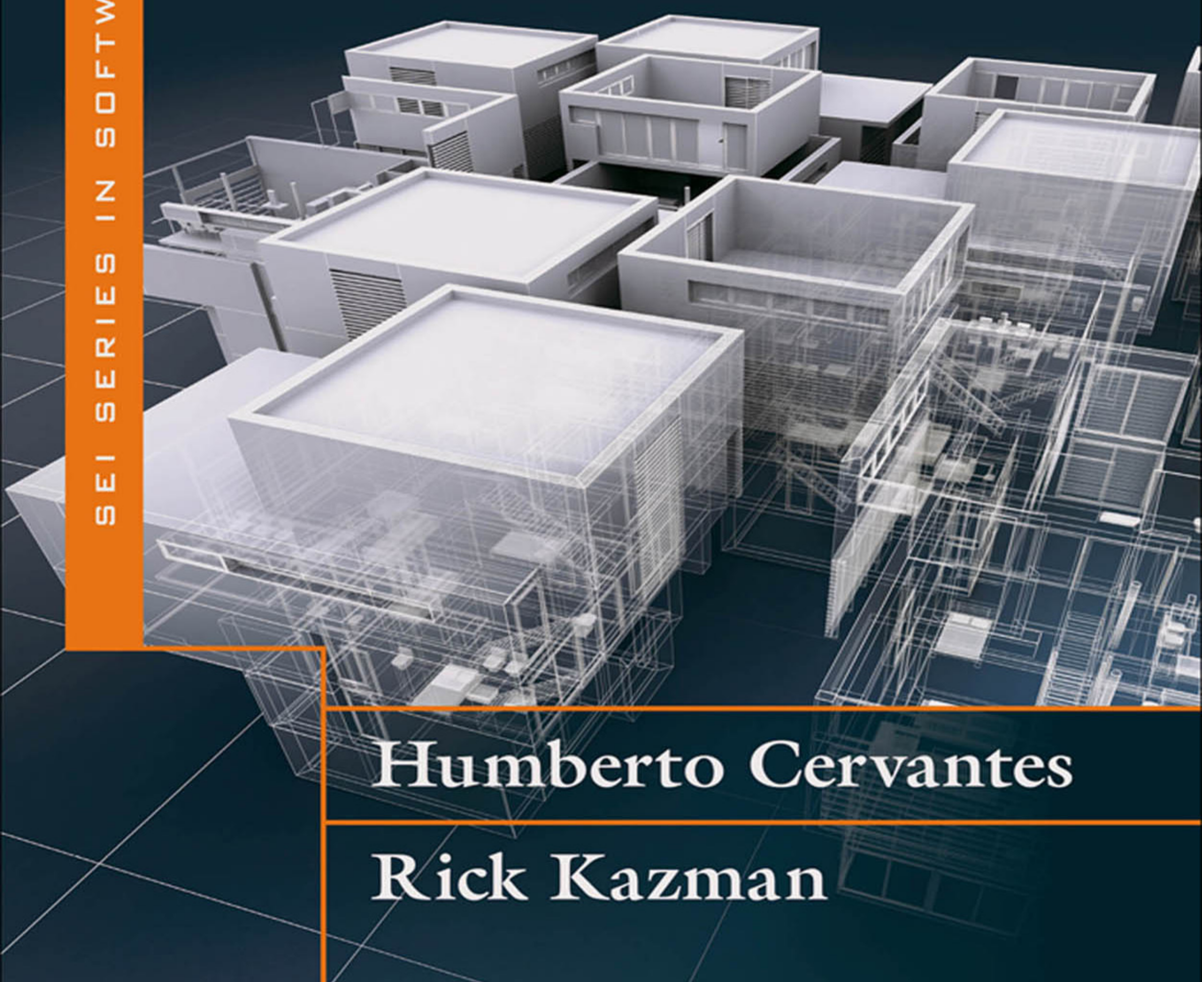
SEI SERIES IN SOFTWARE ENGINEERING

Designing Software Architectures

A Practical Approach



Humberto Cervantes

Rick Kazman

Designing Software Architectures

Designing Software Architectures

A Practical Approach

Humberto Cervantes
Rick Kazman

Addison-Wesley

Boston • Columbus • Indianapolis • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sidney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Software Engineering Institute | Carnegie Mellon

The SEI Series in Software Engineering

Library of Congress Cataloging-in-Publication Data

Names: Cervantes, Humberto, 1974- author. | Kazman, Rick, author.
Title: Designing software architectures : a practical approach / Humberto Cervantes, Rick Kazman.
Description: Boston : Addison-Wesley, [2016] | Series: The SEI series in software engineering | Includes bibliographical references and index.
Identifiers: LCCN 2016005436| ISBN 9780134390789 (hardcover : alk. paper) | ISBN 0134390784 (hardcover : alk. paper)
Subjects: LCSH: Software architecture. | Big data.
Classification: LCC QA76.758 .C44 2016 | DDC 005.1/2—dc23
LC record available at https://lccn.loc.gov/2016005436

ISBN-13: 978-013-439078-9
ISBN-10: 0-13-439078-4

Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
First printing, May 2016

Contents

I dedicate this book to my parents, Ilse and Humberto; to my wife, Gabriela; and to my sons, Julian and Alexis. Thank you for all your love, support, and inspiration.

H. C.

I dedicate this book to my wife, for her loving support, and to my Grandmasters, Hee Il Cho and Philip Ameris, for the examples that they set, leading me to always strive to be my best.

R. K.

This page intentionally left blank

Preface

When asked about software architecture, people think frequently about models—that is, the representations of the structures that constitute the architecture. Less frequently, people think about the thought processes that produce these structures—that is, the *process of design*. Design is a complex activity to perform and a complex topic to write about, as it involves making a myriad of decisions that take into account many aspects of a system. These aspects are oftentimes hard to express, particularly when they originate from experience and knowledge that is hard-earned in the "battlefield" of previous software development projects. Nevertheless, the activity of design is the basis of software architecture and, as such, it begs to be explained. Although experience can hardly be communicated through a book, what *can* be shared is a method that can help you perform the process of design in a systematic way.

This book is about that design process and about one particular design method, called Attribute-Driven Design (ADD). We believe that this method is a powerful tool that will help you perform design in a principled, disciplined, and repeatable way. In this book, employing ADD and several examples of ADD's use in the real world, we show you how to perform architectural design. Even though you may not currently possess sufficient design experience, we illustrate how the method promotes reusing *design concepts*—that is, proven solutions that embody the experience of others.

Although ADD has existed for more than a decade, relatively little has been written about it and few examples have been provided to explain how it is performed. This lack of published information has made it difficult for people to adopt the method or to teach others about it. Furthermore, the documentation that has been published about ADD is somewhat "high level" and can be hard to relate to the concepts, practices, and technologies that architects use in their day-to-day activities.

We have been working with practicing architects for several years, coaching them on how to perform design, and learning in the process. We have learned, for example, that practicing architects take technologies into consideration *early* in the design process and this is something that was not part of the original version of ADD. For this reason, the method felt "disconnected" from reality for many practitioners. In this book, we provide a revised version of ADD in which we have tried to bridge the gap between theory and practice.

We have also been teaching software architecture and software design for many years. Along the way, we realized how hard it is for people without any experience to perform design. This understanding motivated us to create a roadmap for design that, we believe, is helpful in guiding people to perform the design process. We also created a game that is useful in teaching about software design; it can be considered a companion to this book.

The target audience for this book is anyone interested in the design of software architectures. We believe it will be particularly useful for practitioners who must perform this task but who currently perform it in an ad hoc way. Experienced practitioners who already perform design following an established method will also find new ideas—for example, how to track design progress using a Kanban board, how to analyze a design using tactics-based questionnaires, and how to incorporate a design method for early estimation. Finally, people who are already familiar with the other architecture methods from the Software Engineering Institute will find information about the ways to connect ADD to methods such as the Quality Attribute Workshop (QAW), the Architecture Tradeoff Analysis Method (ATAM), and the Cost Benefit Analysis Method (CBAM). This book will also be useful to students and teachers of computer science or software engineering programs. We believe that the case studies included here will help them understand how to perform the design process more easily. Certainly, we have been using similar examples in our courses with great success. As Albert Einstein said, "Example isn't another way to teach; it is the *only* way to teach."

Our hope is that this book will help you in understanding that design can be performed following a method, and that this realization will help you produce better software systems in the future.

The book is structured as follows.

- In Chapter 1, we briefly introduce software architecture and the Attribute-Driven Design method.
- In Chapter 2, we discuss architecture design in more detail, along with the main inputs to the design process—what we call *architectural drivers,* plus the design concepts that will help you satisfy these drivers using proven solutions.
- Chapter 3 presents the ADD method in detail. We discuss each of the steps of the method along with various techniques that can be used to perform these steps appropriately.
- Chapter 4 is our first case study, which illustrates the development of a greenfield system. For this case study, we have made an effort to show how a majority of the concepts described in Chapter 3 are used in the design process, so you can think of this case study as being more "academic" in nature (although it is derived from a real-world system).
- Chapter 5 presents our second case study, which was co-written with practicing software architects and as such is much more technical and detailed

in nature. It will show you the nitty-gritty details of how ADD is used in the design of a Big Data system that involves many different technologies. This example illustrates the development of a system in what we consider to be a "novel" domain, as opposed to the more traditional domain used in Chapter 4.
- Chapter 6 is a shorter case study that illustrates the use of ADD in the design of an extension of a legacy (or brownfield) system, which is a common situation. This example demonstrates that architectural design is not something that is performed only once, when the first version of the system is developed, but rather is an activity that can be performed at different moments of the development process.
- Chapter 7 presents other design methods. In our revision of ADD, we adopted ideas from other authors who have also investigated the process of design, and here we briefly summarize their approaches both as an homage to their work and as a means to compare ADD to these methods.
- Chapter 8 discusses the topic of analysis in depth, even though this is a book on design. Analysis is naturally performed as part of design, so here we describe techniques that can be used both during the design process or after a portion of the design has been completed. In particular, we introduce the use of tactics-based questionnaires, which are helpful in understanding, in a time-efficient and simple manner, the decisions made in the design process.
- Chapter 9 describes how the design process fits at an organizational level. For instance, performing some amount of architectural design at the earliest moments of the project's life is useful for estimation purposes. We also show how ADD can be associated with different software development approaches.
- Chapter 10 concludes the book.

