing the SQL query. The third system requires a new program to be written and scheduled before query results can be obtained. All three support creating new queries, but they differ in how easy it is to do (modifiability).

Notice also that there was no free lunch: promoting one quality inhibited another. The third system was much more scalable than the other two, but its scalability came at the price of reduced ability to make ad hoc queries and a longer wait before results were available. The data in the first system was queryable online (and the second could perhaps be made nearly so), but the third system had to collect data then run a batch process to index the results, which means that query results were a bit stale.

If you are ever in a situation in which you can get great scalability, latency, modifiability, etc., then you should count yourself lucky, because such *quality attributes* usually trade off against each other. Maximizing one quality attribute means settling for less of the others. For example, choosing a more modifiable design may entail worse latency.

Conceptual model. Even without being a software architecture expert, you could read about these three systems and reason about their design from first principles. What advantage would come from being a software architecture expert (i.e., being a coach instead of a rookie)? Both coaches and rookies have an innate ability to reason, but the coach has a head start because of the *conceptual model* he carries in his head that helps him make sense of what he sees.

As an architecture expert, you would be primed to notice the partitioning differences in each system. You would distinguish between chunks of code (modules), runtime chunks (components), and hardware chunks (nodes, or environmental elements). You would notice and already know the names of the architectural patterns

each system employs. You would know which pattern is suited for achieving which quality attributes, so you would have predicted that the client-server system would have lower latency than the map-reduce system. You would sort out and relate domain facts, design choices, and implementation details — and notice when someone else has jumbled them together. Being an expert in software architecture helps you to use your innate reasoning abilities more effectively.

Abstractions and constraints. In software, bigger things are usually built out of smaller things. You can always reason about the smaller things in a system (like individual lines of code) but usually you will find it more efficient to reason about the larger things (like clients and servers). For example, the third system in the Rackspace example scheduled jobs and stored data in a distributed file system. If you needed to solve a problem about how jobs flow through that system, then it would be most efficient to reason about the problem at that level. You could begin your reasoning process by considering the small bits, like objects and procedure calls, but that would be inefficient and likely to swamp you with details.

Furthermore, a "job" is an abstraction that obeys more constraints than an arbitrary "chunk of code" does. The developers imposed those constraints to make reasoning about the system easier. For example, if they constrain jobs to have no side effects, they can run the same job twice, in parallel, just in case one becomes bogged down. It is hard to reason about an arbitrary chunk of code specifically because you cannot tell what it *does not* do. Developers voluntarily impose constraints on the system in order to amplify their reasoning abilities.

1.4 Perspective shift

In 1968 Edsger Dijkstra wrote a now famous letter titled "GOTO Considered Harmful" advocating the use of structured programming. His argument is roughly as follows: Developers build programs containing static statements that execute to produce output. Developers, being human, have a hard time envisioning how the static statements of a program will execute at runtime. GOTO statements complicate the reasoning about runtime execution, so it is best to avoid GOTO statements and embrace structured programming.

Looking back at this debate today, it is hard to imagine disagreeing strongly, but at the time the resistance was substantial. Developers were accustomed to working within the old set of abstractions. They focused on the constraints of the new abstractions rather than the benefits and objected based on corner cases that were hard to express using structured programming. Each similar increase in abstraction is opposed by some who are familiar with the old abstractions. During my programming career, I have seen developers resist abstract data types and object-oriented programming, only to later embrace them.

New architecture abstractions rarely replace old abstractions, but instead coexist with them. Using abstractions such as components and connectors does not mean that objects, methods, and data structures disappear. Similarly, forest fire fighters switch between thinking about individual trees or entire forests depending on what part of their job they are doing at the moment.

Effectively applying software architecture ideas requires a conscious and explicit shift to embrace its abstractions, such as components and connectors, rather than only using the abstractions found in mainstream programming language (often just classes or objects). If you do not consciously choose the architecture of your systems, then it may end up being what Brian Foote and Joseph Yoder call a *big ball of mud* (Foote and Yoder, 2000), which they estimate is the most common software architecture. It is easy to understand this architecture: Imagine what a system with 10 classes would look like, then scale it up to 100, 1000, ..., without any new abstractions or partitions, and let the objects in the system communicate with each other as is expedient.

1.5 Architects architecting architectures

I have sometimes seen software developers design systems with beautiful architectures, then voice their resistance to software architecture. This resistance may stem from resistance to bureaucracy-intensive up-front design processes, to pompous architects, or to having been forced to waste time making diagrams instead of systems. Fortunately, none of these problems need to be a part of creating software architecture.

Job titles, development processes, and engineering artifacts are separable, so it is important to avoid conflating the *job title* "architect," the *process* of architecting a system, and the *engineering artifact* that is the software architecture.

- **The job role: architect.** One possible job title (or role) in an organization is that of a software architect. Some architects sit in corner offices and make pronouncements that are disconnected from the engineering reality, while other architects are intimately involved in the ongoing construction of the software. Either way, the title and the office are not intrinsic to the work of designing or building software. All software developers, not just architects, should understand their software's architecture.

- **The process: architecting.** There is no software at the beginning of a project, but by the end of the project there is a running system. In between, the team performs activities (i.e., they follow a process) to construct the system. Some teams design up-front and other teams design as they build. The process that the team follows is separable from the design that emerges. A team could follow any number of different processes and produce, for example, a 3-tier system. Or

put another way, it is nearly impossible to tell what architecting process a team followed by looking only at the finished software.

- **The engineering artifact: the architecture.** If you look at an automobile, you can tell what type of car it is, perhaps an all-electric car, a hybrid car, or an internal combustion car. That characteristic of the car is distinct from the process followed to design it, and distinct from the job titles used by its designers. The car's design is an engineering artifact. Different choices about process and job titles could still result in their creating, for example, a hybrid car. Software is similar. If you look at a finished software system, you can distinguish various designs; for example: collaborating peer-to-peer nodes in a voice-over-IP network, multiple tiers in information technology (IT) systems, or parallelized map-reduce compute nodes in internet systems. Every software system has an architecture just as every car has a design. Some software is cobbled together without a regular process, yet its architecture is still visible.

This book discusses process in Chapter 2 and Chapter 3. The rest of the book treats architecture as an engineering artifact: something to be analyzed, understood, and designed so that you can build better systems.

1.6 Risk-driven software architecture

Different developers have had success with different processes. Some succeeded with agile processes that have little or no up-front work. Others succeeded with detailed up-front design work. Which should you choose? Ideally, you would have a guiding principle that would help you choose appropriately.

This book suggests using *risk* to decide how much architecture work to do. One way to understand how risk can guide you to good choices about architecture it is to consider the story of my father installing a mailbox.

My father has two degrees in mechanical engineering, but when he put up a mailbox he did it like anyone else would: he dug a hole, put in the post, and filled the hole with some cement. Just because he could calculate moments, stresses, and strains does not mean he must or should. In other situations it would be foolish for him to skip these analyses. How did he know when to use them?

Software architecture is a relatively new technology, and it includes many techniques for modeling and analyzing systems. Yet each of these techniques takes time that could otherwise be spent building the system. This book introduces the *risk-driven model* for software architecture, which guides you to do just enough architecture by selecting appropriate architecture techniques and knowing when you can stop.

Your effort should be commensurate with your risk of failure. Perhaps your system has demanding scalability requirements because you run a popular web service. It would be best to ensure that your system will handle the expected number of users[2] before you invest too much energy in its design (or the site goes live). If modifiability is less of a concern for your system (or usability, etc.), you would spend little time worrying about that risk.

Each project faces different risks, so there is no single correct way to do software architecture: you must evaluate the risks on each project. Sometimes the answer is to do no architecture work, because some projects are so highly precedented that there is almost no risk as long as you reuse a proven architecture. However, when you work in novel domains or push an existing system into uncharted territory you will want to be more careful.

The idea of consistently working to reduce engineering risks echoes Barry Boehm's *spiral model* of software development (Boehm, 1988). The spiral model is a full software development process that guides projects to work on the highest risk items first. Projects face both management and engineering risks, so managers must prioritize both management risks (like the risk of customer rejection) and engineering risks (like the risk that the system is insecure or inefficient).

Compared to the spiral model, the risk-driven model helps you answer narrower questions: How much architecture work should you do, and what kind of architecture techniques should you use? Because the risk-driven model only applies to design work, it means that it can be applied to agile processes, waterfall processes, spiral processes, etc. Regardless of process, you must design the software — the difference is when design happens and which techniques are applied.

1.7 Architecture for agile developers

Agile software development is a reaction to heavyweight development processes and it emphasizes efficiently building products that customers want (Beck et al., 2001). It is increasingly popular, with one study showing 69% of companies trying it on at least some of their projects (Ambler, 2008).

In their desire to cut out unnecessary steps in software development, some agile developers believe they should avoid software architecture techniques. This reluctance is not universal, as many important voices in agile community support some planned design work, including Martin Fowler, Robert Martin, Scott Ambler, and Granville Miller (Fowler, 2004; Martin, 2009; Ambler, 2002; Miller, 2006). Refactoring a poor architecture choice can be prohibitively expensive on large systems. This

[2]Recall that before the popular Facebook and MySpace social networking sites there was Friendster, but it could not handle the rush of users and became too slow to use.

book can help agile developers to use software architecture in a way that is consistent with agile principles for two primary reasons:

- **Just enough architecture.** The risk-driven model of architecture guides developers to do just enough architecture then resume coding. If you only foresee risks on your project that can be handled by refactoring then you would not do any architecture design. But if you are an agilist who is also concerned that refactoring may not be enough to get the security or scalability you need, now you have a way to mitigate those risks — even in your Nth iteration — then get back to coding.
- **Conceptual model.** Agile's primary contribution is to software development process, not design abstractions, and offers a limited number of techniques (such as refactoring and spikes) to produce good designs. The contents of this book augment agile processes by providing a conceptual model to reason about a system's architecture and design, a set of software design and modeling techniques, and expert software architecture knowledge.

This is not a book specifically about agile software architecture, but you will find that its risk-driven approach is well suited to agile projects. In particular, Section 3.11 provides a sketch of how to integrate risks into an iterative, feature-centric process.

1.8 About this book

This book focuses on software architecture as it relates to the *construction of software*, and describes the techniques used to ensure that software satisfies its engineering demands. This book is largely process agnostic because the engineering techniques themselves are largely process agnostic. You will not find advice on management activities like the political responsibilities of architects, when to hold specific kinds of meetings, or how to gather requirements from stakeholders.

This book is divided into two parts. This first part introduces software architecture and the risk-driven approach. The second part helps you build up a mental conceptual model of software architecture and describes in detail the abstractions like components and connectors. What follows is short summaries of each part.

Part I: Risk-driven software architecture

A definition of software architecture is difficult to pin down precisely, but several things about it are quite clear. Software developers, like engineers in other specialties, use abstraction and models to solve large and complex problems. Software architecture acts as the skeleton of a system, influences quality attributes, is orthogonal to functionality, and uses constraints to influence a system's properties. Architecture

is most important when the solution space is small, the failure risks are high, or you face difficult quality attribute demands. You can choose from architecture-indifferent design, architecture-focused design, or even architecture hoisting.

Risks can be used to guide you regarding which design and architecture techniques you should use and regarding how much design and architecture you should do. At its core, the risk-driven model is simple: (1) identify and prioritize risks, (2) select and apply a set of techniques, and (3) evaluate risk reduction.

To reveal the risk-driven model in practice, Chapter 4 provides an example of applying the risk-driven model to a Home Media Player system. The developers on that system have the challenges of team communication, integration of COTS components, and ensuring metadata consistency.

The first part of the book concludes with advice on using models and software architecture, including: use models to solve problems, add constraints judiciously, focus on risks, and distribute architecture skills throughout your team.

Part II: Architecture modeling

The second part of the book helps you build a mental conceptual model of software architecture. It starts with the canonical model structure: the domain model, the design model, and the code model. The domain model corresponds to things in the real world, the design model is the design of the software you are building, and the code model corresponds to your source code. You can build additional models that show selected details, called views, and these views can be grouped into viewtypes.

Building encapsulation boundaries is a crucial skill in software architecture. Users of a component or module can often ignore how it works internally, freeing their minds to solve other hard problems. And the builders of an encapsulated component or module have the freedom to change its implementation without perturbing its users. Builders will only have that freedom if the encapsulation is effective, so this book will teach you techniques for ensuring that it is effective.

A great number of architectural abstractions and modeling techniques have been built up over the years. This book consolidates software architecture techniques found in a variety of other sources, integrating techniques emphasizing quality attributes as well as techniques emphasizing functionality. It also discusses pragmatic ways to build effective models and debug them.

The second part of the book concludes with advice on how to use the models effectively. Any book that covered the advantages but not the pitfalls of a technology should not be trusted, so it also covers problems that you are likely to encounter. By the end of the second part, you should have built up a rich conceptual model of abstractions and relationships that will help you see software systems the way a coach sees a game.

Chapter 3

Risk-Driven Model

As they build successful software, software developers are choosing from alternate designs, discarding those that are doomed to fail, and preferring options with low risk of failure. When risks are low, it is easy to plow ahead without much thought, but, invariably, challenging design problems emerge and developers must grapple with high-risk designs, ones they are not sure will work.

Building successful software means anticipating possible failures and avoiding designs that could fail. Henry Petroski, a leading historian of engineering, says this about engineering as a whole:

> The concept of failure is central to the design process, and it is by thinking in terms of obviating failure that successful designs are achieved. ... Although often an implicit and tacit part of the methodology of design, failure considerations and proactive failure analysis are essential for achieving success. And it is precisely when such considerations and analyses are incorrect or incomplete that design errors are introduced and actual failures occur. (Petroski, 1994)

To address failure risks, the earliest software developers invented design techniques, such as domain modeling, security analyses, and encapsulation, that helped them build successful software. Today, developers can choose from a huge number of design techniques. From this abundance, a hard question arises: *Which design and architecture techniques should developers use?*

If there were no deadlines then the answer would be easy: use all the techniques. But that is impractical because a hallmark of engineering is the *efficient* use of re-

sources, including time. One of the risks developers face is that they waste too much time designing. So a related question arises: *How much design and architecture should developers do?*

There is much active debate about this question and several kinds of answers have been suggested:

- **No up-front design.** Developers should just write code. Design happens, but is coincident with coding, and happens at the keyboard rather than in advance.
- **Use a yardstick.** For example, developers should spend 10% of their time on architecture and design, 40% on coding, 20% on integrating, and 30% on testing.
- **Build a documentation package.** Developers should employ a comprehensive set of design and documentation techniques sufficient to produce a complete written design document.
- **Ad hoc.** Developers should react to the project needs and decide on the spot how much design to do.

The ad hoc approach is perhaps the most common, but it is also subjective and provides no enduring lessons. Avoiding design altogether is impractical when failure risks are high, but so is building a complete documentation package when risks are low. Using a yardstick can help you plan how much effort designing the architecture will take, but it does not help you choose techniques.

This chapter introduces the *risk-driven model* of architectural design. Its essential idea is that the effort you spend on designing your software architecture should be commensurate with the risks faced by your project. When my father installed a new mailbox, he did not apply every mechanical engineering analysis and design technique he knew. Instead, he dug a hole, put in a post, and filled the hole with concrete. The risk-driven model can help you decide when to apply architecture techniques and when you can skip them.

Where a software development process orchestrates every activity from requirements to deployment, the risk-driven model guides only architectural design, and can therefore be used inside any software development process.

The risk-driven model is a reaction to a world where developers are under pressure to build high quality software quickly and at reasonable cost, yet those developers have more architecture techniques than they can afford to apply. The risk-driven model helps them answer the two questions above: how much software architecture work should they do, and which techniques should they use? It is an approach that helps developers follow a middle path, one that avoids wasting time on techniques that help their projects only a little but ensures that project-threatening risks are addressed by appropriate techniques.

In this chapter, we will examine how risk reduction is central to all engineering disciplines, learn how to choose techniques to reduce risks, understand how engineering risks interact with management risks, and learn how we can balance planned design with evolutionary design. This chapter walks through the ideas that underpin the risk-driven model, but if you are the kind of person who would prefer to first see an example of it in use, you can flip ahead to Chapter 4.

3.1 What is the risk-driven model?

The *risk-driven model* guides developers to apply a minimal set of architecture techniques to reduce their most pressing risks. It suggests a relentless questioning process: "What are my risks? What are the best techniques to reduce them? Is the risk mitigated and can I start (or resume) coding?" The risk-driven model can be summarized in three steps:

1. Identify and prioritize risks
2. Select and apply a set of techniques
3. Evaluate risk reduction

You do not want to waste time on low-impact techniques, nor do you want to ignore project-threatening risks. You want to build successful systems by taking a path that spends your time most effectively. That means addressing risks by applying architecture and design techniques but only when they are motivated by risks.

Risk or feature focus. The key element of the risk-driven model is the promotion of risk to prominence. What you choose to promote has an impact. Most developers already think about risks, but they think about lots of other things too, and consequently risks can be overlooked. A recent paper described how a team that had previously done up-front architecture work switched to a purely feature-driven process. The team was so focused on delivering features that they deferred quality attribute concerns until after active development ceased and the system was in maintenance (Babar, 2009). The conclusion to draw is that teams that focus on features will pay less attention to other areas, including risks. Earlier studies have shown that even architects are less focused on risks and tradeoffs than one would expect (Clerc, Lago and van Vliet, 2007).

Logical rationale. But what if your perception of risks differs from others' perceptions? Risk identification, risk prioritization, choice of techniques, and evaluation of risk mitigation will all vary depending on who does them. Is the risk-driven model is merely improvisation?

No. Though different developers will perceive risks differently and consequently choose different techniques, the risk-driven model has the useful property that it yields arguments that can be evaluated. An example argument would take this form:

> We identified A, B, and C as risks, with B being primary. We spent time applying techniques X and Y because we believed they would help us reduce the risk of B. We evaluated the resulting design and decided that we had sufficiently mitigated the risk of B, so we proceeded on to coding.

This allows you to answer the broad question, "How much software architecture should you do?" by providing a plan (i.e., the techniques to apply) based on the relevant context (i.e., the perceived risks).