We also include two appendixes. Appendix A presents *A Design Concepts Catalog*, which, as its name suggests, is a catalog of different types of design concepts that can be used to design for a particular application domain. This catalog includes design concepts that we have gathered from different sources, reflecting how experienced and disciplined architects work in the real world. In this case, our catalog contains a sample of the design concepts used in the case study presented in Chapter 4. Appendix B provides a set of tactics-based questionnaires (as introduced in Chapter 8) for the seven most common quality attributes and an additional questionnaire for DevOps.

This page intentionally left blank

Acknowledgments

The authors wish to acknowledge our reviewers—Marty Barrett, Roger Champagne, Siva Muthu, Robert Nord, Vishal Prabhu, Andriy Shapochka, David Sisk, Perla Velasco-Elizondo, and Olaf Zimmermann—for their generosity in providing both opinions and comments. We also wish to thank Serge Haziyev and Olha Hrytsay for their contributions to Chapter 5. In addition, we would be remiss if we did not thank the many architects at Softserve—Serge, Olha, and Andriy included—for their overall strong support of our work.

Humberto wishes to thank the directors and the group of architects at Quarksoft; many ideas for the revision of ADD and one of the case studies presented in this book originated from putting the method into practice at this company. Thank you to the architects and developers in other companies with whom I have had the opportunity to collaborate and exchange ideas—I have learned a lot from them. I also wish to thank the people at the Software Engineering Institute, who have welcomed me and other academics for many years at the ACE Educators Workshop. I also want to give recognition to my university, Universidad Autónoma Metropolitana Iztapalapa, as it has always supported my work. Thanks to my colleagues Perla Velasco-Elizondo and Luis Castro, who have accompanied me for several years in this architectural journey. Thank you to Alonso Leal, who gave me the opportunity to become a practicing architect many years ago. Thanks to Richard S. Hall, who taught me many skills that have proved invaluable in writing this book. Finally, I wish to thank my coauthor Rick, for being such a nice person and colleague; it is always a pleasure to work and exchange opinions with him.

Rick wishes to thank James Ivers and his research group at the Software Engineering Institute. In particular, I would like to thank Rod Nord, for his careful and insightful review comments and suggestions. I would also like to thank my long-time collaborator and mentor Len Bass, who got me started on this software architecture journey many years ago. Without Len, who knows where I would be today. In addition, I would like to thank Linda Northrop, who vigorously supported my research for many years and provided many wonderful "opportunities to excel." Finally, I would like to thank my coauthor Humberto, who has always been energetic, positive, and a true pleasure to work with.

This page intentionally left blank

2

Architectural Design

We now dive into the process of architecture design: what it is, why it is important, how it works (at an abstract level). and which major concepts and activities it involves. We first discuss architectural drivers: the various factors that "drive" design decisions, some of which are documented as requirements, but many of which are not. In addition, we provide an overview of design concepts—the major building blocks that you will select, combine, instantiate, analyze, and document as part of your design process.

2.1 Design in General

Design is both a verb and a noun. Design is a process, an activity, and hence a verb. The process results in the creation of a design—a description of a desired end state. Thus the output of the design process is the thing, the noun, the artifact that you will eventually implement. Designing means making decisions to achieve goals and satisfy requirements and constraints. The outputs of the design process are a direct reflection of those goals, requirements, and constraints. Think about houses, for example. Why do traditional houses in China look different from those in Switzerland or Algeria? Why does a yurt look like a yurt, which is different from an igloo or a chalet or a longhouse?

The architectures of these styles of houses have evolved over the centuries to reflect their unique sets of goals, requirements, and constraints. Houses in

China feature symmetric enclosures, sky wells to increase ventilation, south-facing courtyards to collect sunlight and provide protection from cold north winds, and so forth. A-frame houses have steep pitched roofs that extend to the ground, meaning minimal painting and protection from heavy snow loads (which just slide off to the ground). Igloos are built of ice, reflecting the availability of ice, the relative poverty of other building materials, and the constraints of time (a small one can be built in an hour).

In each case, the process of design involved the selection and adaptation of a number of solution approaches. Even igloo designs can vary. Some are small and meant for a temporary travel shelter. Others are large, often connecting several structures, meant for entire communities to meet. Some are simple unadorned snow huts. Others are lined with furs, with ice "windows", and doors made of animal skin.

The process of design, in each case, balances the various "forces" facing the designer. Some designs require considerable skill to execute (such as carving and stacking snow blocks in such a way that they produce a self-supporting dome). Others require relatively little skill—a lean-to can be constructed from branches and bark by almost anyone. But the qualities that these structures exhibit may also vary considerably. Lean-tos provide little protection from the elements and are easily destroyed, whereas an igloo can withstand Arctic storms and support the weight of a person standing on the roof.

Is design "hard"? Well, yes and no. *Novel* design is hard. It is pretty clear how to design a conventional bicycle, but the design for the Segway broke new ground. Fortunately, most design is not novel, because most of the time our requirements are not novel. Most people want a bicycle that will reliably convey them from place to place. The same holds true in every domain. Consider houses, for example. Most people living in Phoenix want a house that can be easily and economically kept cool, whereas most people in Edmonton are primarily concerned with a house that can be kept warm. In contrast, people living in Japan and Los Angeles are concerned with buildings that can withstand earthquakes.

The good news for you, the architect, is that there are ample proven designs and design fragments, or building blocks that we call *design concepts*, that can be reused and combined to reliably achieve these goals. If your design is truly novel—if you are designing the next Sydney Opera House—then the design process will likely be "hard". The Sydney Opera House, for example, cost 14 times its original budget estimate and was delivered ten years late. So, too, with the design of software architectures.

2.2 Design in Software Architecture

Architectural design for software systems is no different than design in general: It involves making decisions, working with available skills and materials, to satisfy requirements and constraints. In architectural design, we make decisions to transform our design purpose, requirements, constraints, and architectural concerns—what we call the architectural *drivers*—into structures, as shown in Figure 2.1. These structures are then used to guide the project. They guide analysis and construction, and serve as the foundation for educating a new project member. They also guide cost and schedule estimation, team formation, risk analysis and mitigation, and, of course, implementation.

Architectural design is, therefore, a key step to achieving your product and project goals. Some of these goals are technical (e.g., achieving low and predictable latency in a video game or an e-commerce website), and some are nontechnical (e.g., keeping the workforce employed, entering a new market, meeting a deadline). The decisions that you, as an architect, make will have implications for the achievement of these goals and may, in some cases, be in conflict. The choice

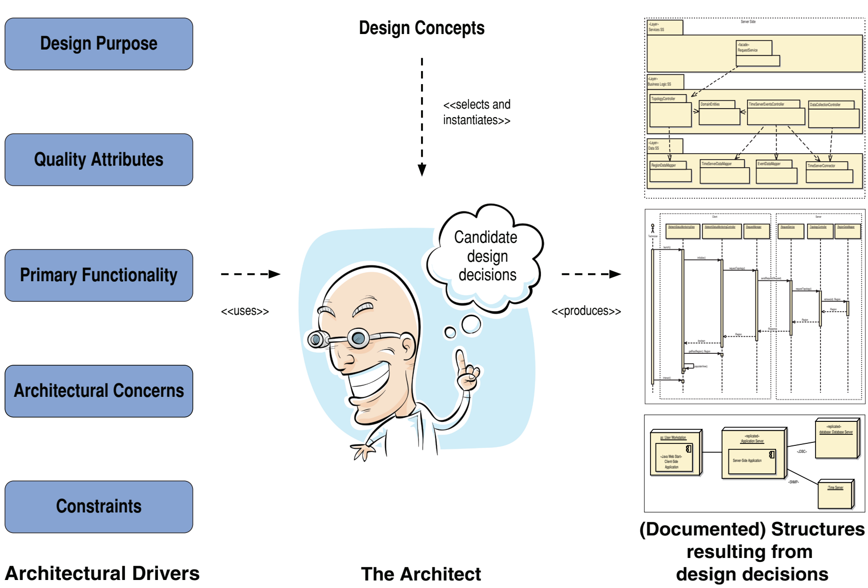


FIGURE 2.1 Overview of the architecture design activity
(Architect image © Brett Lamb | Dreamstime.com)

of a particular reference architecture (e.g., the Rich Client Application) may provide a good foundation for achieving your latency goals and will keep your workforce employed because they are already familiar with that reference architecture and its supporting technology stack. But this choice may not help you enter a new market—mobile games, for example.

In general, when designing, a change in some structure to achieve one quality attribute will have negative effects on other quality attributes. These tradeoffs are a fact of life for every practicing architect in every domain. We will see this over and over again in the examples and case studies provided in this book. Thus the architect's job is not one of finding an *optimal* solution, but rather one of *satisficing*—searching through a potentially large space of design alternatives and decisions until an acceptable solution is found.

2.2.1 Architectural Design

Grady Booch has said, "All architecture is design, but not all design is architecture". What makes a decision "architectural"? A decision is architectural if it has non-local consequences *and* those consequences matter to the achievement of an architectural driver. No decision is, therefore, inherently architectural or non-architectural. The choice of a buffering strategy within a single element may have little effect on the rest of the system, in which case it is an implementation detail that is of no concern to anyone except the implementer or maintainer of that element. In contrast, the buffering strategy may have enormous implications for performance (if the buffering affects the achievement of latency or throughput or jitter goals) or availability (if the buffers might not be large enough and information gets lost) or modifiability (if we wish to flexibly change the buffering strategy in different deployments or contexts). The choice of a buffering strategy, like most design choices, is neither inherently architectural nor inherently non-architectural. Instead, this distinction is completely dependent on the current and anticipated architectural drivers.

2.2.2 Element Interaction Design

Architectural design generally results in the identification of only a subset of the elements that are part of the system's structure. This is to be expected because, during initial architectural design, the architect will focus on the primary functionality of the system. What makes a use case primary? A combination of business importance, risk, and complexity considerations feed into this designation. Of course, to your users, everything is urgent and top priority. More realistically, a small number of use cases provide the most fundamental business value or represent the greatest risk (if they are done wrong), so these are deemed primary.