Other developers might disagree with your assessment, so they could provide a differing argument with the same form, perhaps suggesting that risk D be included. A productive, engineering-based discussion of the risks and techniques can ensue because the rationale behind your opinion has been articulated and can be evaluated.

3.2 Are you risk-driven now?

Many developers believe that they already follow a risk-driven model, or something close to it. Yet there are telltale signs that many do not. One is an inability to list the risks they confront and the corresponding techniques they are applying.

Any developer can answer the question, "Which features are you working on?" but many have trouble with the question, "What are your primary failure risks and corresponding engineering techniques?" If risks were indeed primary then they would find it an easy question to answer.

Technique choices should vary. Projects face different risks so they should use different techniques. Some projects will have tricky quality attribute requirements that need up-front planned design, while other projects are tweaks to existing systems and entail little risk of failure. Some development teams are distributed and so they document their designs for others to read, while other teams are co-located and can reduce this formality.

When developers fail to align their architecture activities with their risks, they will over-use or under-use architectural techniques, or both. Examining the overall context of software development suggests why this can occur. Most organizations guide developers to follow a process that includes some kind of documentation template or a list of design activities. These can be beneficial and effective, but they can also inadvertently steer developers astray.

Here are some examples of well-intentioned rules that guide developers to activities that may be mismatched with their project's risks.

- The team must always (or never) build full documentation for each system.
- The team must always (or never) build a class diagram, a layer diagram, etc.
- The team must spend 10% (or 0%) of the project time on architecture.

Such guidelines can be better than no guidance, but each project will face a different set of risks. It would be a great coincidence if the same set of diagrams or techniques were always the best way to mitigate a changing set of risks.

Example mismatch. Imagine a company that builds a 3-tier system. The first tier has the user interface and is exposed to the internet. Its biggest risks might be usability and security. The second and third tiers implement business rules and persistence; they are behind a firewall. The biggest risks might be throughput and scalability.

If this company followed the risk-driven model, the front-end and back-end developers would apply different architecture and design techniques to address their different risks. Instead, what often happens is that both teams follow the same company-standard process or template and produce, say, a module dependency diagram. The problem is that there is no connection between the techniques they use and the risks they face.

Standard processes or templates are not necessarily bad, but they are often used poorly. Over time, you may be able to generalize the risks on the projects at your company and devise a list of appropriate techniques. The important part is that the techniques match the risks.

The three steps to risk-driven software architecture are deceptively simple but the devil is in the details. What exactly are risks and techniques? How do you choose an appropriate set of techniques? And when do you stop architecting and start/resume building? The following sections dig into these questions in more detail.

3.3 Risks

In the context of engineering, *risk* is commonly defined as the chance of failure times the impact of that failure. Both the probability of failure and the impact are uncertain because they are difficult to measure precisely. You can sidestep the distinction between perceived risks and actual risks by bundling the concept of uncertainty into the definition of risk. The definition of risk then becomes:

$$\text{risk} = \text{perceived probability of failure} \times \text{perceived impact}$$

A result of this definition is that a risk can exist (i.e., you can perceive it) even if it does not exist. Imagine a hypothetical program that has no bugs. If you have never run the program or tested it, should you worry about it failing? That is, should you perceive a failure risk? Of course you should, but after you analyze and test the program, you gain confidence in it, and your perception of risk goes down. So by

Project management risks	Software engineering risks
“Lead developer hit by bus”	“The server may not scale to 1000 users”
“Customer needs not understood”	“Parsing of the response messages may be buggy”
“Senior VP hates our manager”	“The system is working now but if we touch anything it may fall apart”

Figure 3.1: Examples of project management and engineering risks. You should distinguish them because engineering techniques rarely solve management risks, and vice versa.

applying techniques, you can reduce the amount of uncertainty, and therefore the amount of (perceived) risk. You can also under-appreciate or fail to perceive a risk, which we will discuss shortly.

Describing risks. You can state a risk categorically, often as the lack of a needed quality attribute like modifiability or reliability. But often this is too vague to be actionable: if you do something, are you sure that it actually reduces the categorical risk?

It is better to describe risks such that you can later test to see if they have been mitigated. Instead of just listing a quality attribute like reliability, describe each risk of failure as a testable *failure scenario*, such as “During peak loads, customers experience user interface latencies greater than five seconds.”

Engineering and project management risks. Projects face many different kinds of risks, so people working on a project tend to pay attention to the risks related to their specialty. For example, the sales team worries about a good sales strategy and software developers worry about a system’s scalability. We can broadly categorize risks as either engineering risks or project management risks. *Engineering risks* are those risks related to the analysis, design, and implementation of the product. These engineering risks are in the domain of the engineering of the system. *Project management risks* relate to schedules, sequencing of work, delivery, team size, geography, etc. Figure 3.1 shows examples of both.

If you are a software developer, you are asked to mitigate engineering risks and you will be applying engineering techniques. The technique type must match the risk type, so only engineering techniques will mitigate engineering risks. For example, you cannot use a PERT chart (a project management technique) to reduce the chance of buffer overruns (an engineering risk), and using Java will not resolve stakeholder disagreements.

Project domain	Prototypical risks
Information Technology (IT)	Complex, poorly understood problem Unsure we're solving the real problem May choose wrong COTS software Integration with existing, poorly understood software Domain knowledge scattered across people Modifiability
Systems	Performance, reliability, size, security Concurrency Composition
Web	Security Application scalability Developer productivity / expressability

Figure 3.2: While each project can have a unique set of risks, it is possible to generalize by domain. Prototypical risks are ones that are common in a domain and are a reason that software development practices vary by domain. For example, developers on Systems projects tend to use the highest performance languages.

Identifying risks. Experienced developers have an easy time identifying risks, but what can be done if the developer is less experienced or working in an unfamiliar domain? The easiest place to start is with the requirements, in whatever form they take, and looking for things that seem difficult to achieve. Misunderstood or incomplete quality attribute requirements are a common risk. You can use Quality Attribute Workshops (see Section 15.6.2), a Taxonomy-Based Questionnaire (Carr et al., 1993), or something similar, to elicit risks and produce a prioritized list of failure scenarios.

Even with diligence, you will not be able to identify every risk. When I was a child, my parents taught me to look both ways before crossing the street because they identified cars as a risk. It would have been equally bad if I had been hit by a car or by a falling meteor, but they put their attention on the foreseen and high priority risk. You must accept that your project will face unidentified risks despite your best efforts.

Prototypical risks. After you have worked in a domain for a while, you will notice *prototypical risks* that are common to most projects in that domain. For example, Systems projects usually worry more about performance than IT projects do, and Web projects almost always worry about security. Prototypical risks may have been encoded as *checklists* describing historical problem areas, perhaps generated from architecture reviews. These checklists (see Section 15.6.2) are valuable knowledge

for less experienced developers and a helpful reminder for experienced ones.

Knowing the prototypical risks in your domain is a big advantage, but even more important is realizing when your project differs from the norm so that you avoid blind spots. For example, software that runs a hospital might most closely resemble an IT project, with its integration concerns and complex domain types. However, a system that takes 10 minutes to reboot after a power failure is usually a minor risk for an IT project, but a major risk at a hospital.

Prioritizing risks. Not all risks are equally large, so they can be *prioritized*. Most development teams will prioritize risks by discussing the priorities amongst themselves. This can be adequate, but the team's perception of risks may not be the same as the stakeholders' perception. If your team is spending enough time on software architecture for it to be noticeable in your budget, it is best to validate that time and money are being spent in accordance with stakeholder priorities.

Risks can be categorized[1] on two dimensions: their priority to stakeholders and their perceived difficulty by developers. Be aware that some technical risks, such as platform choices, cannot be easily assessed by stakeholders.

Formal procedures exist for cataloging and prioritizing risks using risk matrices, including a US military standard MIL-STD-882D. Formal prioritization of risks is appropriate if your system, for example, handles radioactive material, but most computer systems can be less formal.

3.4 Techniques

Once you know what risks you are facing, you can apply *techniques* that you expect to reduce the risk. The term technique is quite broad, so we will focus specifically on *software engineering risk reduction techniques*, but for convenience continue to use the simple name *technique*. Figure 3.3 shows a short list of software engineering techniques and techniques from other engineering branches.

Spectrum from analyses to solutions. Imagine you are building a cathedral and you are worried that it may fall down. You could build models of various design alternatives and calculate their stresses and strains. Alternately, you could apply a known solution, such as using a flying buttress. Both work, but the former approach has an analytical character while the latter has a known-good solution character.

Techniques exist on a spectrum from pure analyses, like calculating stresses, to pure solutions, like using a flying buttress on a cathedral. Other software architecture and design books have inventoried techniques on the solution-end of the spectrum, and call these techniques *tactics* (Bass, Clements and Kazman, 2003) or *patterns*

[1] This is the same categorization technique used in ATAM to prioritize architecture drivers and quality attribute scenarios, as discussed in Section 12.11.

Software engineering	Other engineering
Applying design or architecture pattern	Stress calculations
Domain modeling	Breaking point test
Throughput modeling	Thermal analysis
Security analysis	Reliability testing
Prototyping	Prototyping

Figure 3.3: A few examples of engineering risk reduction techniques in software engineering and other fields. Modeling is commonplace in all engineering fields.

(Schmidt et al., 2000; Gamma et al., 1995), and include such solutions as using a process monitor, a forwarder-receiver, or a model-view-controller.

The risk-driven model focuses on techniques that are on the analysis-end of the spectrum, ones that are procedural and independent of the problem domain. These techniques include using models such as layer diagrams, component assembly models, and deployment models; applying analytic techniques for performance, security, and reliability; and leveraging architectural styles such as client-server and pipe-and-filter to achieve an emergent quality.

Techniques mitigate risks. Design is a mysterious process, where virtuosos can make leaps of reasoning between problems and solutions (Shaw and Garlan, 1996). For your process to be repeatable, however, you need to make explicit what the virtuosos are doing tacitly. In this case, you need to be able to explicitly state how to choose techniques in response to risks. Today, this knowledge is mostly informal, but we can aspire to creating a handbook that would help us make informed decisions. It would be filled with entries that look like this:

> If you have <a risk>, consider <a technique> to reduce it.

Such a handbook would improve the repeatability of designing software architectures by encoding the knowledge of virtuoso architects as mappings between risks and techniques.

Any particular technique is good at reducing some risks but not others. In a neat and orderly world, there would be a single technique to address every known risk. In practice, some risks can be mitigated by multiple techniques, while others risks require you to invent techniques on the fly.

This frame of mind, where you choose techniques based on risks, helps you to work efficiently. You do not want to waste time (or other resources) on low-impact techniques, nor do you want to ignore project-threatening risks. You want to build

successful systems by taking a path that spends your time most effectively. That means only applying techniques when they are motivated by risks.

Optimal basket of techniques. To avoid wasting your time and money, you should choose techniques that best reduce your prioritized list of risks. You should seek out opportunities to kill two birds with one stone by applying a single technique to mitigate two or more risks. You might like to think of it as an *optimization problem* to choose a set of techniques that optimally mitigates your risks.

It is harder to decide which techniques should be applied than it appears at first glance. Every technique does something valuable, just not the valuable thing your project needs most. For example, there are techniques for improving the usability of your user interfaces. Imagine you successfully used such techniques on your last project, so you choose it again on your current project. You find three usability flaws in your design, and fix them. Does this mean that employing the usability technique was a good idea?

Not necessarily, because such reasoning ignores the *opportunity cost*. The fair comparison is against the other techniques you could have used. If your biggest risk is that your chosen framework is inappropriate, you should spend your time analyzing or prototyping your framework choice instead of on usability. Your time is scarce, so you should choose techniques that are maximally effective at reducing your failure risks, not just somewhat effective.

Cannot eliminate engineering risk. Perhaps you are wondering why we should try to create an optimal basket of techniques when we should go all the way and eliminate engineering risk. It is tempting, since engineers hate ignoring risks, especially those they know how to address.

The downside of trying to eliminate engineering risk is *time*. As aviation pioneers, the Wright brothers spent time on mathematical and empirical investigations into aeronautical principles and thus reduced their engineering risk. But, if they had continued these investigations until risks were eliminated, their first test flight might have been in 1953 instead of 1903.

The reason you cannot afford to eliminate engineering risk is because you must balance it with non-engineering risk, which is predominantly project management risk. Consequently, a software developer does not have the option to apply every useful technique because risk reductions must be balanced against time and cost.

3.5 Guidance on choosing techniques

So far, you have been introduced to the risk-driven model and have been advised to choose techniques based on your risks. You should be wondering how to make good choices. In the future, perhaps a developer choosing techniques will act much like a

mechanical engineer who chooses materials by referencing tables of properties and making quantitative decisions. For now, such tables do not exist. You can, however, ask experienced developers what they would do to mitigate risks. That is, you would choose techniques based on their experience and your own.

However, if you are curious, you would be dissatisfied either with a table or a collection of advice from software veterans. Surely there must be principles that underlie any table or any veteran's experience, principles that explain why technique X works to mitigate risk Y.

Such principles do exist and we will now take a look at some important ones. Here is a brief preview. First, sometimes you have a problem to *find* while other times you have a problem to *prove*, and your technique choice should match that need. Second, some problems can be solved with an *analogic* model while others require an *analytic* model, so you will need to differentiate these kinds of models. Third, it may only be efficient to analyze a problem using a particular type of model. And finally, some techniques have *affinities*, like pounding is suitable for nails and twisting is suitable for screws.

Problems to find and prove. In his book *How to Solve It*, George Polya identifies two distinct kinds of math problems: problems to *find* and problems to *prove* (Polya, 2004). The problem, "Is there a number that when squared equals 4?" is a problem to find, and you can test your proposed answer easily. On the other hand, "Is the set of prime numbers infinite?" is a problem to prove. Finding things tends to be easier than proving things because for proofs you need to demonstrate something is true in all possible cases.

When searching for a technique to address a risk, you can often eliminate many possible techniques because they answer the wrong kind of Polya question. Some risks are specific, so they can be tested with straightforward test cases. It is easy to imagine writing a test case for "Can the database hold names up to 100 characters?" since it is a problem to find. Similarly, you may need to design a scalable website. This is also a problem to find because you only need to design (i.e., find) one solution, not demonstrate that your design is optimal.

Conversely, it is hard to imagine a small set of test cases providing persuasive evidence when you have a problem to prove. Consider, "Does the system always conform to the framework Application Programming Interface (API)?" Your tests could succeed, but there could be a case you have not yet seen, perhaps when a framework call unexpectedly passes a null reference. Another example of a problem to prove is deadlock: Any number of tests can run successfully without revealing a problem in a locking protocol.

Analytic and analogic models. Michael Jackson, crediting Russell Ackoff, distinguishes between *analogic models* and *analytic models* (Jackson, 1995; Jackson, 2000).

In an analogic model, each model element has an analogue in the domain of interest. A radar screen is an analogic model of some terrain, where blips on the screen correspond to airplanes — the blip and the airplane are analogues.

Analogic models support analysis only indirectly, and usually domain knowledge or human reasoning are required. A radar screen can help you answer the question, "Are these planes on a collision course?" but to do so you are using your special purpose brainpower in the same way that an outfielder can tell if he is in position to catch a fly ball (see Section 15.6.1).

An analytic (what Ackoff would call *symbolic*) model, by contrast, directly supports computational analysis. Mathematical equations are examples of analytic models, as are state machines. You could imagine an analytic model of the airplanes where each is represented by a vector. Mathematics provides an analytic capability to relate the vectors, so you could quantitatively answer questions about collision courses.

When you model software, you invariably use symbols, whether they are Unified Modeling Language (UML) elements or some other notation. You must be careful because some of those symbolic models support analytic reasoning while others support analogic reasoning, even when they use the same notation. For example, two different UML models could represent airplanes as classes, one with and one without an attribute for the airplane's vector. The UML model with the vector enables you to compute a collision course, so it is an analytic model. The UML model without the vector does not, so it is an analogic model. So simply using a defined notation, like UML, does not guarantee that your models will be analytic. *Architecture description languages* (ADLs) are more constrained than UML, with the intention of nudging your architecture models to be analytic ones.

Whether a given model is analytic or analogic depends on the question you want it to answer. Either of the UML models could be used to count airplanes, for example, and so could be considered analytic models.

When you know what risks you want to mitigate, you can appropriately choose an analytic or analogic model. For example, if you are concerned that your engineers may not understand the relationships between domain entities, you may build an analogic model in UML and confirm it with domain experts. Conversely, if you need to calculate response time distributions, then you will want an analytic model.

Viewtype matching. The effectiveness of some risk-technique pairings depends on the type of model or view used. *Viewtypes* are not fully discussed until Section 9.6. For now, it is sufficient to know about the three primary viewtypes. The *module viewtype* includes tangible artifacts such as source code and classes; the *runtime viewtype* includes runtime structures like objects; and the *allocation viewtype* includes allocation elements like server rooms and hardware. It is easiest to reason about modifiability from the module viewtype, performance from the runtime viewtype, and security from the deployment and module viewtypes.

Each view reveals selected details of a system. Reasoning about a risk works best when the view being used reveals details relevant to that risk. For example, reasoning about a runtime protocol is easier with a runtime view, perhaps a state machine, than with source code. Similarly, it is easier to reason about single points of failure using an allocation view than a module view.

Despite this, developers are adaptable and will work with the resources they have, and will mentally simulate the other viewtypes. For example, developers usually have access to the source code, so they have become quite adept at imagining the runtime behavior of the code and where it will be deployed. While a developer can make do with source code, reasoning will be easier when the risk and viewtype are matched, and the view reveals details related to the risk.

Techniques with affinities. In the physical world, tools are designed for a purpose: hammers are for pounding nails, screwdrivers are for turning screws, saws are for cutting. You may sometimes hammer a screw, or use a screwdriver as a pry bar, but the results are better when you use the tool that matches the job.

In software architecture, some techniques only go with particular risks because they were designed that way and it is difficult to use them for another purpose. For example, Rate Monotonic Analysis primarily helps with reliability risks, threat modeling primarily helps with security risks, and queuing theory primarily helps with performance risks (these techniques are discussed in Section 15.6).