Every system has many more use cases, beyond the primary ones, that need to be satisfied. The elements that support these nonprimary use cases and their interfaces are identified as part of what we call *element interaction design*. This level of design usually follows architectural design. The location and relationships of these elements, however, are constrained by the decisions that were made during architectural design. These elements can be units of work (i.e., modules) assigned to an individual or to a team, so this level of design is important for defining not only how nonprimary functionality is allocated, but also for planning purposes (e.g., team formation and communication, budgeting, outsourcing, release planning, unit and integration test planning).

Depending on the scale and complexity of the system, the architect should be involved in element interaction design, either directly or in an auditing role. This involvement ensures that the system's important quality attributes are not compromised—for example, if the elements are not defined, located, and connected correctly. It will also help the architect spot opportunities for generalization.

2.2.3 Element Internals Design

A third level of design follows element interaction design, which we call *element internals design*. In this level of design, which is usually conducted as part of the element development activities, the internals of the elements identified in the previous design level are established, so as to satisfy the element's interface.

Architectural decisions can and do occur at the three levels of design. Moreover, during architectural design, the architect may need to delve as deeply as element internals design to achieve a particular architectural driver. An example of this is the selection of a buffering strategy that was previously discussed. In this sense, architectural design can involve considerable detail, which explains why we do not like to think about it in terms of "high-level design" or "detailed design" (see the sidebar "Detailed Design?").

Architectural design precedes element interaction design, which precedes element internals design. This is logically necessary: One cannot design an element's internals until the elements themselves have been defined, and one cannot reason about interaction until several elements and some patterns of interactions among them have been defined. But as projects grow and evolve, there is, in practice, considerable iteration between these activities.

Detailed Design?

The term "detailed design" is often used to refer to the design of the internals of modules. Although it is widely used, we really don't like this term, which is presented as somehow in opposition to "high-level design". We prefer the more precise terms "architectural design", "element interaction design", and "element internals design".

After all, architectural design may be quite detailed, if your system is complex. And some design "details" will turn out to be architectural. For the same reason, we also don't like the terms "high-level design" and "low-level design". Who can really know what these terms actually mean? Clearly, "high-level design" should be somehow "higher" or more abstract, and cover more of the architectural landscape than "low-level design", but beyond that we are at a loss to imbue these terms with any precise meaning.

So here is what we recommend: Just avoid using terms such as "high", "low", or "detailed" altogether. There is always a better, more precise choice, such as "architectural", "element interaction", or "element internals" design!

Think carefully about the impact of the decisions you are making, the information that you are trying to convey in your design documentation, and the likely audience for that information, and then give that process an appropriate, meaningful name.

2.3 Why Is Architectural Design So Important?

There is a very high cost to a project of *not* making certain design decisions, or of not making them early enough. This manifests itself in many different ways. Early on, an initial architecture is critical for project proposals (or, as it is sometimes called in the consulting world, the *pre-sales process*). Without doing some architectural thinking and some early design work, you cannot confidently predict project cost, schedule, and quality. Even at this early stage, an architecture will determine the key approaches for achieving architectural drivers, the gross work-breakdown structure, and the choices of tools, skills, and technologies needed to realize the system.

In addition, architecture is a key enabler of agility, as we will discuss in Chapter 9. Whether your organization has embraced Agile processes or not, it is difficult to imagine anyone who would willingly choose an architecture that is brittle and hard to change or extend or tune—and yet it happens all the time. This so-called *technical debt* occurs for a variety of reasons, but paramount among these is the combination of a focus on features—typically driven by stakeholder demands—and the inability of architects and project managers to measure the return on investment of good architectural practices. Features provide immediate benefit. Architectural improvement provides immediate costs and long-term benefits. Put this way, why would anyone ever "invest" in architecture? The answer is simple: Without architecture, the benefits that the system is supposed to bring will be far harder to realize.

Simply put, if you do not make some key architectural decisions early and if you allow your architecture to degrade, you will be unable to maintain sprint velocity, because you cannot easily respond to change requests. However, we vehemently disagree with what the original creators of the Agile Manifesto claimed: "The best architectures, requirements, and designs emerge from self-organizing teams". Indeed, our demurral with this point is precisely why we have written this book. Good architectural design is difficult (and still rare), and it does not just "emerge". This opinion mirrors a growing consensus within the Agile community. More and more, we see techniques such as "disciplined agility at scale", the "walking skeleton", and the "scaled Agile framework" embraced by Agile thought leaders and practitioners alike. Each of these techniques advocates some architectural thinking and design prior to much, if any, development. To reiterate, architecture enables agility, and not the other way around.

Furthermore, the architecture will influence, but not determine, other decisions that are not in and of themselves design decisions. These decisions do not influence the achievement of quality attributes directly, but they may still need to be made by the architect. For example, such decisions may include selection of tools; structuring the development environment; supporting releases, deployment, and operations; and making work assignments.

Finally, a well-designed, properly communicated architecture is key to achieving *agreements* that will guide the team. The most important kinds to make are agreements on interfaces and on shared resources. Agreeing on interfaces early is important for component-based development, and critically important for distributed development. These decisions *will* be made sooner or later. If you don't make the decisions early, the system will be much more difficult to integrate. In Section 3.6, we will discuss how to define interfaces as part of architectural design—both the external interfaces to other systems and the internal interfaces that mediate your element interactions.

2.4 Architectural Drivers

Before commencing design with ADD (or with any other design method, for that matter), you need to think about what you are doing and why. While this statement may seem blindingly obvious, the devil is, as usual, in the details. We categorize these "what" and "why" questions as architectural drivers. As shown in Figure 2.1, these drivers include a design purpose, quality attributes, primary functionality, architectural concerns, and constraints. These considerations are critical to the success of the system and, as such, they *drive* and shape the architecture.

As with any other important requirements, architectural drivers need to be baselined and managed throughout the development life cycle.

2.4.1 Design Purpose

First, you need to be clear about the purpose of the design that you want to achieve. When and why are you doing this architecture design? Which business goals is the organization most concerned about at this time?

1. You may be doing architecture design as part of a project proposal (for the pre-sales process in a consulting organization, or for internal project selection and prioritization in a company, as discussed in Section 9.1.1). It is not uncommon that, as part of determining project feasibility, schedule, and budget, an initial architecture is created. Such an architecture would not be very detailed; its purpose is to understand and break down the architecture in sufficient detail that the units of work are understood and hence may be estimated.
2. You may be doing architecture design as part of the process of creating an exploratory prototype. In this case, the purpose of the architecture design process is not so much to create a releasable or reusable system, but rather to explore the domain, to explore new technology, to place something executable in front of a customer to elicit rapid feedback, or to explore some quality attribute (such as performance scalability or failover for availability).
3. You may be designing your architecture during development. This could be for an entire new system, for a substantial portion of a new system, or for a portion of an existing system that is being refactored or replaced. In this case, the purpose is to do enough design work to satisfy requirements, guide system construction and work assignments, and prepare for an eventual release.

These purposes may be interpreted and realized differently for greenfield systems in mature domains, for greenfield systems in novel domains, and for existing systems. In a mature domain, the pre-sales process, for example, might be relatively straightforward; the architect can reuse existing systems as examples and confidently make estimates based on analogy. In novel domains, the pre-sales estimation process will be far more complex and risky, and may have highly variable results. In these circumstances, a prototype of the system, or a key part of the system, may need to be created to mitigate risk and reduce uncertainty. In many cases, this architecture may also need to be quickly adapted as new requirements are learned and embraced. In brownfield systems, while the requirements are better understood, the existing system is itself a complex object that must be well understood for planning to be accurate.

Finally, the development organization's goals during development or maintenance may affect the architecture design process. For example, the organization might be interested in designing for reuse, designing for future extension or subsetting, designing for scalability, designing for continuous delivery, designing to best utilize existing project capabilities and team member skills, and so forth. Or the organization might have a strategic relationship with a vendor. Or the CIO might have a specific like or dislike and wants to impose it on your project.

Why do we bother to list these considerations? Because they *will* affect both the process of design and the outputs of design. Architectures exist to help achieve business goals. The architect should be clear about these goals and should communicate them (and negotiate them!) and establish a clear design purpose *before* beginning the design process.

2.4.2 Quality Attributes

In the book *Software Architecture in Practice*, *quality attributes* are defined as being measurable or testable properties of a system that are used to indicate how well the system satisfies the needs of its stakeholders. Because quality tends to be a subjective concept in itself, these properties allow quality to be expressed succinctly and objectively.

Among the drivers, quality attributes are the ones that shape the architecture the most significantly. The critical choices that you make when you are doing architectural design determine, in large part, the ways that your system will or will not meet these driving quality attribute goals.

Given their importance, you must worry about eliciting, specifying, prioritizing, and validating quality attributes. Given that so much depends on getting these drivers right, this sounds like a daunting task. Fortunately, a number of well-understood, widely disseminated techniques can help you here (see sidebar "The Quality Attribute Workshop and the Utility Tree"):

- Quality Attribute Workshop (QAW) is a facilitated brainstorming session involving a group of system stakeholders that covers the bulk of the activities of eliciting, specifying, prioritizing, and achieving consensus on quality attributes.
- Mission Thread Workshop serves the same purpose as QAW, but for a system of systems.
- The Utility Tree can be used by the architect to prioritize quality attribute requirements according to their technical difficulty and risk.

We believe that the best way to discuss, document, and prioritize quality attribute requirements is as a set of scenarios. A *scenario*, in its most basic form, describes the system's response to some stimulus. Why are scenarios the best approach? Because all other approaches are worse! Endless time may be wasted in defining terms such as "performance" or "modifiability" or "configurability", as these discussions tend to shed little light on the real system. It is meaningless to say that a system will be "modifiable", because every system is modifiable with respect to some changes and not modifiable with respect to others. One can, however, specify the modifiability response measure you would like to achieve (say, elapsed time or effort) in response to a specific change request. For example, you might want to specify that "a change to update shipping rates on the e-commerce website is completed and tested in less than 1 person-day of effort"—an unambiguous criterion.

The heart of a quality attribute scenario, therefore, is the pairing of a stimulus with a response. Suppose that you are building a video game and you have a functional requirement like this: "The game shall change view modes when the user presses the <C> button". This functional requirement, if it is important, needs to be associated with quality attribute requirements. For example:

- How fast should the function be?
- How secure should the function be?
- How modifiable should the function be?

To address this problem, we use a scenario to describe a quality attribute requirement. A quality attribute scenario is a short description of how a system is required to respond to some stimulus. For example, we might annotate the functional requirement given earlier as follows: "The game shall change view modes in < 500 ms when the user presses the <C> button". A scenario associates a stimulus (in this case, the pressing of the <C> button) with a response (changing the view mode) that is measured using a response measure (< 500 ms). A complete quality attribute scenario adds three other parts: the source of the stimulus (in this case, the user), the artifact affected (in this case, because we are dealing with end-to-end latency, the artifact is the entire system) and the environment (are we in normal operation, startup, degraded mode, or some other mode?). In total, then, there are six parts of a completely well-specified scenario, as shown in Figure 2.2.

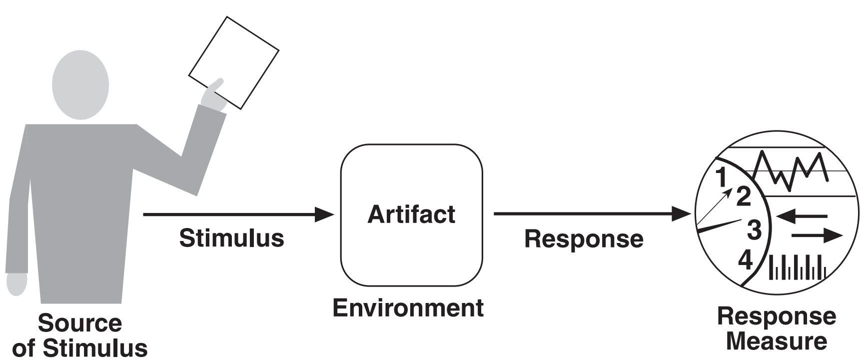


FIGURE 2.2 The six parts of a quality attribute scenario

Scenarios are testable, *falsifiable hypotheses* about the quality attribute behavior of the system under consideration. Because they have explicit stimuli and responses, we can evaluate a design in terms of how likely it is to support the scenario, and we can take measurements and test a prototype or fully fleshed-out system for whether it satisfies the scenario in practice. If the analysis (or prototyping results) indicates that the scenario's response goal cannot be met, then the hypothesis is deemed falsified.

As with other requirements, scenarios should be prioritized. This can be achieved by considering two dimensions that are associated with each scenario and that are assigned a rank of importance:

- The first dimension corresponds to the importance of the scenario with respect to the success of the system. This is ranked by the customer.
- The second dimension corresponds to the degree of technical risk associated with the scenario. This is ranked by the architect.

A low/medium/high (L/M/H) scale is used to rank both dimensions. Once the dimensions have been ranked, scenarios are prioritized by selecting those that have a combination of (H, H), (H, M), or (M, H) rankings.

In addition, some traditional requirements elicitation techniques can be modified slightly to focus on quality attribute requirements, such as Joint Requirements Planning (JRP), Joint Application Design (JAD), discovery prototyping, and accelerated systems analysis.

But whatever technique you use, *do not* start design without a prioritized list of measurable quality attributes! While stakeholders might plead ignorance ("I don't know how fast it needs to be; just make it fast!"), you can almost always elicit at least a range of possible responses. Instead of saying the system should be "fast", ask the stakeholder if a 10-second response time is acceptable. If that is unacceptable, ask if 5 seconds is OK, or 1 second. You will find that, in most cases, users know more than they realize about their requirements, and you can at least "box them in" to a range.

The Quality Attribute Workshop and the Utility Tree

The Quality Attribute Workshop (QAW)

The QAW is a facilitated, stakeholder-focused method to generate, prioritize, and refine quality attribute scenarios. A QAW meeting is ideally enacted before the software architecture has been defined although, in practice, we have seen the QAW being used at all points in the software development life cycle. The QAW is focused on system-level concerns and specifically the role that software will play in the system. The steps of the QAW are as follows:

1. QAW Presentation and Introductions

   The QAW facilitators describe the motivation for the QAW and explain each step of the method.

2. Business Goals Presentation

   A stakeholder representing the project's business concerns presents the system's business context, broad functional requirements, constraints, and known quality attribute requirements. The quality attributes that will be refined in later QAW steps will be derived from, and should be traceable to, the business goals presented in this step. For this reason, these business goals must be prioritized.

3. Architectural Plan Presentation

   The architect presents the system architectural plans as they currently exist. Although the architecture has frequently not been defined yet (particularly for greenfield systems), the architect often knows quite a lot about it even at this early stage. For example, the architect might already know about technologies that are mandated, other systems that this system must interact with, standards that must be followed, subsystems or components that could be reused, and so forth.

4. Identification of Architectural Drivers

   The facilitators share their list of key architectural drivers that they assembled during steps 2 and 3 and ask the stakeholders for clarifications, additions, deletions, and corrections. The idea here is to reach a consensus on a distilled list of architectural drivers that covers major functional requirements, business drivers, constraints, and quality attributes.

5. Scenario Brainstorming

   Given this context, each stakeholder now has the opportunity to express a scenario representing that stakeholder's needs and desires with respect to the system. The facilitators ensure that each scenario has an explicit stimulus and response. The facilitators also ensure traceability and completeness: At least one representative scenario should exist for each architectural driver listed in step 4 and should cover all the business goals listed in step 2.

6. Scenario Consolidation

   Similar scenarios are consolidated where reasonable. In step 7, the stakeholders vote for their favorite scenarios, and consolidation helps to prevent votes from being spread across several scenarios that are expressing essentially the same concern.

7. Scenario Prioritization

   Prioritization of the scenarios is accomplished by allocating to each stakeholder a number of votes equal to 30 percent of the total number of scenarios. The stakeholders can distribute these votes to any scenario or scenarios. Once all the stakeholders have voted, the results are tallied and the scenarios are sorted in order of popularity.

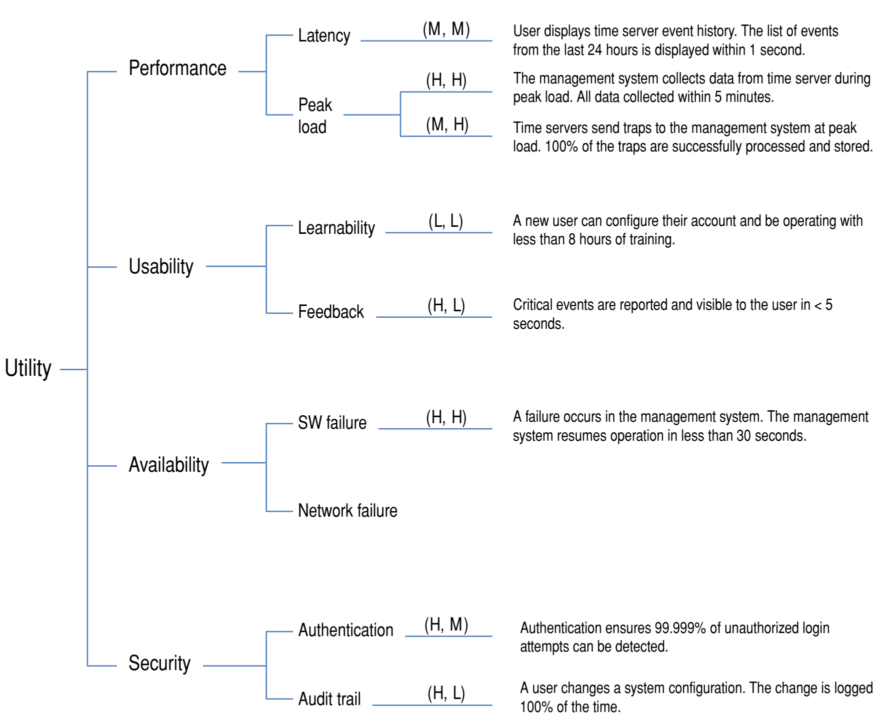
8. Scenario Refinement

   The highest-priority scenarios are refined and elaborated. The facilitators help the stakeholders express these in the form of six-part scenarios: source, stimulus, artifact, environment, response, and response measure.

The output of the QAW is therefore a prioritized list of scenarios, aligned with business goals, where the highest-priority scenarios have been explored and refined. A QAW can be conducted in as little as 2–3 hours for a simple system or as part of an iteration, and as much as 2 days for a complex system where requirements completeness is a goal.

Utility Tree

If no stakeholders are readily available to consult, you still need to decide what to do and how to prioritize the many challenges facing the system. One way to organize your thoughts is to create a Utility Tree. The Utility Tree, such as the one shown in the following figure, helps to articulate your quality attribute goals in detail, and then to prioritize them.



It works as follows. First write the word “Utility” on a sheet of paper. Then write the various quality attributes that constitute utility for your system. For example, you might know, based on the business goals for the system, that the most important qualities for the system are that the system be fast, secure, and easy to modify. In turn, you would write these words underneath “Utility”. Next, because we don’t really know what any of those terms actually means, we describe the aspect of the quality attribute that we are most concerned with. For example, while “performance” is vague, “latency of database transactions” is a bit less vague. Likewise, while “modifiability” is vague, “ease of adding new codecs” is a bit less vague.

The leaves of the tree are expressed as scenarios, which provide concrete examples of the quality attribute considerations that you just enumerated. For example, for “latency of database transactions”, you might create a scenario such as “1000 users simultaneously update their own customer records under normal conditions with an average latency of 1 second”. For “ease of adding new codecs”, you might create a scenario such as “Customer requests that a new custom codec be added to the system. Codec is added with no side effects in 2 person-weeks of effort”.

Finally, the scenarios that you have created must be prioritized. We do this prioritization by using the technique of ranking across two dimensions, resulting in a priority matrix such as the following (where the numbers in the cells are from a set of system scenarios).

Business Importance/ Technical Risk	**L**	**M**	**H**
L	5, 6, 17, 20, 22	1, 14	12, 19
M	9, 12, 16	8, 20	3, 13, 15
H	10, 18, 21	4, 7	2, 11

Our job, as architects, is to focus on the lower-right-hand portion of this table (H, H): those scenarios that are of high business importance and high risk. Once we have satisfactorily addressed those scenarios, we can move to the (M, H) or (H, M) ones, and then move up and to the left until all of the system’s scenarios are addressed (or perhaps until we run out of time or budget, as is often the case).