3.6 When to stop

The beginning of this chapter posed two questions. So far, this chapter has explored the first: Which design and architecture techniques should you use? The answer is to identify risks and choose techniques to combat them. The techniques best suited to one project will not be the ones best suited to another project. But the mindset of aligning your architecture techniques, your experience, and the guidance you have learned will steer you to appropriate techniques.

We now turn our attention to the second question: How much design and architecture should you do? Time spent designing or analyzing is time that could have been spent building, testing, etc., so you want to get the balance right, neither doing too much design, nor ignoring risks that could swamp your project.

Effort should be commensurate with risk. The risk-driven model strives to efficiently apply techniques to reduce risks, which means not over- or under-applying techniques. To achieve efficiency, the risk-driven model uses this guiding principle:

Architecture efforts should be commensurate with the risk of failure.

If you recall the story of my father and the mailbox, he was not terribly worried about the mailbox falling over, so he did not spend much time designing the solution or applying mechanical engineering analyses. He thought about the design a little bit, perhaps considering how deep the hole should be, but most of his time was spent on implementation.

When you are unconcerned about security risks, spend no time on security design. However, when performance is a project-threatening risk, work on it until you are reasonably sure that performance will be OK.

Incomplete architecture designs. When you apply the risk-driven model, you only design the areas where you perceive failure risks. Most of the time, applying a design technique means building a model of some kind, either on paper or a whiteboard. Consequently, your architecture model will likely be detailed in some areas and sketchy, or even non-existent, in others.

For example, if you have identified some performance risks and no security risks, you would build models to address the performance risks, but those models would have no security details in them. Still, not every detail about performance would be modeled and decided. Remember that models are an intermediate product and you can stop working on them once you have become convinced that your architecture is suitable for addressing your risks.

Subjective evaluation. The risk-driven model says to prioritize your risks, apply chosen techniques, then evaluate any remaining risk, which means that you must decide if the risk has been sufficiently mitigated. But what does sufficiently mitigated mean? You have prioritized your risks, but which risks make the cut and which do not?

The risk-driven model is a framework to facilitate your decision making, but it cannot make judgment calls for you. It identifies salient ideas (prioritized risks and corresponding techniques) and guides you to ask the right questions about your design work. By using the risk-driven model, you are ahead because you have identified risks, enacted corresponding techniques, and kept your effort commensurate with your risks. But eventually you must make a subjective evaluation: will the architecture you designed enable you to overcome your failure risks?

3.7 Planned and evolutionary design

You should now be prepared, at a conceptual level at least, to go out and apply software architecture on your projects. You may still have some questions about how to proceed, however, since we have not yet discussed how the risk-driven model interacts with other kinds of guidance you already know, things like planned and evolutionary design, software processes, and specifically agile software development.

The remainder of the chapter shows how the risk-centric model is compatible with each of these and can be used to augment their advice.

We start by discussing three styles of design: planned, evolutionary, and minimal planned design. Planned and evolutionary are the two basic styles of design and minimal planned design is a combination of them.

Evolutionary design. *Evolutionary design* "means that the design of the system grows as the system is implemented" (Fowler, 2004). Historically, evolutionary design has been frowned upon because local and uncoordinated design decisions yield chaos, creating a hodgepodge system that is hard to maintain and evolve any further.

However, recent trends in software processes have re-invigorated evolutionary design by addressing most of its shortcomings. The agile practices of *refactoring*, *test-driven design*, and *continuous integration* work against the chaos. Refactoring (a behavior-preserving transformation of code) cleans up the uncoordinated local designs (Fowler, 1999), test-driven design ensures that changes to the system do not cause it to lose or break existing functionality, and continuous integration provides the entire team with the same codebase. Some argue that these practices are sufficiently powerful that planned design can be avoided entirely (Beck and Andres, 2004).

Of the three practices, refactoring is the workhorse that reduces the hodgepodge in evolutionary design. Refactoring replaces designs that solved older, local problems with designs that solve current, global problems. Refactoring, however, has limits. Current refactoring techniques provide little guidance for architecture scale transformations. For example, Amazon's sweeping change from a tiered, single-database architecture to a service-oriented architecture (Hoff, 2008a) is difficult to imagine resulting from small refactoring steps at the level of individual classes and methods. In addition, legacy code usually lacks sufficient test cases to confidently engage in refactoring, yet most systems have some legacy code.

Though some projects use evolutionary design recklessly, its advocates say that evolutionary design must be paired with supporting practices like refactoring, test-driven design, and continuous integration.

Planned design. At the opposite end of the spectrum from evolutionary design is *planned design*. The general idea behind planned design is that plans are worked out in great detail before construction begins. Analogies with bridge design and construction are often brought up, since bridge construction rarely begins before its design is complete.

Few people advocate[2] doing planned design for an entire software system, an approach sometimes called *Big Design Up Front (BDUF)*. However, complete planning

[2]Model Driven Engineering (MDE) is an exception since it needs a detailed model to generate code.

of just the architecture is suggested by some authors (Lattanze, 2008; Bass, Clements and Kazman, 2003), since it is often hard to know on a large or complex project that *any* system can satisfy the requirements. When you are not sure that any system can be built, it is best to find this out early.

Planned architecture design is also practical when an architecture is shared by many teams working in parallel, and therefore useful to know before the sub-teams start working. In this case, a planned architecture that defines the top-level components and connectors can be paired with *local design*s, where sub-teams design the internal models of the components and connectors. The architecture usually insists on some overall invariants and design decisions, such as setting up a concurrency policy, a standard set of connectors, allocating high-level responsibilities, or defining some localized quality attribute scenarios. Note that architectural modeling elements like components and connectors will be fully described in the second part of this book.

Even when following planned design, an architecture or design should rarely, if ever, be 100% complete before proceeding to prototyping or coding. With current design techniques, it is nearly impossible to perfect the design without feedback from running code.

Minimal planned design. In between evolutionary design and planned design is *minimal planned design*, or *Little Design Up Front* (Martin, 2009). Advocates of minimal planned design worry that they might design themselves into a corner if they did all evolutionary design, but they also worry that all planned design is difficult and likely to get things wrong. Martin Fowler puts estimated numbers on this, saying he does roughly 20% planned design and 80% evolutionary design (Venners, 2002).

Balancing planned and evolutionary design is possible. One way is to do some initial planned design to ensure that the architecture will handle the biggest risks. After this initial planned design, future changes to requirements can often be handled through local design, or with evolutionary design if the project also has refactoring, test-driven-design, and continuous integration practices working smoothly.

If you are concerned primarily with how well the architecture will support global or emergent qualities, you can do planned design to ensure these qualities and reserve any remaining design as evolutionary or local design. For example, if you have identified throughput as your biggest risk, you could engage in planned design to set up throughput budgets (e.g., message deliveries happen in 25ms 90% of the time). The remainder of the design, which ensured that individual components and connectors met those performance budgets, could be done as evolutionary or local design. The general idea is to perform *architecture-focused design* (see Section 2.7) to set up an architecture known to handle your biggest risks, allowing you more freedom in other design decisions.

Which is best? Regardless of which design style you prefer, you must design software before you write the code, whether it is ten minutes before or ten months before. Both design styles have devoted proponents and their debate relies on anecdotes rather than solid data, so for now opinions will vary. If you have high confidence in your ability to do evolutionary design, you will do less planned design.

Realize that different systems will lend themselves to different styles of design. Consider the slow changes to the Apache web server over the past decade. It is suitable for planned design because its design resembles an optimization problem for a stable set of requirements (e.g., high reliability, extensibility, and performance). On the other hand, many projects have rapidly changing requirements that favor evolutionary design.

The essential tension between planned and evolutionary design is this: A long head start on architectural design yields opportunities to ensure global properties, avoid design dead ends, and coordinate sub-teams — but it comes at the expense of possibly making mistakes that would be avoided if decisions were made later. Teams with strong refactoring, test-driven development, and continuous integration practices will be able to do more evolutionary design than other teams.

The risk-driven model is compatible with evolutionary, planned, and minimal planned design. All of these design styles agree that design should happen at some point and they all allocate time for it. In planned design, that time is up-front, so applying the risk-driven model means doing up-front design until architecture risks have subsided. In evolutionary design, it means doing architecture design during development, whenever a risk looms sufficiently large. Applying it to minimal planned design is a combination of the others.

3.8 Software development process

Few developers build systems using only a design style, say evolutionary design, and a compiler. Instead, their activities are structured using a software development process that has been designed to increase their chances of successfully delivering a good system. A good software development process does more than just minimize engineering risks, since it must also factor in other business needs and risks, such as time-to-market pressures.

When you broaden your attention from pure engineering risks to the overall project risks, you find many more risks to worry about. Will the customer accept your system? Will the market have changed by the time you deliver? Will you deliver on time? Did your requirements reflect the customer's desires? Do you have the right people, are they doing the right jobs, and are they communicating effectively? Will there be lawsuits?

Software development process. A *software development process* orchestrates a team's activities with the goal of balancing both engineering and project management risks. It is tempting, but impossible, to cleanly separate engineering process from project management process. A software development process helps you prioritize risks across both engineering and project management, and perhaps to decide that even though engineering risks still exist, other risks outweigh them.

Risk as shared vocabulary. Risks are the shared vocabulary between engineers and project managers. A manager's job is to understand tradeoffs and make decisions across the risks on a project. A manager may not be technical enough to understand why a module does not work as desired, but he will understand the risk of its failure, and the engineer can help him assess the risk's probability and severity.

The concept of a risk is positioned in the common ground between the world of engineering and the world of project management. Engineers may choose to ignore office politics and marketing meetings, and managers may choose to ignore the database schema and performance estimates, but in the idea of risks they find common ground to make decisions about the system.

Baked-in risks. If you had never seen a software development process before, you might imagine it was like a control loop in a program, where during each iteration it prioritizes the risks and plans out the next step accordingly, looping until the system is delivered. In practice, some risk mitigation steps are deliberately *baked-in* to the software development process.

At a large company worried about team coordination, the process might insist on various forms of documentation at project milestones. Agile processes bake-in worries about time-to-market and customer rejecting the product, and consequently insist that the software be built and delivered in short iterations. IT-specific processes often face risks associated with unknown and complex domains, so their processes may bake-in constructing domain models. Whenever I leave the house, I pat my pockets to ensure that I have my wallet and keys because it is enough of a risk to bake-in to my habits.

Baking risk mitigation techniques into the software development process can be a blessing. It is a blessing when the process bakes-in risks that you would prioritize anyway, so it saves you the time of every day deciding that, for example, you should stick to two-week iterations rather than slipping the schedule. It is an efficient means of conveying expertise from experienced software developers, because they can point to successful results of following a process, rather than explaining their philosophy on software development that was baked-in. In an agile method such as XP, a team following the process can succeed even if they do not understand why XP chose its particular set of techniques.

Baking risks into the software development process can be a curse when you get

it wrong. Many years ago, I interviewed with a tiny startup company. The project manager, formerly with $BIGCOMPANY, asked me what I thought about process and I told him that it needed to be appropriate for the project, the domain, and the team. Above all else, I said, applying a process from a book, unaltered, was unlikely to work. Like a scene from a comedy, he swiveled in his chair and picked up a book describing $BIGCOMPANY's development process and said, "This is the process we will be following." Needless to say, I did not end up working there, but I wish I could have seen the five co-located engineers producing detailed design documents and other bureaucracy that are baked-in to processes for large, distributed teams.

If you decide to tailor your software development process to bake-in risks, some important features to consider include project complexity (big, small), team size (big, small), location (distributed, co-located), domain (IT, finance, systems, embedded, safety-critical, etc.), and kind of customer (internal, external, shrink-wrapped).

3.9 Understanding process variations

Before you can see how to apply the risk-driven model to a software development process, you will need to know about the broad categories of processes and some details about them. This section offers an overview that omits details of each process, but it provides adequate background so that you can think about how to apply the risk-driven model.

This overview fits each process into a simple two-part template: An optional up-front design part with one or more iterations that follow. Not every development process here has up-front design, but all of them have at least one iteration. The template varies on four points:

1. Is there up-front design?
2. What is the nature of the design (planned/evolutionary; redesign allowed)?
3. How is work prioritized across iterations?
4. How long is an iteration?

Figure 3.4 summarizes the processes and highlights some of their differences.

Two other important variation points that arise when talking about development process are: how detailed should your design models be, and how long you should hold on to your design models? None of the above processes commits to an answer for these, except for XP, which allows modeling but discourages keeping the models around past an iteration. Applying this simple template to software development processes yields the following descriptions:

Process	Up-front design	Nature of design	Prioritization of work	Iteration length
Waterfall	In analysis & design phases	Planned design; no redesign	Open	Open
Iterative	Optional	Planned or evolutionary; redesign allowed	Open, often feature-centric	Open, usually 1-8 weeks
Spiral	None	Planned or evolutionary	Riskiest work first	Open
UP / RUP	Optional; design activities front-loaded	Planned or evolutionary	Riskiest work first, then highest value	Usually 2-6 weeks
XP	None, but some do in iteration zero	Evolutionary design	Highest customer value first	Usually 2-6 weeks

Figure 3.4: Examples of software development processes and how they treat design issues. For comparison purposes, a waterfall process is treated as having a single long iteration.

Waterfall. The waterfall process proceeds from beginning to end as a single long block of work that delivers the entire project (Royce, 1970). It assumes planned design work that is done in its analysis and design phases. These precede the construction phase, which can be considered a single iteration. With just one iteration, work cannot be prioritized across iterations, but it may be built incrementally within the construction phase. Applying the risk-driven model would mean doing architecture work primarily during the analysis and design phases.

Iterative. An iterative development process builds the system in multiple work blocks, called iterations (Larman and Basili, 2003). With each iteration, developers are allowed to rework existing parts of the system, so it is not just built incrementally. Iterative development optionally has up-front design work but it does not impose a prioritization across the iterations, nor does it give guidance on the nature of design work. Applying the risk-driven model would mean doing architecture work within each iteration and during the optional up-front design.

Spiral. The spiral process is a kind of iterative development, so it has many iterations, yet it is often described as having no up-front design work (Boehm, 1988). Iterations are prioritized by risk, with the first iteration handling the riskiest parts of a project. The spiral model handles both management and engineering risks. For

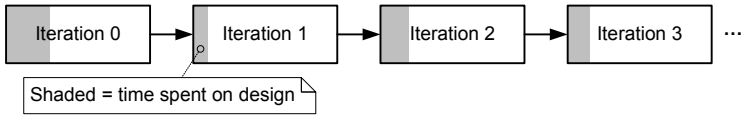

Figure 3.5: An example of how the amount of design could vary across iterations based on your perception of the risks. Based on the amount of time spent, you can infer that the most risk was perceived in iteration 0 and iteration 2.

example, it may address "personnel shortfalls" as a risk. The spiral process gives no guidance about how much architecture/design work to do, or about which architecture and design techniques to use.

[Rational] Unified Process (RUP). The Unified Process and its specialization, the Rational Unified Process, are iterative, spiral processes (Jacobson, Booch and Rumbaugh, 1999; Kruchten, 2003). They highlight the importance of addressing risks early and the use of architecture to address risks. The (R)UP advocates working on architecturally-relevant requirements first, in early iterations. It can accommodate either planned or evolutionary design.

Extreme Programming (XP). Extreme Programming is a specialization of an iterative and agile software development process, so it contains multiple iterations (Beck and Andres, 2004). It suggests avoiding up-front design work, though some projects add an *iteration zero* (Schuh, 2004), in which no customer-visible functionality is delivered. It guides developers to apply evolutionary design exclusively, though some projects modify it to incorporate a small amount of up-front design. Each iteration is prioritized by the customer's valuation of features, not risks.

3.10 The risk-driven model and software processes

It is possible to apply the risk-driven model to any of these software development processes while still keeping within the spirit of each. The waterfall process prescribes planned design in its analysis and design phases, but does not tell you what kind of architecture and design work to do, or how much of it. You can apply the risk-driven model during the analysis and design phases to answer those questions.

The iterative process does not have a designated place for design work, but it could be done at the beginning of each iteration. The amount of time spent on design would vary based on the risks. Figure 3.5 provides a notional example of how the amount of design could vary across iterations based on your perception of the risks.

The spiral process and the risk-driven model are cousins in that risk is primary in both. The difference is that the spiral process, being a full software development process, prioritizes both management and engineering risks and guides what hap-

pens across iterations. The risk-driven model only guides design work to mitigate engineering risks, and only within an iteration. Applying the risk-driven model to the spiral model or the (R)UP works the same as with an iterative process.

You will have noticed that, of the processes listed in Figure 3.4, XP (an agile process) has the most specific advice. Consequently, it is trickiest to apply the risk-driven model into the XP process (or other feature-centric agile processes), so we will look at that process in more depth.

3.11 Application to an agile processes

The following description of using the risk-driven model on an agile project highlights some core issues, such as when to design, and how to mix risks into a feature-driven development process. Since agile projects vary in their process, this description assumes one with a two-week iteration that plays a *planning game* to manage a *feature backlog*. On the engineering side, there are software architecture risks that you should fold into this process, which includes identification, prioritization, mitigation, and evaluation of those risks. The big challenges are: first, how to address initial engineering risks, and second, how to incorporate engineering risks that you later discover into the stack of work to do.

Risks. You will have identified some risks at the beginning of the project, such as the initial choices for architectural style, choice of frameworks, and choice of other COTS (Commercial Off-The-Shelf) components. Some agile projects use an iteration zero to get their development environment set up, including source code control and automated build tools. You can piggyback here to start mitigating the identified risks. Developers could have a simple whiteboard meeting to ensure everyone agrees on an architectural style, or come up with a short list of styles to investigate. If performance characteristics of COTS components are unknown but important, some quick prototyping can be done to provide approximate speed or throughput numbers.

Risk backlog. At the end of an iteration, you need to evaluate how well your activities mitigated your risks. Most of the time you will have reduced a risk sufficiently that it drops off your radar, but sometimes not. Imagine that at the end of the iteration you have learned that prototyping shows that your preferred database will run too slowly. This risk can be written up as a testable feature for the system. This is the beginning of a *risk backlog*. Whenever possible, risks should be written up as testable items.

Some risks are small enough that they can be handled as they arise during an iteration, and never show up on the backlog. But larger risks will need to be scheduled just like features are.

Note that this is not an excuse to turn a nominal iteration zero into a de facto Big Design Up-Front exercise. Instead of extending the time of iteration zero, risks are

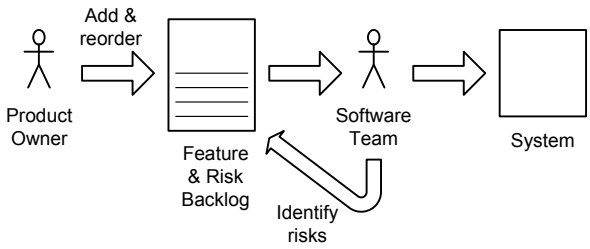

Figure 3.6: One way to incorporate risk into an agile process is to convert the feature backlog into a *feature & risk backlog*. The product owner adds features and the software team adds technical risks. The software team must help the product owner to understand the technical risks and suitably prioritize the backlog.

pushed onto the backlog. This raises a question: how can we handle both backlogged features and risks?

Prioritizing risks and features. Many agile projects divide the world into product owners, who create a prioritized list of features called the backlog, and developers, who take features from the top of the backlog and build them.

It is tempting to put both features and risks on the same backlog, but managing the backlog becomes more complex once you introduce risks, because both features and risks must be prioritized together. Who is qualified to prioritize both?

If you give the product owner the additional responsibility to prioritize architectural risks alongside features, you can simply change the feature backlog into a feature & risk backlog, as seen in Figure 3.6. Software developers may see a feature low in the backlog asking for security. It is their job to educate the product owners that if they ever want to have a secure application, they need to address that risk early, since it will be difficult or impossible to add later. As part of the reflection at the end of each iteration, you should evaluate architectural risks and feed them into the backlog.

Summary. An agile process can handle architectural risks by doing three things. Architectural risks that you know in advance can be (at least partially) handled in a time-boxed iteration zero, where no features are planned to be delivered. Small architectural risks can be handled as they arise during iterations. And large architectural risks should be promoted to be on par with features, and inserted into a combined feature & risk backlog.

3.12 Risk and architecture refactoring

Over time, system developers understand increasingly well how a system should be designed. This is true regardless of which kind of process they follow (e.g., waterfall or iterative processes). In the beginning, they know and understand less. After some work (design, prototyping, iterations, etc.) they have better grounded opinions on suitable designs.

Once they recognize that their code does not represent the best design (e.g., by detecting *code smells*), they have two choices. One is to ignore the divergence, which yields *technical debt*. If allowed to accumulate, the system will become a big ball of mud (see Section 14.7). The other is to refactor the code, which keeps it maintainable. This second option is well described by Brian Foote and William Opdyke in their patterns on software lifecycle (Coplien and Schmidt, 1995).

Refactoring, by definition, means re-design and the scale of that redesign can vary. Sometimes a refactoring involves just a handful of objects or some localized code. But other times it involves more sweeping architectural changes and is called *architecture refactoring*. Since little published guidance exists for refactoring at large scale, architecture refactoring is generally performed ad hoc.

The example from the introduction where Rackspace implemented their query system three different ways (see Section 1.2) is best thought of as architecture refactoring. There, each refactoring of the architecture was precipitated by a pressing failure risk. Object-level refactorings take a negligible amount of time and therefore need little justification, so you should just go ahead and, for example, rename a variable to be more expressive of its intent. An architecture refactoring is expensive, so it requires a significant risk to justify it.

Two important lessons are apparent. First, *design does not exclusively happen up-front*. It is often reasonable to spend time up-front making the best choices you can, but it is optimistic to think you know enough to get all those design decisions right. You should anticipate spending time designing after your project's inception.

Second, *failure risk can guide architecture refactoring*. By the time it is implemented, nearly every system is out of date compared to the best thinking of its developers. That is, some technical debt exists. Perhaps, in hindsight, you wish you had chosen a different architecture. Risks can help you decide how bad it will be if you keep your current architecture.

3.13 Alternatives to the risk-driven model

The risk-driven model does two things: it helps you decide when you can stop doing architecture, and it guides you to appropriate architecture activities. It is not good at predicting how long you will spend designing, but it helps you recognize when you

have done enough. There are several alternatives to the risk-driven model, with their own advantages and disadvantages.

No design. The option of not designing is a bit of a misnomer, especially if you believe that every system has an architecture, because the developers must have thought about it at some point. Perhaps they were thinking about the design (i.e., what they will code) immediately before they start typing, but they do think about the design. Such projects likely borrow heavily from presumptive architectures (see Section 2.4), where the developers pattern their system off of similar successful systems, explicitly or implicitly.

Documentation package. Some people suggest, or at least imply, that you should build a full documentation package that describes your architecture. If you follow this guidance, you will build a set of models and diagrams and write them down in such a way that someone else could read and understand the architecture, which can be quite desirable. If you need documentation, the Documenting Software Architectures book (Clements et al., 2010) will guide you to an effective set of models and diagrams to record.

However, few projects will need to create a full documentation package, and the "3 guys in a garage" startup probably cannot afford to write anything down.

Yardsticks. Empirical data can help you decide how much time should be spent on architecture and design. Barry Boehm has calculated the optimal amount of time to spend on the architecture for small, medium, and large projects based on a variant of his COCOMO model (Boehm and Turner, 2003). For various project sizes, he has plotted curves of architecture effort vs. total project duration. His data indicates that most projects should spend 33-37% of their total time doing architecture, with small projects spending as little as 5% and very large projects spending 40%.

A yardstick like "spend 33% of your time on architecture" can be used by project managers for planning project activities and staffing requirements, yielding a time budget to spend in design.

Yardsticks, however, are little help to developers once the architecture work has started. No reasonable developer should continue design activities for additional days after the risks have been worked out, even if the yardstick provides that budget. Nor should a reasonable developer switch to coding when a major failure risk is outstanding.

It is best to view such yardsticks as heuristics derived from experience combating risks, where projects of a certain size historically needed about that much time to mitigate their risks. That yardstick does not help you decide whether one more (or one less) day of architecture work is appropriate. Also, yardsticks only suggest broad categorical activities rather than guide you to particular techniques.

Ad hoc. When choosing how much architecture to do, most developers probably do not follow any of the alternatives above. Instead, they make a decision in the moment, based on their experience and their best understanding of the project's needs.

This may indeed be the most effective way to proceed, but it is dependent upon the skill and experience of the developer. It is not teachable, since its lessons are not explicit, nor is it particularly helpful in creating project planning estimates. It may be that, in practice, the ad hoc approach is a kind of informal risk-driven model, where developers tacitly weigh the risks and choose appropriate techniques.

3.14 Conclusion

This chapter set out to investigate two questions. First, *which design and architecture techniques should developers use?* And second, *how much design and architecture should developers do?* It reviewed existing answers, including doing no design, using yardsticks, building documentation packages, and proceeding ad hoc. It introduced the *risk-driven model* that encourages developers to: (1) prioritize the risks they face, (2) choose appropriate architecture techniques to mitigate those risks, and (3) re-evaluate remaining risks. It encourages just enough design and architecture by guiding developers to a prioritized subset of architecture activities. Design can happen up-front but it also happens during a project.

The risk-driven model is inspired by my father's work on his mailbox. He did not perform complex calculations — he just stuck the post in the hole then filled it with concrete. Low-risk projects can succeed without any planned architecture work, while many high-risk ones would fail without it.

The risk-driven model walks a middle path that avoids the extremes of complete architecture documentation packages and architecture avoidance. It follows the principle that your architecture efforts should be commensurate with the risk of failure. Avoiding failure is central to all engineering and you can use architecture techniques to mitigate the risks. The key element of the risk-driven model is the promotion of risk to prominence. Each project will have a different set of risks, so it likely needs a different set of techniques. To avoid wasting your time and money, you should choose techniques that best reduce your prioritized list of risks.

The question of how much software architecture work you should do has been a thorny one for a long time. The risk-driven model transforms that broad question into a narrow one: "Have your chosen techniques sufficiently reduced your failure risks?" Evaluation of risk mitigation is still subjective, but it is one that developers can have a focused conversation about.

Engineering techniques address engineering risks, but projects face a wide variety of risks. Software development processes must prioritize both management risks and engineering risks. You cannot reduce engineering risks to zero because there

are also project management risks to consider, including time-to-market pressure. By applying the risk-driven model, you ensure that whatever time you devote to software architecture reduces your highest priority engineering risks and applies relevant techniques.

Agile software development approaches often emphasize evolutionary design over planned design. A middle path, minimal planned design, can be used to avoid the extremes. The essential tension is this: A long head start on architectural design yields opportunities to ensure global properties, avoid design dead ends, and coordinate sub-teams — but it comes at the expense of possibly making mistakes that would be avoided if decisions were made later. Agile processes focusing on features can be adapted slightly to add risk to the feature backlog, with developers educating product owners on how to prioritize the feature & risk backlog.

Some readers may be frustrated that this chapter does not prescribe a list of techniques to use and a single process to follow. These are missing because the techniques that work great on one project would be inappropriate on another. And there is not yet enough data to overcome opinions about the best process to recommend. Indeed, you may not have a choice about which process you follow, but within that process you likely have the ability to use the risk-driven model. This chapter has tried to provide relevant information about how to make your own choices so that you can do just enough architecture for your projects.

3.15 Further reading

The invention of risk as a concept likely occurred quite early, with references to it in Greek antiquity, but it took on its modern, more general, idea as late as the 17th century, where it increasingly displaced the concept of *fortunes* as what drove life's outcomes (Luhmann, 1996). A few minutes after that, project managers started using risk to drive their projects. This longstanding tradition in project management has carried over into software process design, with many authors emphasizing the role of risk in software development, including Philippe Kruchten (Kruchten, 2003), Ivar Jacobson, Grady Booch, and James Rumbaugh (Jacobson, Booch and Rumbaugh, 1999), and specifically noting the connection between architecture and risk.

Barry Boehm wrote about risk in the context of software development with his paper on the spiral model of software development (Boehm, 1988), which is an interesting read even if you already understand the model. The risk-driven model would, on first glance, appear to be quite similar to the spiral model of software development, but the spiral model applies to the entire development process, not just the design activity. A single turn through the spiral has a team analyzing, designing, developing, and testing software. The full spiral covers the project from inception to deployment. The risk-driven model, however, applies just to design, and can be incorporated into

nearly any software development process. Furthermore, the spiral model guides a team to build the riskiest parts first, but does not guide them to specific design activities. Both the spiral model and the risk-driven model are in strong agreement in their promotion of risk to a position of prominence.

Barry Boehm and Richard Turner followed this up with a book on risk and agile processes (Boehm and Turner, 2003). The summary of their judgment is, "The essence of using risk to balance agility and discipline is to apply one simple question to nearly every facet of process within a project: Is it riskier for me to apply (more of) this process component or to refrain from applying it?"

Mark Denne and Jane Cleland-Huang discuss both architecture and risk in the context of software project management (Denne and Cleland-Huang, 2003). They advocate managing projects by chunking development into Minimum Marketable Features, which has the consequence of incrementally constructing your architecture.

The risk-driven model is similar to *global analysis* as described by Christine Hofmeister, Robert Nord, and Dilip Soni (Hofmeister, Nord and Soni, 2000). Global analysis consists of two steps: (1) analyzing organizational, technical, and product factors; and (2) developing strategies. Factors and strategies in global analysis map to risks and activities in the risk-driven model. Factors are broader than the technical risks in the risk-driven model, and could include, for example, headcount concerns. Both global analysis and the risk-driven model are similar in that they externalize a structured thought process of the form: I am doing X because Y might cause problems. In the published descriptions, the intention of global analysis is not to optimize the amount of effort spent on architecture, but rather to ensure that all factors have been investigated.

Two publications from the SEI can help you become more consistent and thorough in your identification and explanation of risks. Carr et al. (1993) describe a taxonomy-based method for identifying risks and Gluch (1994) introduces the condition-transition-consequence format for describing risks.

The risk-driven model advocates building limited architecture models that have detail only where you perceive risks. Similarly, authors have been advocating building minimally sufficient models for years, including Desmond D'Souza, Alan Wills, and Scott Ambler (D'Souza and Wills, 1998; Ambler, 2002). Tailoring the models built on a project to the nature of the project (greenfield, brownfield, coordination, enhancement) is discussed in Fairbanks, Bierhoff and D'Souza (2006).

The idea of cataloging techniques, or tactics, is described in the context of Attribute Driven Design in Bass, Clements and Kazman (2003). Attribute Driven Design (ADD) relies on a mapping from quality attributes to tactics (discussed in Section 11.3.4), much like global analysis. The concept in this book of mapping development techniques is similar in nature. ADD guides developers to an appropriate design (a pattern), while the risk-driven model guides developers to an activity, such as perfor-

mance modeling or domain analysis. The risk-driven model can be seen as taking the promotion of risk from the spiral model and adapting the tabular mapping of ADD to map risks to techniques.

Knowing what tactics or techniques to apply would be valuable knowledge to include in a software architecture handbook, and would accelerate the learning of novice developers. Such knowledge is already in the heads of *virtuosos*, as described by Mary Shaw and David Garlan (Shaw and Garlan, 1996). The better our field encodes this knowledge, the more compact it becomes and the faster the next generation of developers absorbs it and sees farther.

Though tactics and techniques were described in this chapter as tables, they could be expressed as a pattern language, as originally described by Christopher Alexander for the domain of buildings (Alexander, 1979; Alexander, 1977), and later adapted to software in the Design Patterns book (Gamma et al., 1995) by Erich Gamma and others.

Martin Fowler's essay, "Is Design Dead?" (Fowler, 2004) provides a very readable introduction to evolutionary design and the agile practices that are required to make it work.

Merging risk-based software development and agile processes is an open research area. Jaana Nyfjord's thesis (Nyfjord, 2008) proposes the creation of a Risk Management Forum to prioritize risk across products and projects in an organization. Since the goal here is to handle architecture risks that are only a subset of all project risks, a smaller change to the process may work.

This book uses risk to help you decide which techniques to use and how many of them to apply, assuming the requirements are not negotiable. Another way to use it is to help determine the scope of the projects, assuming the requirements can be changed. Such a quantitative technique is described in Feather and Hicks (2006), with the result being a bag of requirements that gives you the most benefit for the risk that you take on.

With many developers seeking lighter weight processes, agile development is popular. Ambler (2009) provides an overview of how architecture can be woven into agile processes, and Fowler (2004) discusses how evolutionary design can complement planned design. Boehm and Turner (2003) discuss the tension between moving fast and getting it right. A thorough treatment of a practical process for software architecture is found in (Eeles and Cripps, 2009).

Chapter 7

Conceptual Model of Software Architecture

In this book's introduction, you read a story about a coach and a rookie watching the same game. They both saw the same things happening on the field, but despite the rookie's eyes being younger and sharper, the coach was better at understanding and evaluating the action. As a software developer, you would like to understand and evaluate software as effectively as the coach understands the game. This and subsequent chapters will help you build up a mental representation of how software architecture works so that when you see software you will understand it better and will design it better.

The idea of using models, however, is often wrongly conflated with the choice of software process (i.e., waterfall) and has been associated with analysis paralysis. This book is not advocating building lots of written models (i.e., documentation) up front, so it is best to knock down a few strawmen arguments or misunderstandings:

- **Every project should document its architecture: False**. You should make plans before going on a road trip, but do you plan your commute to work in the morning? Models help you solve problems and mitigate risks, but while some problems are best solved with models, others can be solved directly.

- **Architecture documents should be comprehensive: False**. You may decide to build a broad architecture document, or even a comprehensive one, but only in some circumstances — perhaps to communicate a design with others. Most

often you can model just the parts that relate to your risks, so a project with scalability risks would build a narrow model focusing on scalability.

- **Design should always precede coding: False**. In one sense this is true, because code does not flow from your fingers until you have thought about what you will build. But it is false to believe that a design phase (in the software process sense) must precede coding. In fact, early coding may help you discover the hardest problems.

So you should set these strawmen ideas aside. The real reason to use software architecture models is because they help you perform like the coach, not the rookie. If you are not already at the coach level, you want to get there as soon as possible. The standard architecture models represent a condensed body of knowledge that enables you to efficiently learn about software architecture and design. Afterwards, you will find that having a standard model frees your mind to focus on the problem at hand rather than on inventing an new kind of model for each problem.

Conceptual models accelerate learning. If you want to become as effective as a coach, you could simply work on software and wait until you are old. Eventually, all software developers learn something about architecture, even if they sneak up on that knowledge indirectly. It just takes practice, practice, practice at building systems. There are several problems with that approach, however. First, not all old software developers are the most effective ones. Second, the approach takes decades. And third, your understanding of architecture will be idiosyncratic, so you will have a hard time communicating with others, and vice versa.

Consider another path, one where you see farther by standing on the shoulders of others. Perhaps we are still waiting for the Isaac Newton of software engineering, but there is plenty to learn from those who have built software before us. Not only have they given us tangible things like compilers and databases, they have given us a set of abstractions for thinking about programs. Some of these abstractions have been built into our programming languages — functions, classes, modules, etc. Others likely will be, such as components, ports, and connectors[1].

Some people are born brilliant, but for those of us who are not, how effective is standing on the shoulders of those who came before us? Consider this: you are probably a better mathematician than all but a handful of the people in the 17th century. Then, as now, math virtuosos had talent and practiced hard, but today you have the benefit of centuries of compacted understanding. By the time you leave high school, you solve math problems that required a virtuoso a few hundred years ago. And before that, the virtuosos of the 17th century had the benefit of someone else inventing the positional number system and the concept of zero. As you consider

[1] Research languages like ArchJava have already added these concepts to Java.

the two paths, remember that you can and should do both: learn the condensed understanding of architecture and then practice, practice, practice.

Conceptual models free the mind. A condensed understanding can take the form of a conceptual model. The coach's conceptual model includes things like offense and defense strategies, positions, and plays. When he watches the movement of players on the field, he is categorizing what he sees according to his conceptual model. He sees the motion of a player as more than that — it is an element of a play, which is part of a strategy. The rookie, with his limited conceptual model, sees less of this.