It should be noted that the QAW and the Utility Tree are two different techniques that are aimed at the same goal—eliciting and prioritizing the most important quality attribute requirements, which will be some of your most critical architectural drivers. There is no reason, however, to choose between these techniques. Both are useful and valuable and, in our experience, they have complementary strengths: The QAW tends to focus more on the requirements of external stakeholders, whereas the Utility Tree tends to excel at eliciting the requirements of internal stakeholders. Making all of these stakeholders happy will go a long way toward ensuring the success of your architecture.

2.4.3 Primary Functionality

Functionality is the ability of the system to do the work for which it was intended. As opposed to quality attributes, the way the system is structured does not normally influence functionality. You can have all of the functionality of a given system coded in a single enormous module, or you can have it neatly distributed across many smaller, highly cohesive modules. Externally the system will look and work the same way if you consider only functionality. What matters, though, is what happens when you want to make changes to such system. In the former case, changes will be difficult and costly; in the latter case, they should be much easier and cheaper to perform. In terms of architectural design, allocation of functionality to elements, rather than the functionality per se, is what matters. A good architecture is one in which the most common changes are localized in a single or a few elements, and hence easy to make.

When designing an architecture, you need to consider at least the primary functionality. Primary functionality is usually defined as functionality that is critical to achieve the business goals that motivate the development of the system. Other criteria for primary functionality might be that it implies a high level of technical difficulty or that it requires the interaction of many architectural elements. As a rule of thumb, approximately 10 percent of your use cases or user stories are likely to be primary.

There are two important reasons why you need to consider primary functionality when designing an architecture:

1. You need to think how functionality will be allocated to elements (usually modules) to promote modifiability or reusability, and also to plan work assignments.
2. Some quality attribute scenarios are directly connected to the primary functionality in the system. For example, in a movie streaming application, one of the primary use cases is, of course, to watch a movie. This use case is associated with a performance quality attribute scenario such as "Once the user presses play, the movie should begin streaming in no more than *5* seconds". In this case, the quality attribute scenario is directly associated with the primary use case, so making decisions to support this scenario also requires making decisions about how its associated functionality will be supported. This is not the case for all quality attributes. For example, an availability scenario can involve recovery from a system failure, and this failure may occur when any of the system's use cases are being executed.

Decisions regarding the allocation of functionality that are made during architectural design establish a precedent for how the rest of the functionality should be allocated to modules as development progresses. This is usually not the work of the architect; instead, this activity is typically performed as part of the element interaction design process described in Section 2.2.2.

Finally, bad decisions that are made regarding the allocation of functionality result in the accumulation of technical debt. (Of course, these decisions may reveal themselves to be bad only in hindsight.) This debt can be paid through the use of refactoring, although this impacts the project's rate of progress, or velocity (see the sidebar "Refactoring").

Refactoring

If you refactor a software architecture (or part of one), what you are doing is maintaining the same functionality but changing some quality attribute that you care about. Architects often choose to refactor because a portion of the system is difficult to understand, debug, and maintain. Alternatively, they may refactor because part of the system is slow, or prone to failure, or insecure.

The goal of the refactoring in each case is not to change the functionality, but rather to change the quality attribute response. (Of course, additions to functionality are sometimes lumped together with a refactoring exercise, but that is not the core *intent* of the refactoring.) Clearly, if we can maintain the same functionality but change the architecture to achieve different quality attribute responses, these requirement types are orthogonal to each other—that is, they can vary independently.

2.4.4 Architectural Concerns

Architectural concerns encompass additional aspects that need to be considered as part of architectural design but that are not expressed as traditional requirements. There are several different types of concerns:

- *General concerns.* These are "broad" issues that one deals with in creating the architecture, such as establishing an overall system structure, the allocation of functionality to modules, the allocation of modules to teams, organization of the code base, startup and shutdown, and supporting delivery, deployment, and updates.
- *Specific concerns.* These are more detailed system-internal issues such as exception management, dependency management, configuration, logging, authentication, authorization, caching, and so forth that are common across large numbers of applications. Some specific concerns are addressed in reference architectures (see Section 2.5.1), but others will be unique to your system. Specific concerns also result from previous design decisions. For example, you may need to address session management if you previously decided to use a reference architecture for the development of web applications.

- *Internal requirements.* These requirements are usually not specified explicitly in traditional requirement documents, as customers usually seldom express them. Internal requirements may address aspects that facilitate development, deployment, operation, or maintenance of the system. They are sometimes called "derived requirements".
- *Issues.* These result from analysis activities, such as a design review (see Section 8.6), so they may not be present initially. For instance, an architectural evaluation may uncover a risk that requires some changes to be performed in the current design.

Some of the decisions surrounding architectural concerns might be trivial or obvious. For example, your deployment structure might be a single processor for an embedded system, or a single cell phone for an app. Your reference architecture might be constrained by company policy. Your authentication and authorization policies might be dictated by your enterprise architecture and realized in a shared framework. In other cases, however, the decisions required to satisfy particular concerns may be less obvious—for example, in exception management or input validation or structuring the code base.

From their past experience, wise architects are usually aware of the concerns that are associated with a particular type of system and the need to make design decisions to address them. Inexperienced architects are usually less aware of such concerns; because these concerns tend to be tacit rather than explicit, they may not consider them as part of the design process, which often results in problems later on.

Architectural concerns frequently result in the introduction of new quality attribute scenarios. The concern of "supporting logging", for example, is too vague and needs to be made more specific. Like the quality attribute scenarios that are provided by the customer, these scenarios need to be prioritized. For these scenarios, however, the customer is the development team, operations, or other members of the organization. During design, the architect must consider both the quality attribute scenarios that are provided by the customer and those scenarios that are derived from architectural concerns.

One of the goals of our revision of the ADD method was to elevate the importance of architectural concerns as explicit inputs to the architecture design process, as will be highlighted in our examples and case studies in Chapters 4, 5, and 6.

2.4.5 Constraints

You need to catalog the constraints on development as part of the architectural design process. These constraints may take the form of mandated technologies, other systems with which your system needs to interoperate or integrate, laws and standards that must be complied with, the abilities and availability of your developers, deadlines that are non-negotiable, backward compatibility with older versions of systems, and so on. An example of a technical constraint is the use of open source technologies, whereas a nontechnical constraint is that the system must obey the Sarbanes-Oxley Act or that it must be delivered by December 15.

A constraint is a decision over which you have little or no control as an architect. Your job is, as we mentioned in Chapter 1, to *satisfice*: to design the best system that you can, despite the constraints you face. Sometimes you might be able to argue for loosening a constraint, but in most cases you have no choice but to design around the constraints.

2.5 Design Concepts: The Building Blocks for Creating Structures

Design is not random, but rather is planned, intentional, rational, and directed. The process of design may seem daunting at first. When facing the "blank page" at the beginning of any design activity, the space of possibilities might seem impossibly huge and complex. However, there is some help here. The software architecture community has created and evolved, over the course of decades, a body of generally accepted design principles that can guide us to create high-quality designs with predictable outcomes.

For example, some well-documented design principles are oriented toward the achievement of specific quality attributes:

- To help achieve high modifiability, aim for good modularity, which means high cohesion and low coupling.
- To help achieve high availability, avoid having any single point of failure.
- To help achieve scalability, avoid having any hard-coded limits for critical resources.
- To help achieve security, limit the points of access to critical resources.
- To help achieve testability, externalize state.
- . . . and so forth.

In each case, these principles have been evolved over decades of dealing with those quality attributes in practice. In addition, we have evolved reusable realizations of these abstract approaches in design and, eventually, in code. We call these reusable realizations *design concepts*, and they are the building blocks from which the structures that make up the architecture are created. Different types of design concepts exist, and here we discuss some of the most commonly used, including reference architectures, deployment patterns, architectural patterns, tactics, and externally developed components (such as frameworks). While the first four are conceptual in nature, the last one is concrete.

2.5.1 Reference Architectures

Reference architectures are blueprints that provide an overall logical structure for particular types of applications. A reference architecture is a reference model mapped onto one or more architectural patterns. It has been proven in business and technical contexts, and typically comes with a set of supporting artifacts that eases its use.

An example of a reference architecture for the development of web applications is shown in Figure 2.3 on the next page. This reference architecture establishes the main layers for this type of application—presentation, business, and data—as well as the types of elements that occur within the layers and the responsibilities of these elements, such as UI components, business components, data access components, service agents, and so on. Also, this reference architecture introduces cross-cutting concerns, such as security and communication, that need to be addressed. As this example shows, when you select a reference architecture for your application, you also adopt a set of issues that you need to address during design. You may not have an explicit requirement related to communications or security, but the fact that these elements are part of the reference architecture require you to make design decisions about them.

Reference architectures may be confused with architectural styles, but these two concepts are different. Architectural styles (such as “Pipe and Filter” and “Client Server”) define types of components and connectors in a specified topology that are useful for structuring an application either logically or physically. Such styles are technology and domain agnostic. Reference architectures, in contrast, provide a structure for applications in specific domains, and they may embody different styles. Also, while architectural styles tend to be popular in academia, reference architectures seem to be preferred by practitioners—which is also why we favor them in our list of design concepts.

While there are many reference architectures, we are not aware of any catalog that contains an extensive list of them.

2.5.2 Architectural Design Patterns

Design patterns are conceptual solutions to recurring design problems that exist in a defined context. While design patterns originally focused on decisions at the object scale, including instantiation, structuring, and behavior, today there are catalogs with patterns that address decisions at varying levels of granularity. In addition, there are specific patterns to address quality attributes such as security or integration.

While some people argue for the differentiation between what they consider to be architectural patterns and the more fine-grained design patterns, we believe there is no principled difference that can be solely attributed to scale. We consider a pattern to be architectural when its use directly and substantially influences the satisfaction of some of the architectural drivers (see Section 2.2).