Conceptual models accelerate progress in many fields. If you ever took physics, you may have forgotten most of the equations you learned, but you will still conceive of forces acting on bodies. Your physics teacher's lessons were designed to instill that conceptual model. Similarly, if you have ever studied design patterns, you cannot help but recognize those patterns in programs you encounter.

A conceptual model can save you time through faster recognition and consistency, and amplify your reasoning. Alfred Whitehead, said "By relieving the brain of all unnecessary work, a good notation sets it free to concentrate on more advanced problems, and in effect increases the mental power of the race." (Whitehead, 1911) This applies equally to conceptual models. As mentioned in the introduction, Alan Kay has observed that a "point of view is worth 80 IQ points," continuing to say that the primary reason we are better engineers than in Roman times is because we have better problem representations (Kay, 1989).

There is a general consensus on the essential elements and techniques for architecture modeling, though different authors emphasize different parts. For example, the Software Engineering Institute (SEI) emphasizes techniques for quality attribute modeling (Bass, Clements and Kazman, 2003; Clements et al., 2010). The Unified Modeling Language (UML) camp emphasizes techniques for functional modeling (D'Souza and Wills, 1998; Cheesman and Daniels, 2000). The conceptual model in this book integrates both quality attribute and functional models.

Chapter goals and organization. The goal of this part of the book is to provide you with a conceptual model of software architecture, one that enables you to quickly make sense of the software you see and reason about the software you design. The conceptual model includes a set of abstractions, standard ways of organizing models, and know-how. You will never become good at anything without talent and practice, but you can accelerate your progress by building up a mental conceptual model.

This chapter shows you how to partition your architecture into three primary models: the domain, design, and code. It relates these models using designation and refinement relationships. Within each model, details are shown using views. The three chapters that follow this one examine the domain, design, and code models in more detail. An example system for a website called Yinzer runs throughout. A *Yinzer* is a

slang term for someone from Pittsburgh, home of Carnegie Mellon University, and is derived from *yinz*, which is Pittsburgh dialect equivalent to *y'all*.

> Yinzer offers its members online business social networking and job advertisement services in the Pittsburgh area. Members can add other members as business contacts, post advertisements for jobs, recommend a contact for a job, and receive email notifications about matching jobs.

Subsequent chapters cover other details on modeling and give advice about how to use models effectively.

7.1 Canonical model structure

Once you start building models, there are lots of bits and pieces to keep track of. If you see a UML class diagram for the Yinzer system that shows a Job Advertisement associated with a Company, you want to know what it represents: is it things from the real world, your design, or perhaps even your database schema? You need an organization that helps you sort those bits into the right places and to make sense of the whole thing.

The *canonical model structure* presented here provides you with a standard way to organize and relate the facts you encounter and the models you build. You will not always build models that cover the whole canonical model structure, but most projects over time will have bits and pieces of models that follow the canonical structure.

7.1.1 Overview

The essence of the *canonical model structure* is simple: Its models range from abstract to concrete, and it uses views to drill down into the details of each model.

There are three primary models: the domain model, the design model, and the code model, as seen in Figure 7.1. The canonical model structure has the most abstract model (the domain) at the top and the most concrete (the code) at the bottom. The *designation* and *refinement* relationships ensure that the models correspond, yet enable them to differ in their level of abstraction.

Each of the three primary models (the domain, design, and code models) are like databases in that they are comprehensive, but are usually too large and detailed to work on directly. (More on this shortly, in Section 7.4). *Views* allow you to select just a subset of the details from a model. For example, you can select just the details about a single component or just the dependencies between modules. You have no doubt worked with views before, such as a data dictionary or a system context diagram. Views allow you to relate these lists and diagrams back to the canonical model structure. Organizing the models in the canonical structure aids categorization and simplification.

The canonical model structure categorizes different kinds of facts into different models. Facts about the domain, design, and code go into their own models. When you encounter a domain fact like "billing cycles are 30 days," a design fact like "font resources must always be explicitly de-allocated," or an implementation fact like "the customer address is stored in a varchar(80) field," it is easy to sort these details into an existing mental model.

The canonical model structure shrinks the size of each problem. When you want to reason about a domain problem you are undistracted by code details, and vice versa, which makes each easier to reason about.

Let's first take a look at the domain, design, and code models before turning our attention to the relationships between them.

7.2 Domain, design, and code models

The *domain model* describes enduring truths about the domain; the *design model* describes the system you will build; and the *code model* describes the system source code. If something is "just true" then it probably goes in the domain model; if something is a design decision or a mechanism you design then it probably goes in the design model; and if something is written in a programming language, or is a model at that same level of abstraction, then it goes in the code model. Figure 7.1 shows the three models graphically and summarizes the contents of each.

Domain model. The domain model expresses enduring truths about the world that are relevant to your system. For the Yinzer system, some relevant truths would include definitions of important types like Ads and Contacts, relationships between those types, and behaviors that describe how the types and relationships change over time. In general, the domain is not under your control, so you cannot decide that weeks have six days or that you have a birthday party every week.

Design model. In contrast, the design is largely under your control. The system to be built does not appear in the domain model, but it makes its appearance in the design model. The design model is a partial set of design commitments. That is, you leave undecided some (usually low-level) details about how the design will work, deferring them until the code model.

The design model is composed of recursively nested *boundary models* and *internals models*. A boundary model and an internals model describe the same thing (like a component or a module), but the boundary model only mentions the publicly visible interface, while the internals model also describes the internal design.

Code model. The code model is either the source code implementation of the system or a model that is equivalent. It could be the actual Java code or the result of

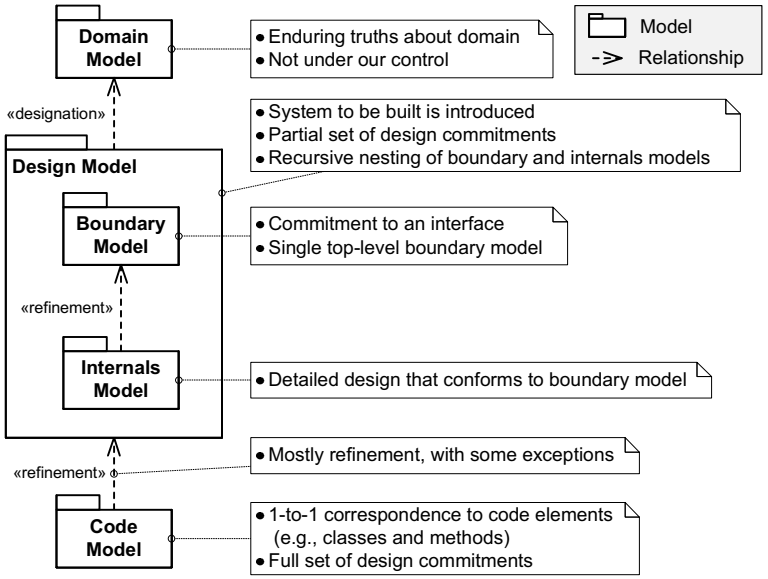

Figure 7.1: The canonical model structure that organizes the domain, design, and code models. The design model contains a top-level boundary model and recursively nested internals models.

running a code-to-UML tool, but its important feature is that has a full set of design commitments.

Design models often omit descriptions of low-risk parts knowing that the design is sufficient so long as the developer understands the overall design and architecture. But where the design model has an incomplete set of design commitments, the code model has a complete set, or at least a sufficiently complete set to execute on a machine.

7.3 Designation and refinement relationships

You no doubt have an intuitive sense of how the domain relates to the design and how the design relates to the code. Because this chapter seeks to divide up models and relate them, it is a good idea to examine these relationships carefully so that you can fully understand them.

Designation. The *designation* relationship enables you to say that similar things in different models should correspond. Using the Yinzer example, the domain model describes domain truths, such as people building a network of contacts and companies posting ads. Using the designation relationship, these truths carry over into the design, as seen in Figure 7.2.

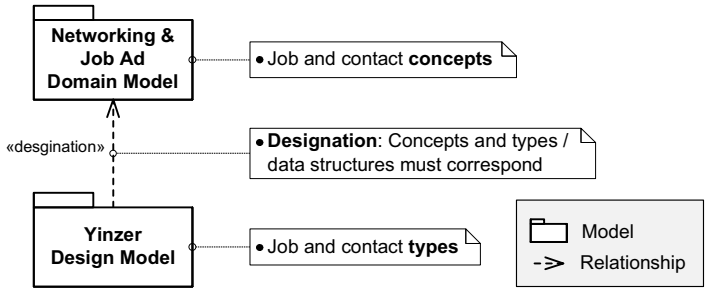

Figure 7.2: The designation relationship ensures that types you choose from the domain correspond to types or data structures in your design.

You have leeway in your design but it should not violate domain truths. You can designate that selected types from the domain must correspond to types and data structures from the design. Things that you do not designate are unconstrained.

While in practice the designation relationship is rarely written down precisely, it would be a mapping that defined the correspondence between the domain elements (e.g., Advertisement and Job types) and the design elements (e.g., Advertisement and Job types and data structures).

Perhaps surprisingly, the design is rarely 100% consistent with the domain because systems often use a simplified or constrained version the domain types. For example, the system may not realize that the same person reads email at two different email addresses, and so might consider them two different people. Or the system may restrict domain types, such as limiting the number of contacts a person can have in the system. But when correspondence with the domain is broken, bugs often follow. The designation relationship is covered in more detail in Section 13.6.

Refinement. *Refinement* is a relationship between a low-detail and a high-detail model of the same thing. It is used to relate a boundary model with an internals model, since they are both models of the same thing, but vary in the details that they expose. Refinement is useful because it lets you decompose your design into smaller pieces. Perhaps the Yinzer system is made up of a client and a server piece, and the server is made up of several smaller pieces. Refinement can be used to assemble these parts into a whole, and vice versa. The mechanics of refinement are discussed in depth in Section 13.7.

Refinement is also used to relate the design model with the code model, but there it is not so straightforward. The structural elements in the design model map neatly to the structural elements in the code model. For example, a module in the design maps to packages in the code, and a component in the design maps to a set of classes in the code.

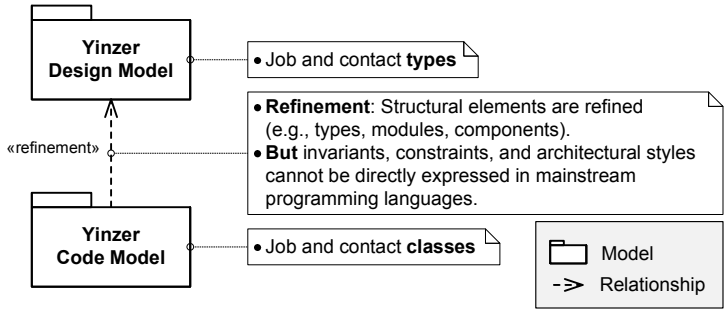

Figure 7.3: The refinement relationship ensures that types you choose from the domain correspond to types or data structures in your design. Be aware that there are elements in the design (invariants, constraints, styles) that cannot be expressed in programming languages.

However, as shown in Figure 7.3, other parts of the design model are absent in the code model: invariants, constraints, and architectural styles. Essentially no mainstream programming languages can directly express the constraints from the design model. It is true that constraints such as "all web requests must complete within 1 second," or "adhere to the pipe-and-filter style" can be *respected* by the code but they cannot be directly *expressed*. This gap between design and code models is discussed in more depth in Section 10.1.

7.4 Views of a master model

In your head, you understand how any number of systems work and carry around models that describe them, such as models of your neighborhood or how you manage your household. From time to time, you sketch out excerpts of those models, such as a map for a friend showing him how to get to that great restaurant, or you write down a list of groceries. These excerpts are consistent with that comprehensive model from your head. For example, you could have written out a full map for your friend, but presumably the one you drew is accurate so far as it goes, and is sufficient to get him there. And your grocery list represents the difference between your eating plans and the contents of your refrigerator.

The domain, design, and code models are comprehensive models like these. They are jam-packed full of details since, conceptually at least, they contain everything that you know about those topics. It would be difficult or impossible to write down all those details, and even keeping them straight in your head is difficult. So, if you want to use a model to reason about security, scalability, or any other reason, you need to winnow down the details so that you can see the relevant factors clearly. This is done with *views*.

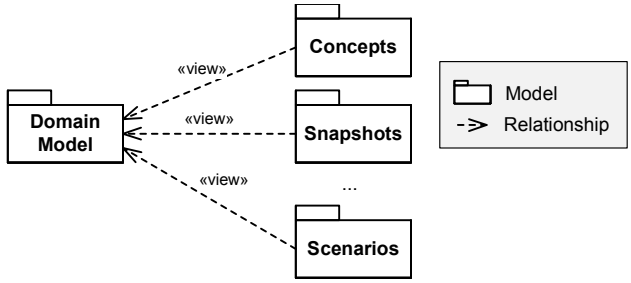

Figure 7.4: The domain model acts as a master model containing all details. Views show selected details from the master model. Because they are all views of the same master model, all of the views are consistent with each other.

Definition. A view, also called a *projection*, shows a defined subset of a model's details, possibly with a transformation. The domain, design, and code models each have many standard views. Views of the domain model include lists of types, lists of relationships between them, and scenarios that show how the types and relationships change over time (see Figure 7.4). Design views include the system context diagram and the deployment diagram. You can invent new views as appropriate.

Philippe Kruchten, in his paper on 4+1 views of architecture, showed that it is impractical to use a single diagram to express everything about your architecture and design (**?**,). He explained that you need distinct architectural views because each has its own abstractions, notation, concerns, stakeholders, and patterns. Each view can use an appropriate notation and focus on a single concern, which makes it easy to understand. Together, the views comprise a full architecture model, and each view presents a subset of the details from that full model.

View consistency. Each view (or diagram) you create of a domain, design, or code model shows a single perspective on that model, exposing some details and hiding others. The diagrams are not isolated parts of the model, like drawers in a cabinet. Instead, they are live projections of the model and the views are consistent with each other. So if the model changes, the views do too. House blueprints are views of a house (or its design) so you expect them to be consistent with each other.

For example, imagine that you have two views of the domain model: a list of types in the networking and job advertisement domain (such as Ads, Jobs, and Contacts), and a scenario (a story) describing them. We will describe scenarios in more detail soon, but for now consider it a story told about how the domain types interact over time. If you were to revise the scenario to reference a new domain type, like a declined invitation to join a contact network, you would expect to see that type in the list of defined types. If it is not there, it is a bug in your domain model.

Master models. The domain, design, and code models are each conceptually a single *master model*. Every view you draw must be consistent with that master model. Think of it this way: when you revised the scenario to refer to a new type, your understanding of the master model was revised. Since any other view is derived from the master model, it should reflect your new understanding. Disregarding pragmatics for a moment, all of the diagrams you build should be consistent at all times with each other because that is the way that you, in your mind, understand the domain to work. Pragmatically, however, while you are building models there will be times when they are inconsistent with each other, but you strive to eliminate those bugs.

To reinforce the idea of unified and consistent models, it may be helpful to imagine a programming environment where all of these elements fit together and are typechecked. In that programming environment, a scenario that tried to refer to a type that was not defined in the master model would yield a type checking error.

Discussing views formally makes them sound difficult, but in reality people can use them with almost no effort. For example, you can imagine your bookcase as it is now, or imagine it with only the red books, or imagine it with the red books rotated so that you can see the cover instead of the spine. Each of these is a view of your master model of the bookcase. Notice that while you have never written down a model of your bookcase, you nonetheless have one in your head that you can manipulate. One of the challenges in software development is ensuring that developers, subject matter experts, and others all have the same master model in their heads.

Examples of master models. Master models are a helpful concept because they explain what your views refer to, but you have options regarding what the master model represents. The most straightforward example of a master model is an already existing system. You can create many views of an existing system. Consider your neighborhood as an example of an existing system. You do not have a complete model of your neighborhood written down anywhere, but you do have the neighborhood itself. Views of the neighborhood can be tested against the neighborhood to see if they are consistent with that master model.

Another example of a master model is a system that will be built. Unlike your neighborhood, this system does not yet exist, so it is a bit trickier to build views of it and ensure the views are consistent. Yet somehow things tend to work out OK. You might embark on a project to renovate a room in your house without writing down any explicit models, but you must have a master model in your head in some form. That model includes details about what should happen when (for example, demolition happens *before* painting) and cost estimates. That model in your head is likely incomplete, so views of it will necessarily be incomplete too.

Here are some concrete examples of master models of software systems. The master model may be the system you previously built or a system you plan to build. It can be a combination of the two, such as an existing system with planned additions.

Or it could be even more complex, such as a model of the system as you expect it to look at three-month intervals over the next few years.

Limiting size and focusing attention. You use views in modeling to limit the size of the diagrams and to focus attention. Imagine how confusing a medium-sized domain model would be if you tried to show all the types, definitions, behaviors, etc., on the same diagram. You may have seen the giant printouts of corporate database schemas taped to a wall somewhere and seen people trying to use them by putting a finger in one place and tracing the lines to other parts of the diagram. Views avoid that.

7.5 Other ways to organize models

The canonical model structure from this chapter consists of a domain model, a design model, and a code model. This basic organization of models has a long history, visible in the Syntropy software development process (Cook and Daniels, 1994), though it probably traces back even further.

Other authors have proposed similar model structures, and while there are some differences in their organizations and nomenclature, there is a core similarity shared by all. With only a little bit of squinting, one can identify the domain, design (boundary and internals), and code models. Figure 7.5 is a summary that maps this book's model names to some of those found elsewhere.

Despite the broad strokes of similarity between authors, there are differences. The one concept that does not align well across authors is *requirements*, because it can mean different things to different people. Requirements models could overlap with business models, domain models, boundary models, or internals models.

7.6 Business modeling

There is a kind of model not found in this book's canonical model structure: *business models*. Business models describe what a business or organization does and why it does it. Different businesses in the same domain will have different strategies, capabilities, organizations, processes, and goals and therefore different business models.

Domain modeling is related to *business modeling*, which includes not only facts but also decisions and goals that organizations must make. Someone at some point decides what the organization does and the processes it follows. Some of the processes are partly or fully automated with software. The goals and decisions of an organization can be influenced by the software that you build and buy.

So why include domain models but not business models in this book? This book includes domain modeling because misunderstanding the domain is a common cause of failure in IT projects. Misunderstanding business processes can also cause failures, but those are rarely engineering failures.

	Business Model	Domain Model	Design Model		Code Model
			Boundary Model	Internals Model	
Bosch			System context	Component design	Code
Cheesman & Daniels		Business concept	Type specs	Component architecture	Code
D'Souza (MAp)	Business architecture	Domain	Blackbox	Whitebox	Code
Software Engineering Institute (SEI)			Requirements	Architecture	Code
Jackson		Domain	Domain + machine	Machine	
RUP	Business modeling	Business modeling	Requirements	Analysis & design	Code
Syntropy		Essential	Specification	Implementation	Code

Figure 7.5: A table summarizing the models proposed by various authors and how they map to the business, domain, design (boundary, internals), and code models found in this book.

7.7 Use of UML

This book uses Unified Modeling Language (UML) notation because it is ubiquitous and its addition of architectural notation in UML 2.0 has brought it visually closer to special purpose architecture languages. This book deviates from strict UML in a few places. Any remaining deviations from UML are inadvertent.

- In UML, connectors can be solid lines or ball-and-socket style. They are distinguished using stereotypes to indicate their types. In this book, connectors are shown using a variety of line styles, which is a more compact way to convey their types and can be less cluttered.
- In UML, a port's type is shown with a text label near it. This book uses that style, but it sometimes clutters the diagram, in which case ports are shaded and defined in a legend. Not all UML tools allow shading or coloring of ports.

7.8 Conclusion

Once you begin to build models of your system, you realize that understanding and tracking lots of little models is hard, but building a single gigantic model is impractical. The strategy proposed in this chapter is to build small models that fit into a canonical model structure. If you understand the canonical structure then you will understand where each model fits in.

The first big idea was to use designation and refinement to create models that differ in their abstraction. The primary models are the domain model, design model, and code model, and they range from abstract to concrete. The second big idea was to use *views* to zoom in on the details of a model. Since the views are all projections of a single master model, their details are consistent (or are intended to be). In order to hierarchically nest design models, you use refinement to relate boundary and internals models.

Coaches see and understand more than rookies not because they have sharper eyes, but because they have a conceptual model that helps them categorize what they are seeing. This chapter describes the entire canonical model structure in detail, but do not let this alarm you. In practice you would rarely, if ever, create every possible model and view. Once you have internalized these ideas, they will help you to understand where a given detail, diagram, and model fits. As shown in the case study (Chapter 4) and the chapter on the risk-driven model (Chapter 3), following a risk-driven approach to architecture encourages you to build a subset of models, ones that help you reduce risks you have identified. This chapter, and subsequent ones, provides detailed descriptions to help you can internalize the models and thus be better at building software, not to encourage you towards analysis paralysis.

7.9 Further reading

This book is a synthesis of the architectural modeling approaches invented by other authors. It has three primary influences. The first is the work on modeling components in UML from D'Souza and Wills (1998) and Cheesman and Daniels (2000), which focus primarily on modeling functionality. The second is the quality attribute centric approach from the Software Engineering Institute (Bass, Clements and Kazman, 2003; Clements et al., 2010) and Carnegie Mellon University (Shaw and Garlan, 1996). The third is the agile software development community (Boehm and Turner, 2003; Ambler, 2002) which encourages efficient software development practices.

There are several good books that describe the general concepts of software architecture. Bass, Clements and Kazman (2003), describes a quality-attribute centric view of software architecture and provides case studies of applying their techniques. Taylor, Medvidović and Dashofy (2009) is a more modern treatment and is logically

organized like a textbook. Shaw and Garlan (1996) is becoming dated but is the best book for understanding the promise of software architecture. Clements et al. (2010) is an excellent reference book for architecture concepts and notations (and also has a useful appendix on using UML as an architecture description language). These books rarely venture down into objects and design, but D'Souza and Wills (1998) and Cheesman and Daniels (2000) do, showing how architecture fits into object-oriented design.

Probably more than any other book, Bass, Clements and Kazman (2003) has shaped the way the field thinks about software architecture, shifting the focus away from functionality and towards quality attributes. It describes not only the theory but also processes for analyzing architectures and discovering quality attribute requirements. The book also contains a great discussion of the orthogonality of functionality and quality attributes.

Rozanski and Woods (2005) offer perhaps the most complete treatment of how to understand and use multiple views in software architecture. It also contains valuable checklists relating to several standard concerns.

The simplest pragmatic approach to component-based development is found in Cheesman and Daniels (2000). They lay out an organizational structure for models using UML and treat components as abstract data types with strict encapsulation boundaries. A similar approach, but with greater detail, is found in D'Souza and Wills (1998). Both emphasize detailed specifications, such as pre- and post-conditions, as a way to catch errors during design. This book de-emphasizes pre- and post-conditions because on most projects they are too expensive, but the mindset they encourage is excellent.

The best book at articulating a vision of software engineering that includes software architecture is probably Shaw and Garlan (1996). While reading it, it is difficult not to share their enthusiasm for how architecture can help our field.

The nuts and bolts of architectural modeling, including pitfalls, are well described by Clements et al. (2010). One of the book's goals is to teach readers how to document the models in a documentation package, which can be important on large projects.

To date, the most comprehensive treatment of software architecture is by Taylor, Medvidović and Dashofy (2009) in their textbook on software architecture. It covers real-world examples of software architecture as well as research developments on formalisms and analysis.

Developers working in the field of Information Technology (IT) will be well served by Ian Gorton's treatment of software architecture, as his book covers not only the basics of software architecture, but also the common technologies in IT, such as Enterprise Java Beans (EJB), Message-Oriented Middleware (MOM), and Service Oriented Architecture (SOA) (Gorton, 2006).

Using abstraction to organize a stack of models is an old technique. It is used in the Syntropy object oriented design method (Cook and Daniels, 1994) and is central to Cheesman and Daniels (2000), Fowler (2003a), and D'Souza and Wills (1998).

Many authors have suggested ways of organizing and relating architecture models. Jan Bosch models the system context, the archetypes, and the main components (Bosch, 2000). John Cheesman and John Daniels propose building a model of the requirements (a business information model and a scenario model) and a model of the system specification (a business type model, interface specifications, component specifications, and the component architecture) (Cheesman and Daniels, 2000). Desmond D'Souza, in MAp, suggests modeling the business architecture, the domain, and the design as a blackbox and a whitebox (D'Souza, 2006). David Garlan conceives as architecture being a bridge between the requirements and the implementation (Garlan, 2003). Michael Jackson suggests modeling the domain, the domain with the machine, and the machine (Jackson, 1995). Jackson's primary focus is on system requirements engineering, not design, but his specifications overlap well with design. The Rational Unified Process (RUP) does not advocate specific models, but suggests activities for business modeling, requirements, and analysis & design (Kruchten, 2003).

Every developer should be familiar with the 4+1 architecture views paper (**?**,), but also be aware that it is just one of many different sets of views that have been proposed for architecture, such as the Siemens Four Views (Hofmeister, Nord and Soni, 2000).

You should also be aware of the IEEE standard description of software architecture, IEEE 1471-2000 (Society, 2000). In it, you will find most of the same concepts as in this book. It has a few additions and differences worth noting. While it uses *views*, it treats them as requirements from the *viewpoint* of a *stakeholder* focused on a particular *concern*, rather than as projections of a consistent master model, what it would call an *architecture description*. It also describes the *environment* the system inhabits, its *mission*, and *library viewpoints* (which are reusable viewpoint definitions).

Authors are increasingly paying attention to business process modeling in addition to domain modeling. Martin Ould provides a practical process for modeling business processes (Ould, 1995). Desmond D'Souza describes how to connect business processes to software architecture by connecting business goals to system goals (D'Souza, 2006).

The relationship between software architecture (specifically enterprise architecture) and business strategy is covered in Ross, Weill and Robertson (2006). As software developers, we perhaps assume that the natural future state should be that all systems can inter-operate. The surprising thesis of the book is that the level of integration should relate to the chosen business strategy.

Glossary

Action specification A (sometimes formal) specification of a method, procedure, or more abstraction behavior. Often consists of a pre-condition (what must be true for the method to successfully run) and a post-condition (what the method guarantees will be true after it completes). See *design by contract*.

Agile process A style of software development process characterized by iterative development. See *waterfall* process, *Extreme Programming*, *iterative process*, *agile process*, and *spiral process*.

Allocation element (i.e., *UML node* or *environmental element*) Hardware (such as computers) and geographical locations (such as datacenters) that can host *modules* and *component* instances. UML (Booch, Rumbaugh and Jacobson, 2005) refers to places where software can be deployed as *nodes* and the SEI authors (Bass, Clements and Kazman, 2003) refer to it as an *environmental element*.

Allocation viewtype The viewtype that contains views of elements related to the deployment of the software onto hardware. It includes deployment diagrams, descriptions of environmental elements like servers, and descriptions of communication channels like ethernet links. It may also include geographical elements, so that you can describe two servers in different cities. See *runtime viewtype* and *module viewtype*.

Analogic model In an analogic model, each model element has an analogue in the domain of interest. A radar screen is an analogic model of some terrain, where blips on the screen correspond to airplanes — they are analogues. Analogic models support analysis only indirectly, and usually domain knowledge and human reasoning are required. See *analytic model.*

Analysis paralysis The situation where a developer spends inordinate time analyzing or building models, and not building a solution.

Analytic model An analytic model directly supports analysis of the domain of interest. Mathematical equations are examples of analytic models, as are state machines. You could imagine an analytic model of the airplanes where each is represented by a vector. Mathematics provides an analytic model to evaluate the vectors, so you could quantitatively answer questions about collision courses. See *analogic model.*

Anonymous instance An instance (such as an object or a component instance) that has not been given a name. Graphically, it is labeled ": TypeName". In contrast, a named instance would have a name preceding the colon.

Application architect Application architects are developers who are responsible for a single application. It is possible for them to understand and manage thousands of objects that comprise their application. Application architects are like movie directors whose daily actions create the shape of the product.

Application Programming Interface (API) A set of operations that can be performed on a module, component, or object. When we refer to API-level operations, we mean that they are not abstract, and those operations are exactly what would be seen in the programming language.

Architectural style (i.e., architectural pattern). An architectural style is "a specialization of element and relation types, together with a set of constraints on how they can be used." (Clements et al., 2010)

Architecturally-evident coding style A style of programming that encodes additional design intent by providing hints about the system's architecture. It encourages you to embed hints in the source code that make the architecture evident to a developer who reads the code. It follows the *model-in-code principle*.

Architecture see *software architecture*.

Architecture description language (ADL) A language used to describe architectures that defines elements (e.g., components, connectors, modules, ports) and relationships. Examples include UML, C2, AADL, and Acme.

Architecture drift Architecture drift is the tendency for a system, over time, to violate its initial design. (Perry and Wolf, 1992)

Architecture driver *Quality attribute scenarios* or *functionality scenarios* that are both important to stakeholders and difficult to achieve. As such, they are the scenarios that you should pay most attention to when designing the system (Bass, Clements and Kazman, 2003)

Architecture hoisting When following an *architecture hoisting* approach, developers design the architecture with the intent of guaranteeing a goal or property of the system. The idea is that once a goal or property has been hoisted into the architecture, developers should not need to write any additional code to achieve it. See *architecture-focused design* and *architecture-indifferent design*.

Architecture refactoring A *refactoring* of a system's architecture, possibly from one architectural style to another, or the introduction of consistency (see *constraints*) where none were present before.

Architecture-focused design In *architecture-focused design*, developers are aware of their system's software architecture and they deliberately choose it so that their system can achieve its goals. See *architecture-indifferent design* and *architecture hoisting*.

Architecture-indifferent design In *architecture-indifferent design*, developers are oblivious to their system's architecture and *do not* consciously choose an architecture to help them reduce risks, achieve features, or ensure qualities. The developers may simply ignore their architecture, copy the architecture from their previous project, use the presumptive architecture in their domain, or follow a corporate standard. See *architecture-focused design* and *architecture hoisting*.

Baked-in risks When a process is designed to always address a certain risk, that risk is said to be baked-in to the process. For example, agile processes address customer rejection risk by building and delivering the system incrementally.

Big Design Up Front (BDUF) In Big Design Up Front (BDUF), the early weeks or months of a project are primarily spent designing instead of prototyping or building. It is a pejorative term coined by people, like agile advocates, who are concerned about *analysis paralysis*, a situation where a project spends too much time designing and not enough time building. BDUF is associated more with waterfall processes than spiral processes.

Binary connector A connector that can be attached to just two components. See *N-way connector*.

Binding (1) Using bindings, ports on an external component are bound to compatible or identical ports on internal components. Invariants and quality attribute scenarios on the external component must be satisfied by the internal components. (2) The binding relationship is used to show correspondence between parts in pattern and elements in a model using that pattern.

Boundary model The boundary model is what outsiders can see of the system (or an element in the system), which includes its behavior, interchange data, and quality attributes. The boundary is a commitment to an interface but not to implementation details. The boundary model describes what a user needs to know to understand how a system works. It is an encapsulated view of the system that hides internal details. When developers change the internal design, users are undisturbed. See *internals model*.

Business model A business model describes what a business or organization does and why it does it. Business models rarely talk about software. Different businesses in the same domain will have different strategies, capabilities, organizations, processes, and objectives and therefore different business models. It describes not only facts (which would appear in a *domain model*) but also decisions and goals that organizations must make.

Canonical model structure A set of models, ranging from abstract to concrete, that use views to drill down into the details of each model. It consists of three primary models: the domain model, the design model, and the code model. The canonical model structure has the most abstract model (the domain) at the top and the most concrete (the code) at the bottom. The *designation* and *refinement* relationships ensure that the models correspond, yet enable them to differ in their level of abstraction.

Classification relationship A classification relationship is the same one that exists between classes and objects in object-oriented programming.

Closed semantics In refinement with closed semantics, the refinement restricts what kinds of new items can be introduced by listing the kinds of items that will not change. See *open semantics*.

Code model The code model describes the system source code. The code model is either the source code implementation of the system or a model that is equivalent. It could be the actual Java code or the result of running a code-to-UML tool, but its important feature is that has a full set of design commitments. Where the design model has an incomplete set of design commitments, the code model has a complete set, or at least a sufficiently complete set to execute on a machine. Compare with the *domain model* and *design model*. All three are part of the *canonical model structure*.

Commercial Off-The-Shelf (COTS) *Modules*, *components*, or other source code available from third parties. This term is often used even if they are open source or from a non-commercial group.

Communication channel (i.e., *connection* or *environmental element*) Hardware that allows *allocation elements* to communicate. UML (Booch, Rumbaugh and Jacobson, 2005) refers to the communication channels between *nodes* as *connections* and the SEI authors (Bass, Clements and Kazman, 2003) refer to them as *environmental elements*.

Component "Components [are] the principal computation elements and data stores that execute in a system." (Clements et al., 2010) Usually refers to a component instance, but could also refer to a component type. See *module*.

Component assembly (i.e., component and connector diagram) A component assembly shows a specific configuration of component, port, and connector instances or types. Their arrangement is the component design and different arrangements will yield different qualities. It may show *bindings* between external and internal *ports*.

Component-Based Development (CBD) Software development whose end-product is loosely-coupled components to be sold in a component marketplace.

Conceptual model A conceptual model identifies salient features and how they operate. Introductory physics classes teach Newtonian mechanics, a conceptual model of how physical objects behave, which includes features like mass and forces.

Connector A connector is a pathway of runtime interaction between two or more components. This is just slightly different than the definition from (Clements et al., 2010), which states that a "connector [is] a runtime pathway of interaction between two or more components."

Constraint See *invariant*.

System context diagram A component assembly that focuses on the system being designed and includes all external systems that the system connects to.

Design by contract Bertrand Meyer popularized the concept of design by contract where method pre- and post-conditions as well as object *invariants* are inserted into the source code and checked by automated tools (Meyer, 2000). By relying on a method's contract, clients can safely ignore any internal implementation and treat the method or the entire object as a black box.

Design decision Decisions made by developers during the course of designing the system that commit the project to a particular design choice or restrict the design space. See *invariant*.

Design intent The understanding and intentions of the system's developers. Design intent is imperfectly contained in the source code of a system, forcing developers to infer parts of it.

Design model The design model describes the system you will build, and is largely under your control. The system to be built appears in the design model. The design model is a partial set of design commitments. That is, you leave undecided some (usually low-level) details about how the design will work, deferring them until the code model. The design model is composed of recursively nested *boundary models* and *internals models*. Compare with the *domain model* and *code model*. All three are part of the *canonical model structure*.

Designation A designation relationship allows you to show correspondences between two domains, for example between the real world and a problem domain model. It identifies that something from one domain corresponds to something in a second domain.

Documentation package A complete, or mostly complete, written description of a software architecture.

Domain connector A connector that bridges the domains of the components it connects. When two components interact, there is often some logic that is dependent on the domain of both components. By putting this logic into a domain connector, you insulate each of the components from knowing unnecessary details about the other.

Domain driven design Domain driven design advocates a embedding the *domain model* in the source code (Evans, 2003). It is compatible with the *model-in-code principle* but goes further by encouraging an *agile* development process and discouraging expressing domain models on paper.

Domain model The domain model describes enduring truths about the domain that are relevant to your system. In general, the domain is not under your control, so you cannot decide that weeks have six days or that you have a birthday party every week. The system to be built does not appear in the domain model. Compare with the *design model* and *code model*. All three are part of the *canonical model structure*.

Dominant decomposition The organizational system of a system that promotes a single concern. Problems related to that dominant concern will be easier to solve, but problems related to other concerns will be harder. For example, if you organize books by their size, then it will be easy to find the tallest books but harder to find ones by a specific author. This problem of one concern dominating others is referred to as the *tyranny of the dominant decomposition* (Tarr et al., 1999).

Driver See *architecture driver*.

Dynamic architecture model A model that generalizes all the possible instantaneous configurations (e.g., topology of component instances) of an architecture. Most systems change during startup and shutdown, but have a long steady-state configuration in between that is modeled as a static architecture model.

Effective encapsulation Encapsulation where the boundary does not unnecessarily leak abstractions across its interface. Ultimately, what counts as effective is subjective and requires good judgment.