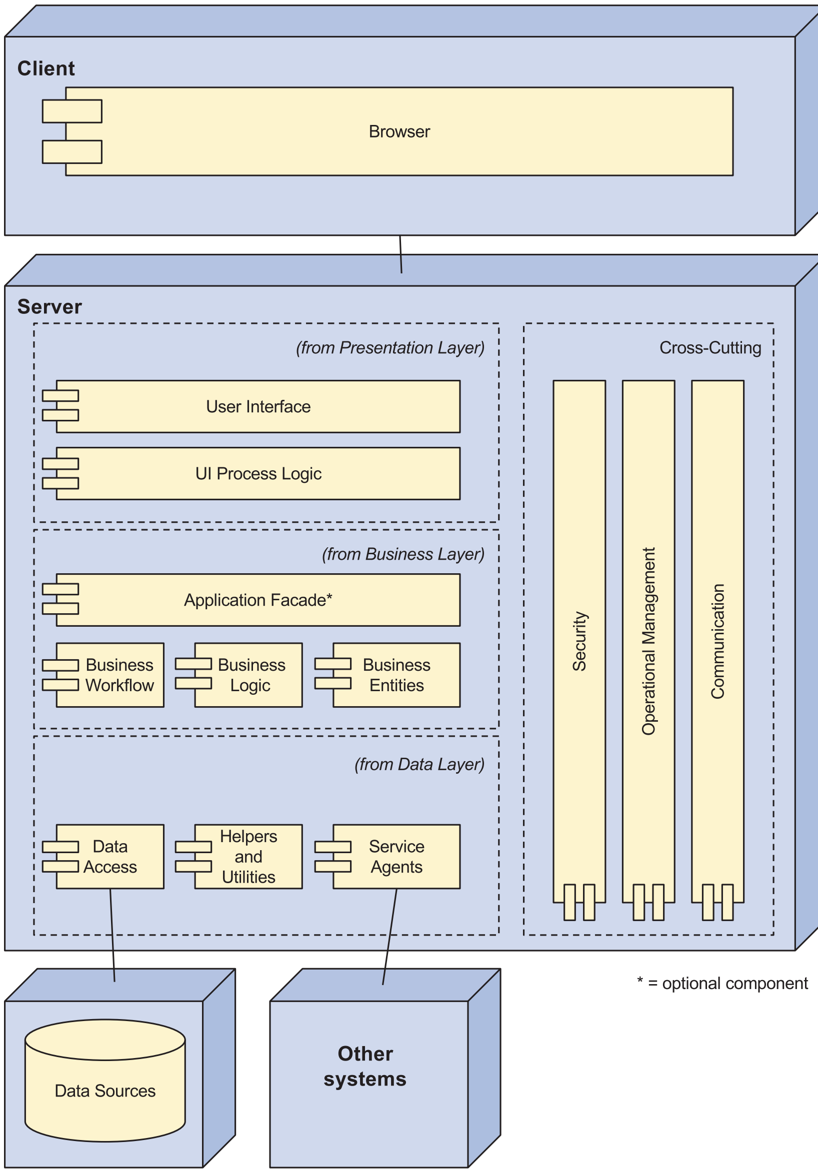


FIGURE 2.3 Example reference architecture for the development of web applications from the *Microsoft Application Architecture Guide* (Key: UML)

Figure 2.4 shows an example architectural pattern that is useful for structuring the system, the Layers pattern. When you choose a pattern such as this one, you must decide how many layers you will need for your system. Figure 2.5 shows a pattern to support concurrency, which is useful to increase performance. This pattern, too, needs to be instantiated—that is, it needs to be adapted to the specific problem and design context. Instantiation is discussed in Chapter 3.

Although reference architectures may be considered as a type of pattern, we prefer to consider them separately because of the important role they play in structuring an application and because they are more directly connected to technology stacks. Also, a reference architecture typically incorporates other patterns and often constrains these patterns. For example, the reference architecture for web applications shown in Figure 2.3 incorporates the Layers pattern but also establishes how many layers need to be used. This reference architecture also incorporates other patterns such as an Application Facade and Data Access Components.

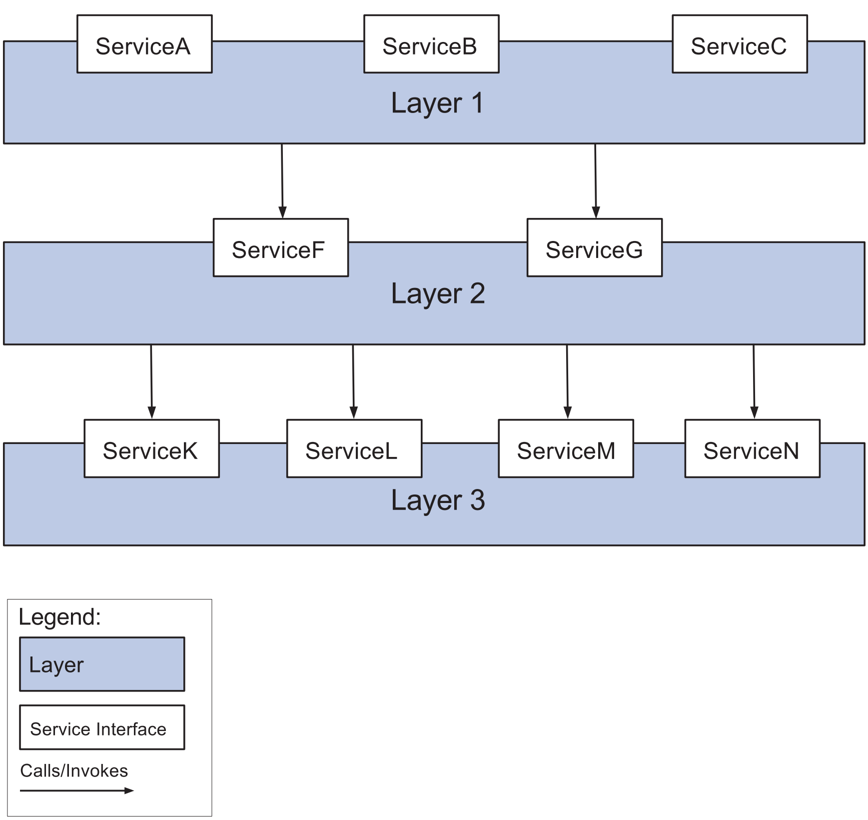


FIGURE 2.4 The Layers pattern for structuring an application from *Pattern-Oriented Software Architecture*

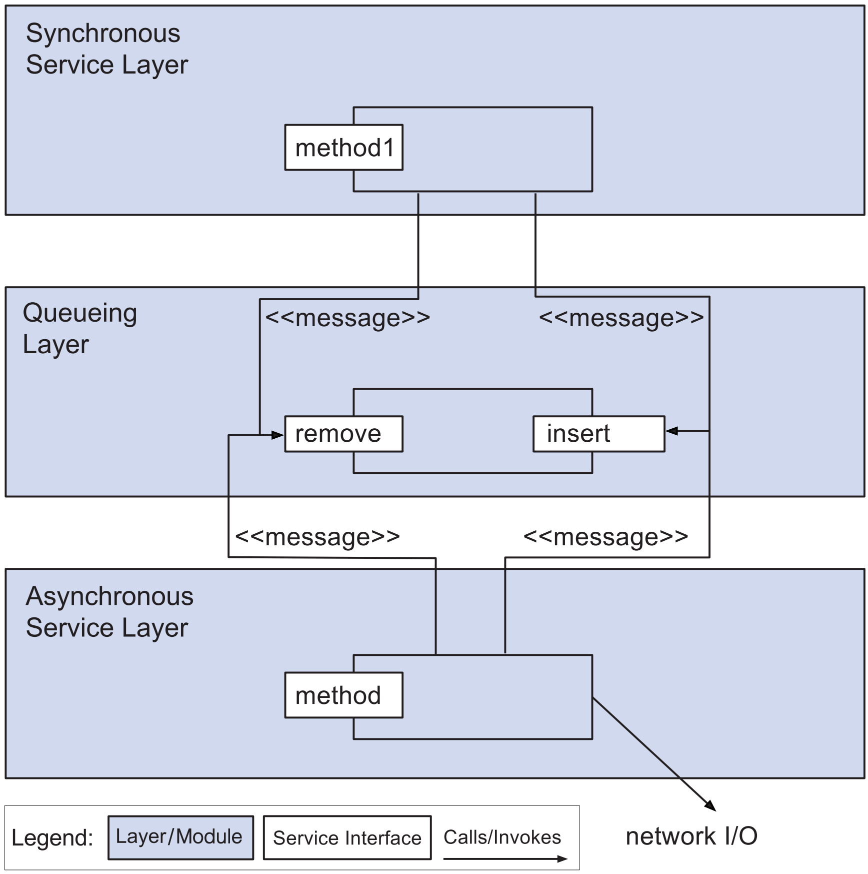


FIGURE 2.5 The Half-Sync/Half-Async pattern to support concurrency from *Pattern-Oriented Software Architecture* (Source: Softserve)

2.5.3 Deployment Patterns

Another type of pattern that we prefer to consider separately is *deployment patterns*. These patterns provide models on how to physically structure the system to deploy it. Some deployment patterns, such as the one shown in Figure 2.6, are useful to establish an initial physical structure of the system in terms of tiers (physical nodes). More specialized deployment patterns, such as the Load-Balanced Cluster in Figure 2.7, are used to satisfy quality attributes such as availability, performance, and security.

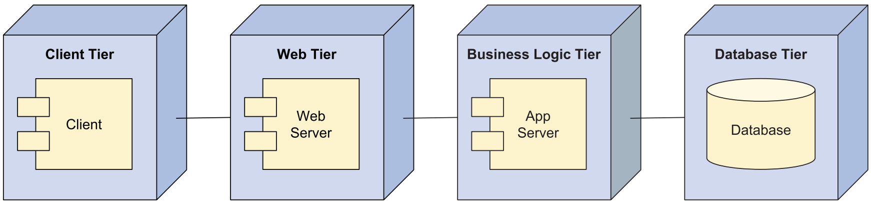


FIGURE 2.6 Four-tier deployment pattern from the *Microsoft Application Architecture Guide* (Key: UML)

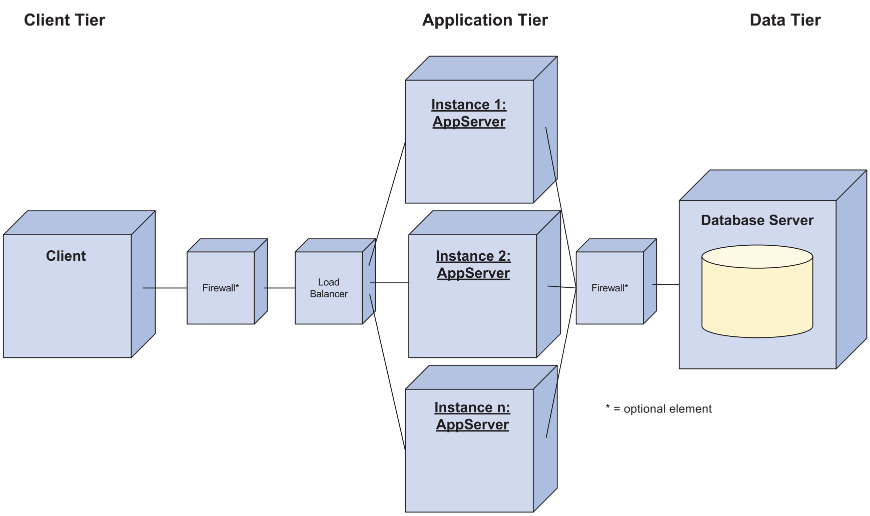


FIGURE 2.7 Load-Balanced Cluster deployment pattern for performance from the *Microsoft Application Architecture Guide* (Key: UML)

In general, an initial structure for the system is obtained by mapping the logical elements that are obtained from reference architectures (and other patterns) into the physical elements defined by deployment patterns.