Encapsulation "[T]he process of compartmentalizing the elements of an abstraction that constitute its structure and behavior; encapsulation serves to separate the contractual interface of an abstraction and its implementation." (Booch et al., 2007)

Engineering risk A risk related to the analysis, design, and implementation of the product. See *project management risk*.

Enterprise architect Architects who are responsible for many applications, who do not control the functionality of any one application, and who instead design an ecosystem inside which individual applications contribute to the overall enterprise. Enterprise architects are like movie producers in that they influence the outcome only indirectly.

Enterprise architecture The software architecture of an organization that spans multiple applications (systems).

Environmental element (i.e., UML node) Hardware with the primary purpose of running software that communicates via *communication channels*.

Event bus An *N-way* publish-subscribe connector.

Evolutionary design Evolutionary design "means that the design of the system grows as the system is implemented" (Fowler, 2004). Often paired with *refactoring*. Compare with *planned design*.

Extensional element Extensional elements are enumerated, such as "The system is composed of a client, an order processor, and an order storage components." Examples include modules, components, connectors, ports, and component assemblies. See *intensional element*.

Extreme Programming_(XP) A specialization of an *iterative* and *agile* software development process, so it contains multiple iterations (Beck and Andres, 2004). It suggests avoiding up-front design work, though some projects add an *iteration zero* (Schuh, 2004), in which no customer-visible functionality is delivered. It guides developers to apply *evolutionary design* exclusively, though some projects modify it to incorporate a small amount of *planned design*. Each iteration is prioritized by the customer's valuation of features, not risks. Compare with *waterfall* process, *iterative process*, *agile process*, and *spiral process*.

Framework (i.e., software framework or object-oriented framework) A form of software reuse characterized by inversion of control. Frameworks, in contrast with libraries, are an effective means of sharing or reusing a software architecture.

Functionality scenario Functionality scenarios, also called simply *scenarios,* express a series of events that cause changes to a model. A scenario describes a single possible path rather than generalizing many paths. See *use case*.

Generalization The relationship between a more general type and a more specific type, such as furniture and chair.

Goal connector A goal connector has an assigned goal, or objective, that it is responsible for accomplishing. A developer who builds a goal connector must avoid failure by looking into the problem, discovering possible failure cases, and ensuring that the connector handles them. Goal connectors are usually complex as they have real domain work to do, and are responsible for seeing it completed. See *micromanaged connector*.

Information model A set of types and their definitions that describes the things that exist in the domain. It also describes the *relationships* between those types. It can be drawn textually, often as a table, or graphically, often using UML class diagram syntax.

Information Technology (IT) A specialty inside software design that focuses on "the study, design, development, implementation, support or management of computer-based information systems, particularly software applications and computer hardware." (Information Technology Association of America).

Intensional element Intensional elements are those that are universally quantified, such as "All filters can communicate via pipes." Examples include styles, invariants, responsibility allocations, design decisions, rationale, protocols, and quality attributes. See *extensional element*.

Internals model An internals model is a refinement of a *boundary model*. Both are views of the design model but they differ in the details they reveal. Anything that is true in the boundary model must be true in the internals model. Any commitments made in the boundary model (the number and type of ports, QA scenarios) must be upheld in the internals model. See *boundary model*.

Invariant Approximately the same as a *constraint*. Can be expressed as a predicate that is always true with respect to the system or design. Sometimes divided into static invariants (or representation invariants) that deal with static structures and dynamic invariants that deal with behaviors. The term *invariant* is more often used to apply to source code or data structures. When referring to systems, the term *constraint* is more often used.

Iteration A period of time in an iterative process where all software development activities can take place.

Iterative process An iterative development process builds the system in multiple work blocks, called *iterations* (Larman and Basili, 2003). With each iteration, developers are allowed to rework existing parts of the system, so it is not just built incrementally. Iterative development optionally has up-front design work but it does not impose a prioritization across the iterations, nor does it give guidance on the nature of design work. See *waterfall process*, *Extreme Programming*, *agile process*, and *spiral process*.

Layer A layered system organizes its modules such that lower layers act as virtual machines to higher layers. Dependencies are (almost) exclusively downward, where higher layers can use and depend on lower layers but not the reverse.

Link An edge between two objects in a snapshot (or instance diagram).

Master model A model that contains a complete set of details necessary to project out the views you build.

Method signature A specification of a method or procedure that usually includes the method name, its return type, and the types of its parameters. It can be augmented with pre-conditions and post-conditions to form an *action specification*.

Micromanaged connector An connector that simply does a job you assign to it. If it fails that is because you did not supervise it sufficiently. Its job is only to do what you told it to do. Micromanaged connectors are simple connectors. See *goal connector*.

Minimal planned design (i.e., Little Design Up Front) In between *evolutionary design* and *planned design* is minimal planned design (Martin, 2009). Advocates of minimal planned design worry that they might design themselves into a corner if they did all evolutionary design, but they also worry that all planned design is difficult and likely to get things wrong.

Model A symbolic representation of a system that contains only selected details.

Model-code gap The difference between how we express the solution in the design model and how we express it in the source code. See *intensional element* and *extensional element*.

Model-in-code principle Expressing a model in the system's code helps comprehension and evolution. A corollary of this principle is that expressing a model in code necessarily involves doing more work than is strictly necessary for the solution to work.

Module (i.e., package) A collection of implementation artifacts, such as source code (classes, functions, procedures, rules, etc.), configuration files, and database schema definitions. Modules can group together related code, revealing an interface but hiding the implementation.

Module viewtype The viewtype that contains views of the elements you can see at compile-time. It includes artifacts like source code and configuration files. Definitions of component types, connector types, and port types are also in the module viewtype, as are definitions of classes and interfaces. See *runtime viewtype* and *allocation viewtype*.

N-way connector A connector that can join one to many components, usually three or more, such as an *event bus*. See *binary connector*.

Navigation The idea that you can traverse from node to node in a model across edges. For example, you can navigate across a UML class diagram from class to class across the associations. See *Object Constraint Language*.

Object Constraint Language (OCL) A precise language for expressing invariants and constraints over UML models. See *navigation*.

Open semantics In refinement with open semantics, the refinement can introduce whatever new items it pleases. See *closed semantics*.

Parnas module A modularization technique where you ensure that the details likely to change are hidden inside the module, and that changes to those details will not influence the module's interface. A Parnas module hides a secret to minimize coupling, rather than just grouping together related code. See *encapsulation* and *effective encapsulation*.

Partition (1) As a noun, a relationship between parts and a whole such that the parts combine to form exactly the whole, no more and no less. (2) As a verb, a loose synonym with "divide" or "decompose." Or as in (1), the division of a system into disjoint pieces.

Pattern A pattern is a reusable solution to a recurring problem (Gamma et al., 1995).

Planned design (i.e., up-front design) A kind of software development process where design is done mostly or completely before implementation begins. See *evolutionary design* and *evolutionary design*.

Port All communication in or out of a component is done via ports on the component. All of the publicly available methods that a component supports, and all of the public events it responds to, will be specified in its ports. There is no necessary connection between ports on components and ports in an operating system.

Pre-condition & Post-condition See *action specification*.

Presumptive architecture A software architecture (or, more carefully, a family of architectures) that is dominant in a particular domain. Rather than justifying their choice to use it, developers in that domain may have to justify a choice that differs from the presumptive architecture. For example, a 3-Tier architecture is a presumptive architecture in many Information Technology (IT) groups. See *reference architecture*.

Projection See *view*.

Project management risk Risks related to schedules, sequencing of work, delivery, team size, geography, etc. See *engineering risk*.

Property Model elements can be annotated with properties that elaborate details about the element. For example, a connector can be annotated with a property describing its protocol or its throughput.

Prototype (i.e., architectural spike or proof of concept) An implementation intended to reduce risk by demonstrating feasibility, evaluating properties, or similar. Not used pejoratively (i.e., "throwaway code") in this book.

Prototypical risk Each domain has a set of prototypical risks that is different from other domains. For example, Systems projects usually worry more about performance than IT projects.

Quality attribute (i.e., QA's, extra-functional requirements, or the "-ities") A quality attribute is a kind of extra-functional requirement, such as performance, security, scalability, modifiability, or reliability.

Quality attribute scenario (i.e., QA scenario) A concise description of an extra-functional requirement, consisting of a source, stimulus, environment, artifact, response, and response measure.

Rational architecture choice Rational architecture choices are ones where your tradeoffs align with your quality attribute priorities. They often follow this template: Since <x> is a priority, we chose design <y>, and accepted downside <z>.

Rational Unified Process (RUP) A meta-process that can be tailored, for example into an *iterative*, *spiral*, or *waterfall* process.

Refactoring A code or design transformation that improves its structure, or other quality, while preserving its behavior. See *architecture refactoring*. (Fowler, 1999)

Reference architecture A specification that describes an prescribed architectural solution to a problem. Reference architectures are often proposed by vendors or experts as the canonical architectures for given problems. See *presumptive architecture*. (Bass, Clements and Kazman, 2003)

Refinement Refinement is a relationship between a low-detail and a high-detail model of the same thing.

Responsibility-driven design In contrast to thinking about data and algorithms, responsibility-driven design focuses on roles and responsibilities.

Risk In this book, risk is the perceived probability of failure times the perceived impact.

Risk-driven model The risk-driven model of software architecture guides developers to apply a minimal set of architecture techniques to reduce their most pressing risks. It suggests a relentless questioning process: "What are my risks? What are the best techniques to reduce them? Is the risk mitigated and can I start coding?" The key element of the risk-driven model is the promotion of risk to prominence.

Role (1) In a UML class diagram, the name on the end of an association. (2) The typed end of a connector, roughly equivalent to a port on a component. (3) In a pattern, a part that can be bound or substituted for a concrete part in the implementation.

Runtime viewtype (i.e., component & connector viewtype) The viewtype that contains views of elements that you can see at runtime. It includes artifacts like functionality scenarios, responsibility lists, and the component assemblies. Instances of components, connectors, and ports are in the runtime viewtype, as are objects (class instances). See *module viewtype* and *allocation viewtype*.

Scale When referring to software, scale usually refers to the absolute size of a system, often counted in lines of code. Scalability (a quality attribute) refers to the ability of a system to handle a greater load than it currently does, such as running on larger hardware (as in vertical scalability) or more copies of the hardware (horizontal scalability). Somewhat confusingly, the question, "Will it scale?" refers to a system's scalability, not its lines of code.

Scenario Usually refers to a *functionality scenario* but could also refer to a *quality attribute scenario*.

Snapshot (i.e., instance diagram) A diagram showing objects or component instances at an instant in time.

Software architecture This is the standard definition from the SEI: "The software architecture of a computing system is the set of structures needed to reason about the system, which comprise software elements, relations among them and externally visible properties of both." (Clements et al., 2010)

Software Engineering Institute (SEI) A federally funded research and development center whose mission is to "advance software engineering and related disciplines to ensure the development and operation of systems with predictable and improved cost, schedule, and quality."

Source code The programming language statements typed by developers that appear cryptic to the uninitiated.

Spanning viewtype The viewtype that contains views that cross over between two or more viewtypes. An example of a tradeoff that spans viewtypes is: you decide to denormalize a database schema (which would be described in the module viewtype) in order to achieve greater transaction throughput (which would be described in the runtime viewtype), so you describe that tradeoff in the spanning viewtype.

Spiral process The spiral process (Boehm, 1988) is a kind of iterative development, so it has many iterations, yet it is often described as having no up-front design work. Iterations are prioritized by risk, with the first iteration handling the riskiest parts of a project. The spiral model handles both management and engineering risks. For example, it may address "personnel shortfalls" as a risk. The spiral process gives no guidance on the nature of design work, or on which architecture and design techniques to use. See *waterfall process*, *Extreme Programming*, *iterative process*, and *agile process*.

Stakeholder A customer or other person who has an interest in the features or success of a system.

Static architecture model A model of a system that shows it at an instant in time or in its steady state configuration. See *dynamic architecture model*.

Story at many levels A way of structuring your software such that each level of nesting tells a story about how those parts interact. A developer who was unfamiliar with the system could be dropped in at any level and still make sense of it rather than being swamped. Its primary benefit is cognitive, not technical.

Subject Matter Expert (SME) A domain expert, sometimes a customer.

System context diagram A *component assembly* diagram in the *top-level boundary model* that includes the system (as a component) and its connections (as connectors) to external systems. See *use case diagram*.

Tactic In Attribute Driven Design, a tactic is a kind of pattern that is bigger than a design pattern and smaller than an architectural style. Examples of tactics include: Ping/Echo, Active Redundancy, Runtime Registration, Authenticate Users, and Intrusion Detection (Bass, Clements and Kazman, 2003).

Technical debt The accumulated misalignment of code with respect to the current understanding of the problem (Cunningham, 1992; Fowler, 2009)

Technique A software engineering activity performed by developers. Techniques exist on a spectrum from pure analyses, like calculating stresses, to pure solutions, like using a flying buttress on a cathedral. Other software architecture and design books have inventoried techniques on the solution-end of the spectrum, and call these techniques *tactics* (Bass, Clements and Kazman, 2003) or *patterns* (Schmidt et al., 2000; Gamma et al., 1995). This book focuses on techniques that are on the analysis-end of the spectrum, procedural, and independent of the problem domain.

Top-Down Design Top-down design is the process of refining a high-level specification of an element (component, module, etc.) into a detailed design by decomposing the element into smaller pieces and specifying those pieces by allocating responsibilities.

Top-level boundary model The top-level boundary model is the single, topmost encapsulated view of the *design model*. It can be refined into an *internals model* to show internal, non-encapsulated design details.

Tradeoff Sometimes getting more of one thing entails getting less of something else. Tradeoffs can exist between *quality attributes*, such as adding security can trade off against usability.

Two-level scenarios A *functionality scenario* that has been elaborated to show an additional level of internal messages, such as between the *components* in an *internals model*.

Ubiquitous language A common language shared by developers and domain experts, as opposed to the developers using one term and the domain experts using different one for the same concept. See *domain driven design*.

Unified Modeling Language (UML) A common modeling language suited to object-oriented design and software architecture.

Use case Use cases are largely equivalent to *functionality scenarios*, but there are some important differences. Use cases are activities that are high-level and visible to the users of the system. Use cases are often defined to be accomplishing a goal of an actor outside the system, so internal system activities would not count as use cases. Where functionality scenarios are a single trace of behavior, use cases can include variation steps that allow them to describe multiple traces.

Use case diagram A UML diagram showing actors, the system, and *use cases*.

View (i.e., projection) A view shows a defined subset of a *model*'s details, possibly with a transformation.

Viewpoint The view of a system from a single perspective, such as the view of a single stakeholder. Used in the IEEE definition of software architecture. Viewpoints are used in the views-as-requirements approach, rather than the *master model* approach to views.

Viewtype A set or category of views that can be easily reconciled with each other (Clements et al., 2010). See *module viewtype*, *runtime viewtype*, and *allocation viewtype*.

Waterfall process The waterfall process (Royce, 1970) proceeds from beginning to end as a single long block of work which delivers the entire project. It assumes planned design work that is done in its analysis and design phases. These precede the construction phase, which can be considered a single iteration. With just one iteration, work cannot be prioritized across iterations, but it may be built incrementally within the construction phase. See *Extreme Programming*, *iterative process*, *agile process*, and *spiral process*.

XP See *Extreme Programming*.

Yinzer A slang term for someone from Pittsburgh, home of Carnegie Mellon University, and is derived from *yinz*, which is Pittsburgh dialect equivalent to *y'all*, the plural form of *you*.

Bibliography

Abi-Antoun, Marwan, Wang, Daniel and Torr, Peter, Checking Threat Modeling Data Flow Diagrams for Implementation Conformance and Security. in: ASE '07: Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering. ACM, 2007, pp. 393–396.

Aldrich, Jonathan, Chambers, Craig and Notkin, David, ArchJava: Connecting Software Architecture to Implementation. in: ICSE '02: Proceedings of the 24th International Conference on Software Engineering. New York, NY, USA: ACM Press, 2002, pp. 187–197.

Alexander, Christopher, A Pattern Language: Towns, Buildings, Construction (Center for Environmental Structure Series). Oxford University Press, USA, 1977.

Alexander, Christopher, The Timeless Way of Building. Oxford University Press, 1979.

Ambler, Scott, Agile Modeling: Effective Practices for Extreme Programming and the Unified Process. Wiley, 2002.

Ambler, Scott, Agile Adoption Rate Survey Results: February 2008. Dr. Dobb's Journal, May 2008 ⟨http://www.ambysoft.com/surveys/agileFebruary2008.html⟩.

Ambler, Scott, Agile Architecture: Strategies for Scaling Agile Development. 2009 ⟨http://www.agilemodeling.com/essays/agileArchitecture.htm⟩.

Amdahl, Gene, Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities. AFIPS Conference Proceedings, 30 1967, pp. 483–485.

Apache Software Foundation, Hadoop Website. 2010 ⟨http://hadoop.apache.org⟩.

Babar, Muhammad Ali, An Exploratory Study of Architectural Practices and Challenges in Using Agile Software Development Approaches. Joint Working IEEE/IFIP Conference on Software Architecture 2009 & European Conference on Software Architecture 2009 September 2009.

Bach, James, Good Enough Quality: Beyond the Buzzword. IEEE Computer, 30 1997:8, pp. 96–98.

Barbacci, Mario et al., Quality Attributes. Software Engineering Institute, Carnegie Mellon University, 1995 (CMU/SEI-95-TR-021, ESC-TR-95-021). – Technical report.

Barbacci, Mario R. et al., Quality Attribute Workshops (QAWs), Third Edition. Software Engineering Institute, Carnegie Mellon University, 2003 (CMU/SEI-2003-TR-016). – Technical report.

Barrett, Anthony et al., Mission Planning and Execution Within the Mission Data System. in: Proceedings of the International Workshop on Planning and Scheduling for Space (IWPSS). 2004.

Bass, Len, Clements, Paul and Kazman, Rick, Software Architecture in Practice. 2nd edition. Addison-Wesley, 2003.

Bass, Len and John, Bonnie E., Linking Usability to Software Architecture Patterns through General Scenarios. Journal of Systems and Software, 66 2003:3, pp. 187–197.