2.5.4 Tactics

Architects can use collections of fundamental design techniques to achieve a response for particular quality attributes. We call these architectural design primitives *tactics*. Tactics, like design patterns, are techniques that architects have been using for years. We do not invent tactics, but simply capture what architects actually have done in practice, over the decades, to manage quality attribute response goals.

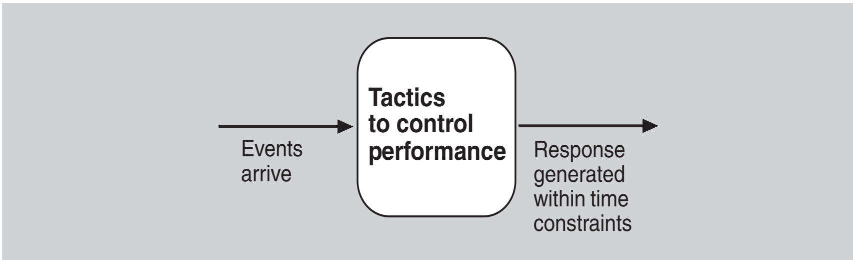


FIGURE 2.8 Tactics mediate events and responses.

Tactics are design decisions that influence the control of a quality attribute response. For example, if you want to design a system to have low latency or high throughput, you could make a set of design decisions that would mediate the arrival of events (requests for service), resulting in responses that are produced within some time constraints, as shown in Figure 2.8.

Tactics are both simpler and more primitive than patterns. They focus on the control of a single quality attribute response (although they may, of course, trade off this response with other quality attribute goals). Patterns, in contrast, typically focus on resolving and balancing multiple forces—that is, multiple quality attribute goals. By way of analogy, we can say that a tactic is an atom, whereas a pattern is a molecule.

Tactics provide a top-down way of thinking about design. A tactics categorization begins with a set of design objectives related to the achievement of a quality attribute, and presents the architect with a set of options from which to choose. These options then need to be further instantiated through some combination of patterns, frameworks, and code.

For example, in Figure 2.9, the design objectives for performance are "Control Resource Demand" and "Manage Resources". An architect who wants to create a system with "good" performance needs to choose one or more of these options. That is, the architect needs to decide if controlling resource demand is feasible, and if managing resources is feasible. In some systems, the events arriving at the system can be managed, prioritized, or limited in some way. If this is not possible, then the architect can manage resources only as part of an attempt to generate responses within acceptable time constraints. Within the "Manage Resources" category, an architect might choose to increase resources, introduce concurrency, maintain multiple copies of computations, maintain multiple copies of data, and so forth. These tactics then need to be instantiated. As an example, an architect might choose the Half-Sync/Half-Async pattern (see Figure 2.5) as a way of introducing (and managing) concurrency, or the Load-Balanced Cluster deployment pattern (see Figure 2.7) to maintain multiple copies of computations. As we will see in Chapter 3, the choice, combination, and tailoring of tactics and

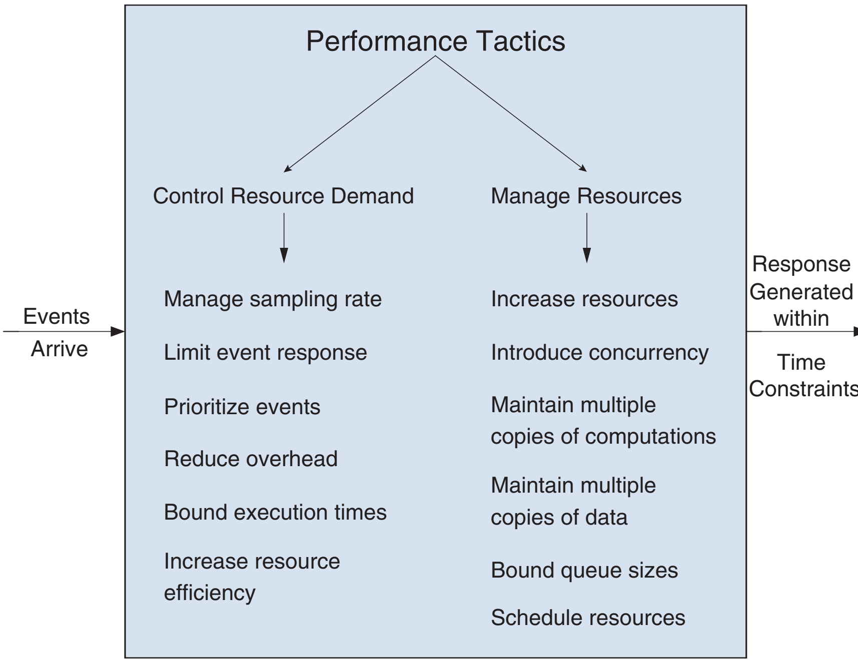


FIGURE 2.9 Performance tactics from *Software Architecture in Practice*

patterns are some of the key steps of the ADD process. There are existing tactics categorizations for the quality attributes of availability, interoperability, modifiability, performance, security, testability, and usability.

2.5.5 Externally Developed Components

Patterns and tactics are abstract in nature. However, when you are designing a software architecture, you need to make these design concepts concrete and closer to the actual implementation. There are two ways to achieve this: You can code the elements obtained from tactics and patterns or you can associate technologies with one or more of these elements in the architecture. This "buy versus build" choice is one of the most important decisions you will make as an architect.

We consider technologies to be *externally developed components*, because they are not created as part of the development project. Several types of externally developed components exist:

- *Technology families*. A technology family represents a group of specific technologies with common functional purposes. It can serve as a placeholder until a specific product or framework is selected. An example is a relational database management system (RDBMS) or an object-oriented to relational mapper (ORM). Figure 2.10 shows different technology families in the Big Data domain (in regular text).

- *Products*. A product (or software package) refers to a self-contained functional piece of software that can be integrated into the system that is being designed and that requires only minor configuration or coding. An example is a relational database management system, such as Oracle or Microsoft SQL Server. Figure 2.10 shows different products in the Big Data domain (in italics).
- *Application frameworks*. An application framework (or just framework) is a reusable software element, constructed out of patterns and tactics, that provides generic functionality addressing recurring domain and quality attribute concerns across a broad range of applications. Frameworks, when carefully chosen and properly implemented, increase the productivity of programmers. They do so by enabling programmers to focus on business logic and end-user value, rather than underlying technologies and their implementations. As opposed to products, framework functions are generally invoked from the application code or are "injected" using some type of aspect-oriented approach. Frameworks usually require extensive configuration, typically through XML files or other approaches such as annotations in Java. A framework example is Hibernate, which is used to perform object-oriented to relational mapping in Java. Several types of frameworks are available: Full-stack frameworks, such as Spring, are usually associated with reference architectures and address general concerns across the different elements of the reference architecture, while non-full-stack frameworks, such as JSF, address specific functional or quality attribute concerns.
- *Platforms*. A platform provides a complete infrastructure upon which to build and execute applications. Examples of platforms include Java, .Net, or and Google Cloud.

The selection of externally developed components, which is a key aspect of the design process, can be a challenging task because of their extensive number. Here are a few criteria you should consider when selecting externally developed components:

- *Problem that it addresses.* Is it something specific, such as a framework for object-oriented to relational mapping or something more generic, such as a platform?
- *Cost.* What is the cost of the license and, if it is free, what is the cost of support and education?
- *Type of license.* Does it have a license that is compatible with the project goals?

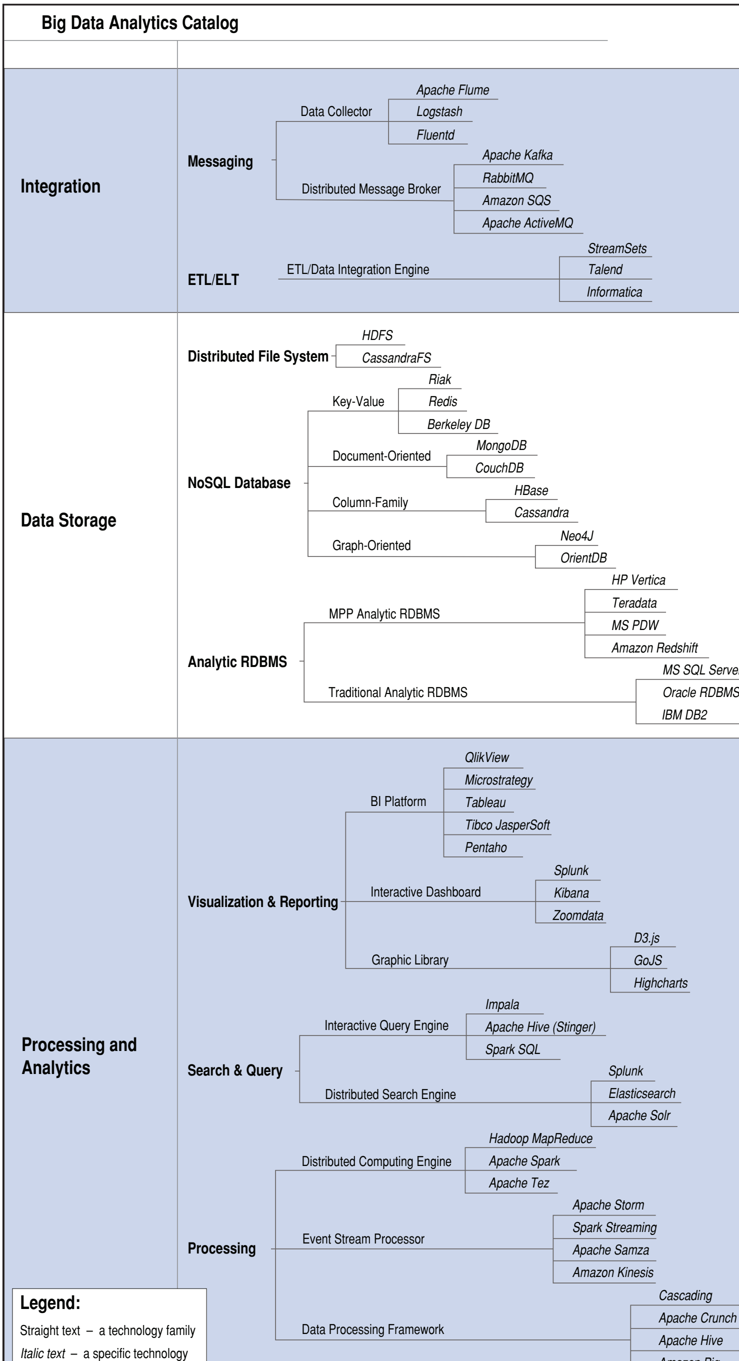


FIGURE 2.10 A technology family tree for the Big Data application domain

- *Support.* Is it well supported? Is there extensive documentation about the technology? Is there an extensive user or developer community that you can turn to for advice?
- *Learning curve.* How hard is it to learn this technology? Have others in your organization already mastered it? Are there courses available?
- *Maturity.* Is it a technology that has just appeared on the market, which may be exciting but still relatively unstable or unsupported?
- *Popularity.* Is it a relatively widespread technology? Are there positive testimonials or adoption by mature organizations? Will it be easy to hire people who have deep knowledge of it? Is there an active developer community or user group?
- *Compatibility and ease of integration.* Is it compatible with other technologies used in the project? Can it be integrated easily in the project?
- *Support for critical quality attributes.* Does it limit attributes such as performance? Is it secure and robust?
- *Size.* Will the use of the technology have a negative impact on the size of the application under development?

Unfortunately, the answers to these questions are not always easy to find and the selection of a particular technology may require you do some research or, eventually, to create prototypes that will help you in the selection process. These criteria will have a significant effect on your total cost of ownership.

2.6 Architecture Design Decisions

As we said at the beginning of this chapter, design is the process of making decisions. But the act of making a decision is a *process*, not a moment in time. Experienced architects, when faced with a design challenge, typically entertain a set of "candidate" decisions (as shown in Figure 2.1); from this set, they choose a best candidate and instantiate that. They might select this "best" candidate based on experience, constraints, or some form of analysis such as prototyping or simulation. The reality is that an architect will often make a choice and "ride the horse until it drops"—that is, commit to a decision and revisit it only if it appears to be compromising the success of the project. These decisions have serious consequences!

Recall that, in the early stages of design, decisions focus on the biggest, most critical choices that will have substantial downstream consequences: reference architectures, major technologies (such as frameworks), and patterns. Reference architectures, deployment patterns, and other kinds of patterns have been widely discussed—there are many books, websites, and conferences devoted to the creation and validation of patterns and pattern languages. Nevertheless, the output of these activities is always a set of documented patterns. Interpreting the patterns from a pattern catalog is a critical part of the selection activity for an architect. Each candidate pattern must be chosen and its instantiation must be analyzed. For example, if you chose the Layers pattern from Figure 2.4, you would still have many decisions to make: how many layers there will be, how strict the layering will be, which specific services will be placed into each layer, what the interfaces between these functions will be, and so forth. If you chose the Load-Balanced Cluster deployment pattern from Figure 2.7, you would have to decide how many servers will be balanced, how many load balancers you will use, where these servers and load balancers will physically reside, which kinds of networks will connect these servers, which form of encryption you will use on those network connections, which form of health monitoring the load balancers will employ, and so forth. These decisions are important and will affect the success of the instantiated pattern, so they need to be analyzed. In addition, the quality of the *implementation* of these decisions will affect the success of the pattern. As we like to quip, the architecture giveth and the implementation taketh away.

Furthermore, the many catalogs and web pages that present design concepts use different conventions and notations. The focus of our book is on the design method and how it can be used with these external sources. For this reason we just take examples from outside sources and show them here as they were originally presented. This book is not intended to be another design patterns catalog—we want to alert you to the presence of these catalogs and show how they can be an incredibly useful resource for an architect, but they must be interpreted and used with care! In fact, one of your many jobs as an architect is to understand and interpret these catalogs, with their different notations and conventions. This is the reality that you will have to deal with.

Finally, once a design decision has been made, you should think about how you will *document* it. You could, of course, do no documentation. This is, in fact, what is most common in practice. Architectural concepts are often vague and conveyed informally, in "tribal knowledge": personal communications, emails, naming conventions, and so forth. Alternatively, you could create and maintain full, formal documentation, as is done for some projects with demanding quality attribute requirements, such as safety-critical or high-security systems. If you are designing flight-control software, you will probably end up at this end of the spectrum. In between these endpoints is a broad set of possibilities, and in this space we see less formal (and less costly) forms of architecture documentation, such as sketches (as we will discuss in Section 3.7).

The decision of what, when, and how to document should be risk based. You should ask yourself: What is the risk of *not* documenting this decision? Could it be misinterpreted and undermined by future developers? Could it contribute to near-term or long-term problems in the system? For example, if the rationale for layering is not carefully documented, the layering will inevitably break down, losing coherence and tending toward increased coupling. Over time, this trend will increase the system's technical debt, making it harder to find and fix bugs or add new features. To take another example, if the rationale for allocation of a critical resource is not documented, that resource might become an unintended contention area, resulting in bottlenecks and failures.

2.7 Summary

In this chapter, we introduced the idea of design as a set of decisions to satisfy requirements and constraints. We also introduced the notion of "architectural" design and showed that it does not differ from design in general, other than that it addresses the satisfaction of *architectural drivers*: the purpose, primary functionality, quality attribute requirements, architectural concerns, and constraints. What makes a decision "architectural"? A decision is architectural if it has nonlocal consequences *and* those consequences matter to the achievement of an architectural driver.

We also discussed why architectural design is so important: because it is the embodiment of early, far-reaching, hard-to-change decisions. These decisions will help you meet your architectural drivers, will determine much of your project's work-breakdown structure, and will affect the tools, skills, and technologies needed to realize the system. Thus architectural design decisions should be scrutinized well, as their consequences are profound. In addition, architecture is a key enabler of agility.

Architectural design is guided by certain principles. For example, to achieve good modularity, high coupling, and low cohesion, the wise architect will probably include some form of layering in the architecture being designed. Similarly, to achieve high availability, an architect will likely choose a pattern involving some form of redundancy and failover, such as active–passive redundancy, where an active server sends real-time updates to a passive server, so that the passive server can replace the active server in case it fails, with no loss of state.

Design concepts, such as reference architectures, deployment patterns, architectural patterns, tactics, and externally developed components, are the building blocks of design, and they form the foundation for architectural design as it is performed using ADD. As you will see in our step-by-step explanation of ADD in Chapter 3, some of the most important design decisions that an architect makes are how design concepts are selected, how they are instantiated, and how they are combined. Also, in Appendix A, we present a design concepts catalog that includes several instances of the design concepts presented here.

From these foundations, an architecture can be confidently and predictably constructed.

2.8 Further Reading

A more in-depth treatment of scenarios and architectural drivers can be found in L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice,* 3rd ed., Addison-Wesley, 2012. Also found in this book is an extensive discussion of architectural tactics, which are useful in guiding an architecture to achieve quality attribute goals. Likewise, this book contains an extensive discussion of QAW and Utility Trees.

The Mission Thread Workshop is discussed in R. Kazman, M. Gagliardi, and W. Wood, "Scaling Up Software Architecture Analysis", *Journal of Systems and Software*, 85, 1511–1519, 2012; and in M. Gagliardi, W. Wood, and T. Morrow, *Introduction to the Mission Thread Workshop*, Software Engineering Institute Technical Report CMU/SEI-2013-TR-003, 2013.

An overview of discovery prototyping, JRP, JAD, and accelerated systems analysis can be found in any competent book on systems analysis and design, such as J. Whitten and L. Bentley, *Systems Analysis and Design Methods,* 7th ed., McGraw-Hill, 2007. The combination of architectural approaches with Agile methods will be discussed in Chapter 9.

A catalog of reference architectures and deployment patterns appears in the book by the Microsoft Patterns and Practices Team: *Microsoft® Application Architecture Guide,* 2nd ed., Microsoft Press, 2009. This book also provides an extensive list of architectural concerns associated with the reference architectures that are documented.

An extensive collection of architectural design patterns for the construction of distributed systems can be found in F. Buschmann, K. Henney, and D. Schmidt, *Pattern-Oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing*, Wiley, 2007. Other books in the POSA (Patterns Of Software Architecture) series provide additional pattern catalogs. Many other pattern catalogs specializing in particular application domains and technologies exist. A few examples are listed here:

- E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2003.
- E. Fernandez-Buglioni. *Security Patterns in Practice: Designing Secure Architectures Using Software Patterns*. Wiley, 2013.
- G. Hohpe and B. Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, 2004.

The evaluation and selection of software packages is discussed in A. Jadhav and R. Sonar, "Evaluating and Selecting Software Packages: A Review", *Journal of Information and Software Technology,* 51, 555–563, 2009.

The "bible" for software architecture documentation is P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, P. Merson, R. Nord, and J. Stafford, *Documenting Software Architectures: Views and Beyond*, 2nd ed., Addison-Wesley, 2011.

The technology family tree for the Big Data application domain is based on the Smart Decisions Game by H. Cervantes, S. Haziyev, O. Hrytsay, and R. Kazman, which can be found at http://smartdecisionsgame.com.

Index

A

S