Beck, Kent, Smalltalk Best Practice Patterns. Prentice Hall PTR, 1996.

Beck, Kent and Andres, Cynthia, Extreme Programming Explained: Embrace Change (2nd Edition). 2nd edition. Addison-Wesley Professional, 2004.

Beck, Kent et al., Manifesto for Agile Software Development. 2001 ⟨http://agilemanifesto.org⟩.

Beck, Kent and Cunningham, Ward, A Laboratory for Teaching Object Oriented Thinking. OOPSLA '89: Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications, 1989, pp. 1–6.

Bloch, Joshua, Extra, Extra - Read All About It: Nearly All Binary Searches and Mergesorts are Broken. June 2006 ⟨http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html⟩.

Boehm, Barry, A Spiral Model of Software Development and Enhancement. IEEE Computer, 21(5) 1988, pp. 61–72.

Boehm, Barry and Turner, Richard, Balancing Agility and Discipline: A Guide for the Perplexed. Addison-Wesley Professional, 2003.

Booch, Grady, Software Architecture presentation. 2004 ⟨http://www.booch.com/architecture/blog/artifacts/Software%20Architecture.ppt⟩.

Booch, Grady et al., Object-Oriented Analysis and Design with Applications. 3rd edition. Addison-Wesley Professional, 2007.

Booch, Grady, Rumbaugh, James and Jacobson, Ivar, The Unified Modeling Language User Guide. 2nd edition. Addison-Wesley Professional, 2005.

Bosch, Jan, Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach (ACM Press). Addison-Wesley Professional, 2000.

Bowker, Geoffrey C. and Star, Susan Leigh, Sorting Things Out: Classification and its Consequences. MIT Press, 1999.

Box, George E. P. and Draper, Norman R., Empirical Model-Building and Response Surfaces (Wiley Series in Probability and Statistics). Wiley, 1987.

Bredemeyer, Dana and Malan, Ruth, Bredemeyer Consulting. 2010 ⟨http://bredemeyer.com⟩.

Brooks, Frederick P, The Mythical Man-Month: Essays on Software Engineering. 2nd edition. Addison-Wesley Professional, 1995.

Buschmann, Frank et al., Pattern-Oriented Software Architecture Volume 1: A System of Patterns. Wiley, 1996.

Butler, Shawn A., Security Attribute Evaluation Method: A Cost-Benefit Approach. Proceedings of ICSE 2002, 2002, pp. 232–240.

Carr, Marvin J. et al., Taxonomy-Based Risk Identification. Software Engineering Institute, Carnegie Mellon University, June 1993 (CMU/SEI-93-TR-6). – Technical report.

Cheesman, John and Daniels, John, UML Components: A Simple Process for Specifying Component-Based Software. Addison-Wesley, 2000.

Chomsky, Noam, Syntactic Structures. 2nd edition. Walter de Gruyter, 2002.

Clements, Paul et al., Documenting Software Architectures: Views and Beyond. 2nd edition. Addison-Wesley, 2010.

Clerc, Viktor, Lago, Patricia and Vliet, Hans van, The Architect's Mindset. Third International Conference on Quality of Software Architectures (QoSA), 2007, pp. 231–248.

Cockburn, Alistair, Writing Effective Use Cases (Agile Software Development Series). Addison-Wesley Professional, 2000.

Coleman, Derek, Object-Oriented Development: The Fusion Method. Prentice Hall, 1993.

Conway, Melvin, How do Committees Invent? Datamation, 14 (5) 1968, pp. 28–31.

Cook, Steve and Daniels, John, Designing Object Systems: Object-Oriented Modelling with Syntropy. Prentice Hall, 1994.

Cook, William, On Understanding Data Abstraction, Revisited. OOPSLA: Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications 2009.

Coplien, James O. and Schmidt, Douglas C., Pattern Languages of Program Design. Addison-Wesley Professional, 1995.

Cunningham, Ward, The WyCash Portfolio Management System. OOPSLA '92: Addendum to the proceedings on Object-Oriented Programming Systems, Languages, and Applications, 1992, pp. 29–30.

Dean, Jeffrey and Ghemawat, Sanjay, MapReduce: Simplified Data Processing on Large Clusters. OSDI'04: Sixth Symposium on Operating System Design and Implementation December 2004.

Denne, Mark and Cleland-Huang, Jane, Software by Numbers: Low-Risk, High-Return Development. Prentice Hall, 2003.

Dijkstra, Edsger, Go-to Statement Considered Harmful. Communications of the ACM, 11 1968:3, pp. 147–148.

D'Souza, Desmond F., MAp: Model-driven Approach for Business-Aligned Architecture RoadMAps. 2006 ⟨http://www.kinetium.com/map/demo/demo_index.html⟩.

D'Souza, Desmond F. and Wills, Alan Cameron, Objects, Components and Frameworks with UML: The Catalysis Approach. Addison-Wesley, 1998.

Dvorak, Daniel, Challenging Encapsulation in the Design of High-Risk Control Systems. Proceedings of 2002 Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) 2002.

Eden, Amnon H. and Kazman, Rick, Architecture, Design, Implementation. International Conference on Software Engineering (ICSE), 2003, pp. 149–159.

Eeles, Peter and Cripps, Peter, The Process of Software Architecting. Addison-Wesley Professional, 2009.

Evans, Eric, Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley Professional, 2003.

Fairbanks, George, Why Can't They Create Architecture Models Like "Developer X"?: An Experience Report. in: ICSE '03: Proceedings of the 25th International Conference on Software Engineering. 2003, pp. 548–552.

Fairbanks, George, Bierhoff, Kevin and D'Souza, Desmond, Software Architecture at a Large Financial Firm. Proceedings of ACM SIGPLAN Conference on Object Oriented Programs, Systems, Languages, and Applications (OOPSLA) 2006.

Fay, Dan, An Architecture for Distributed Applications on the Internet: Overview of Microsoft's .NET Platform. IEEE International Parallel and Distributed Processing Symposium April 2003.

Feather, Steven Cornford Martin and Hicks, Kenneth, DDP: A Tool for Life-Cycle Risk Management. IEEE Aerospace and Electronics Systems Magazine, 21 2006:6, pp. 13–22.

Firesmith, Donald G., Common Concepts Underlying Safety, Security, and Survivability Engineering. Software Engineering Institute, Carnegie Mellon University, December 2003 (CMU/SEI-2003-TN-033). – Technical Note.

Foote, Brian and Yoder, Joseph, Chap. 29, Big Ball of Mud. In Pattern Languages of Program Design 4. Addison-Wesley, 2000.

Fowler, Martin, Analysis Patterns: Reusable Object Models. Addison-Wesley Professional, 1996.

Fowler, Martin, Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional, 1999.

Fowler, Martin, Patterns of Enterprise Application Architecture. Addison-Wesley Professional, 2002.

Fowler, Martin, UML Distilled: A Brief Guide to the Standard Object Modeling Language. 3rd edition. Addison-Wesley Professional, 2003a.

Fowler, Martin, Who Needs an Architect? IEEE Software, 20 (5) 2003b, pp. 11–13.

Fowler, Martin, Is Design Dead? 2004 ⟨http://martinfowler.com/articles/designDead.html⟩.

Fowler, Martin, Technical Debt. February 2009 ⟨http://martinfowler.com/bliki/TechnicalDebt.html⟩.

Gabriel, Richard P., Lisp: Good News Bad News How to Win Big. AI Expert, 6 1994, pp. 31–39 ⟨http://www.laputan.org/gabriel/worse-is-better.html⟩.

Gamma, Erich et al., Design Patterns: Elements of Reusable Object-Oriented Software (Addison-WesleyProfessional Computing Series). Addison-Wesley Professional, 1995.

Garlan, David, Software Architecture Course. 2003 ⟨http://www.cs.cmu.edu/~garlan/courses/Architectures-S03.html⟩.

Garlan, David, Allen, Robert and Ockerbloom, John, Architectural Mismatch, or, Why It's Hard to Build Systems Out of Existing Parts. in: Proceedings of the 17th International Conference on Software Engineering (ICSE). Seattle, Washington, April 1995, pp. 179–185.

Garlan, David, Monroe, Robert T. and Wile, David, Acme: Architectural Description of Component-Based Systems. in: **Leavens, Gary T. and Sitaraman, Murali, editors:** Foundations of Component-Based Systems. Cambridge University Press, 2000. – chapter 3, pp. 47–67.

Garlan, David and Schmerl, Bradley, AcmeStudo web page. 2009 ⟨http://www.cs.cmu.edu/~acme/AcmeStudio/index.html⟩.

Gluch, David P., A Construct for Describing Software Development Risks. Software Engineering Institute, Carnegie Mellon University, July 1994 (CMU/SEI-94-TR-14).– Technical report.

Gorton, Ian, Essential Software Architecture. Springer, 2006.

Harrison, William H. and Ossher, Harold, Subject-Oriented Programming (A Critique of Pure Objects). Proceedings of 1993 Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), 1993, pp. 411–428.

Heineman, George T. and Councill, William T., Component-Based Software Engineering: Putting the Pieces Together. Addison-Wesley Professional, 2001.

Hoff, Todd, Amazon Architecture. 2008a ⟨http://highscalability.com/amazon-architecture⟩.

Hoff, Todd, How Rackspace Now Uses MapReduce and Hadoop to Query Terabytes of Data. HighScalability.com, January 30 2008b ⟨http://highscalability.com/how-rackspace-now-uses-mapreduce-and-hadoop-query-terabytes-data⟩.

Hofmeister, Christine, Nord, Robert and Soni, Dilip, Applied Software Architecture. Addison-Wesley, 2000.

Holmes, James, Struts: The Complete Reference, 2nd Edition. McGraw-Hill Osborne Media, 2006.

Holmevik, Jan R., Compiling SIMULA: A Historical Study of Technological Genesis. IEEE Annals of the History of Computing, 16 1994:4, pp. 25–37.

Holzmann, Gerard J., The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley Professional, 2003.

Hood, Stu, MapReduce at Rackspace. January 2008 ⟨http://blog.racklabs.com/?p=66⟩.

Ingham, Michel D. et al., Engineering Complex Embedded Systems with State Analysis and the Mission Data System. AIAA Journal of Aerospace Computing, Information and Communication, 2 December 2005:12, pp. 507–536.

Jackson, Daniel, Alloy: A Lightweight Object Modelling Notation. ACM Transactions on Software Engineering and Methodology (TOSEM'02), 11 April 2002:2, pp. 256–290.

Jackson, Michael, Software Requirements and Specifications. Addison-Wesley, 1995.

Jackson, Michael, Problem Frames: Analyzing and Structuring Software Development Problems. Addison-Wesley, 2000.

Jacobson, Ivar, Booch, Grady and Rumbaugh, James, The Unified Software Development Process. Addison-Wesley Professional, 1999.

Kay, Alan, Predicting the Future. Stanford Engineering, 1 Autumn 1989:1, pp. 1–6.

Kruchten, Philippe, The Rational Unified Process: An Introduction. 3rd edition. Addison-Wesley Professional, 2003.

Larman, Craig and Basili, Victor R., Iterative and Incremental Development: A Brief History. IEEE Computer, 36 2003:6, pp. 47–56.

Lattanze, Anthony J., Architecting Software Intensive Systems: A Practitioners Guide. Auerbach Publications, 2008.

Liskov, Barbara, Keynote address - Data Abstraction and Hierarchy. Conference on Object Oriented Programming Systems Languages and Applications, 1987, pp. 17 – 34.

Luhmann, Niklas, Modern Society Shocked by its Risks. Social Sciences Research Centre: Occasional Papers, 1996 ⟨http://hub.hku.hk/handle/123456789/38822⟩.

Magee, Jeff and Kramer, Jeff, Concurrency: State Models and Java Programs. 2nd edition. Wiley, 2006.

Maranzano, Joseph, Architecture Reviews: Practice and Experience. IEEE Computer, 2005, pp. 34–43.

Martin, Robert, The Scatology of Agile Architecture. April 2009 ⟨http://blog.objectmentor.com/articles/2009/04/25/the-scatology-of-agile-architecture⟩.

Meier, J.D. et al., Checklists for Application Architecture. 2003 ⟨http://www.codeplex.com/wikipage?ProjectName=AppArch&title=Checklists⟩.

Meyer, Bertrand, Object-Oriented Software Construction. 2nd edition. Prentice Hall PTR, 2000.

Meyer, Kenny, Mission Data System Website. 2009 ⟨http://mds.jpl.nasa.gov⟩.

Miller, Granville, Second Generation Agile Software Development. March 2006 ⟨http://blogs.msdn.com/randymiller/archive/2006/03/23/559229.aspx⟩.

Monson-Haefel, Richard, Enterprise JavaBeans. 3rd edition. O'Reilly, 2001.

Moriconi, Mark, Qian, Xiaolei and Riemenschneider, R. A., Correct Architecture Refinement. IEEE Transactions on Software Engineering, 21 1995:4, pp. 356–372.

Nyfjord, Jaana, Towards Integrating Agile Development and Risk Management. Ph. D thesis, Stockholm University, 2008.

Oreizy, Peyman, Medvidović, Nenad and Taylor, Richard N., Runtime Software Adaptation: Framework, Approaches, and Styles. in: ICSE Companion '08: Companion of the 30th International Conference on Software Engineering. ACM, 2008, pp. 899–910.

OSGi Alliance, OSGi website. 2009 ⟨http://www.osgi.org⟩.

Ould, Martin, Business Processes - Modeling and Analysis for Re-engineering and Improvement. John Wiley and Sons, 1995.

Parnas, David, Software Fundamentals: Collected Papers by David L. Parnas. Addison-Wesley Professional, 2001, Editors: Daniel M. Hoffman and David M. Weiss.

Perry, Dewayne E. and Wolf, Alex L., Foundation for the Study of Software Architecture. ACM SIGSOFT Software Engineering Notes, 17 1992:4, pp. 40–52.

Petroski, Henry, Design Paradigms: Case Histories of Error and Judgment in Engineering. Cambridge University Press, 1994.

Polya, George, How to Solve It: A New Aspect of Mathematical Method (Princeton Science Library). Princeton University Press, 2004.

Rosch, Elanor and Lloyd, Barbara; Rosch, Elanor and Lloyd, Barbara, editors, Cognition and Categorization. Lawrence Erlbaum, 1978.

Ross, Jeanne W., Weill, Peter and Robertson, David, Enterprise Architecture as Strategy: Creating a Foundation for Business Execution. Harvard Business School Press, 2006.

Royce, Winston W., Managing the Development of Large Software Systems: Concepts and Techniques. in: Technical Papers of Western Electronic Show and Convention (WesCon). 1970.

Rozanski, Nick and Woods, Eóin, Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives. Addison-Wesley Professional, 2005.

Schmidt, Douglas et al., Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects. Wiley, 2000.

Schmidt, Douglas C. and Buschmann, Frank, Patterns, Frameworks, and Middleware: Their Synergistic Relationships. in: ICSE '03: Proceedings of the 25th International Conference on Software Engineering. Washington, DC, USA: IEEE Computer Society, 2003, pp. 694–704.

Schuh, Peter, Integrating Agile Development in the Real World. Charles River Media, 2004.

SEI Library, Software Engineering Institute Library. 2009 ⟨http://www.sei.cmu.edu/library⟩.

Selic, Bran, Brass Bubbles: An Overview of UML 2.0 (and MDA). 2003a ⟨http://www.omg.org/news/meetings/workshops/UML%202003%20Manual/Tutorial7-Hogg.pdf⟩.

Selic, Bran, The Pragmatics of Model-Driven Development. IEEE Software, 20 2003b:5, pp. 19–25.

Shaw, Mary, Abstraction, Data Types, and Models for Software. ACM SIGPLAN Notices, 16 January 1981:1, pp. 189–91.

Shaw, Mary and Clements, Paul, A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems. Proc. COMPSAC97 21st Int'l Computer Software and Applications Conference, 1997, pp. 6–13.

Shaw, Mary and Garlan, David, Software Architecture: Perspectives on an Emerging Discipline. Prentice-Hall, 1996.

Simon, Herb, The Sciences of the Artificial. 2nd edition. MIT Press, 1981.

Society, IEEE Computer, IEEE 1471-2000 IEEE Recommended Practice for Architectural Description of Software-Intensive Systems. IEEE Std 1471-2000 edition. Software Engineering Standards Committee of the IEEE Computer Society, 2000.

Sutherland, Dean, The Code of Many Colors: Semi-automated Reasoning about Multi-Thread Policy for Java. Ph. D thesis, Carnegie Mellon University Institute for Software Research, 2008, ⟨http://reports-archive.adm.cs.cmu.edu/anon/isr2008/CMU-ISR-08-112.pdf⟩.

Szyperski, Clemens, Component Software: Beyond Object-Oriented Programming. 2nd edition. Addison-Wesley Professional, 2002.

Tarr, Peri L. et al., N Degrees of Separation: Multi-Dimensional Separation of Concerns. in: International Conference on Software Engineering. 1999, pp. 107–119.

Taylor, Richard, Medvidović, Nenad and Dashofy, Eric, Software Architecture: Foundations, Theory, and Practice. Wiley, 2009.

The Open Group, TOGAF Version 9 - A Manual. 9th edition. Van Haren Publishing, 2008.

Venners, Bill, A Conversation with Martin Fowler, Part III. Artima Developer, 2002 ⟨http://www.artima.com/intv/evolutionP.html⟩.

Warmer, Jos and Kleppe, Anneke, The Object Constraint Language: Getting Your Models Ready for MDA. 2nd edition. Addison-Wesley Professional, 2003.

Whitehead, Alfred North, An Introduction to Mathematics. Forgotten Books Reprint 2009, 1911.

Wing, Jeannette M., A Study of 12 Specifications of the Library Problem. IEEE Software, 5 1988:4, pp. 66–76.

Wirfs-Brock, Rebecca, Wilkerson, Brian and Wiener, Lauren, Designing Object-Oriented Software. PTR Prentice Hall, 1990.

Wisnosky, Dennis E., DoDAF Wizdom: A Practical Guide. Wizdom Press, 2004.

Zachman, John, A Framework for Information Systems Architecture. IBM Systems Journal, 26 (3) 1987, pp. 276–292.

Index