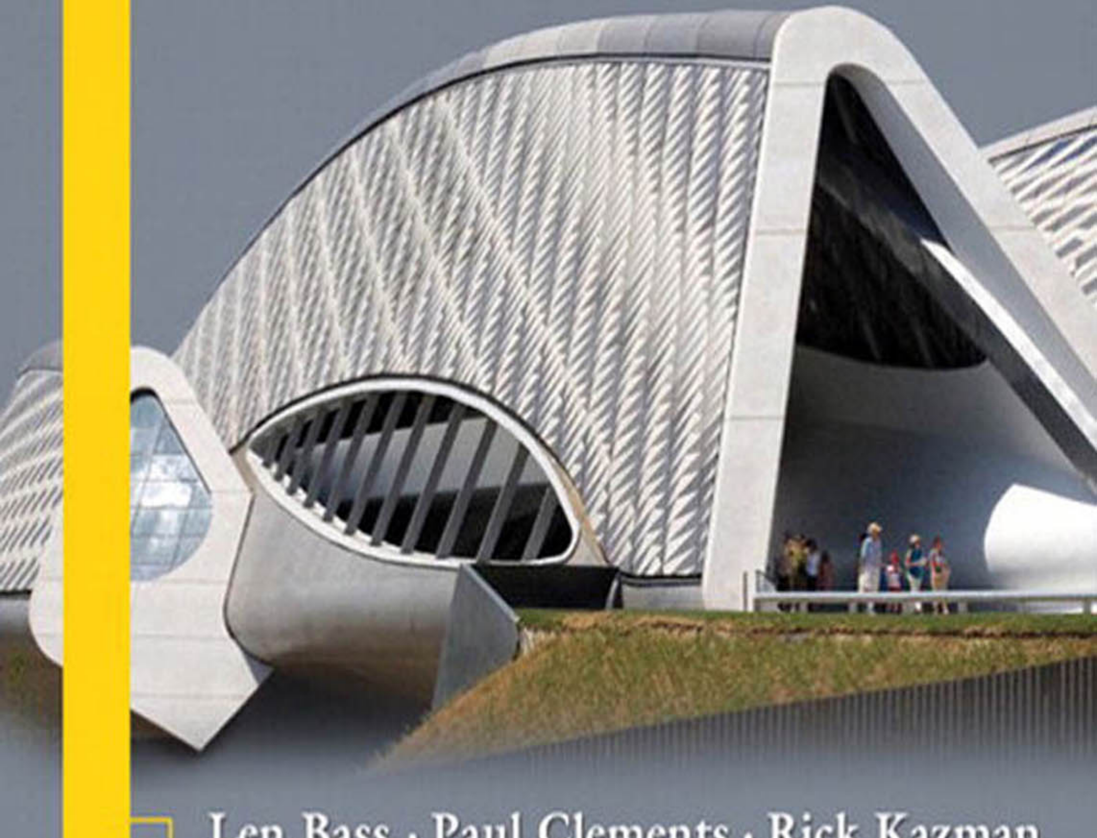
Software Architecture in Practice

Third Edition



Len Bass · Paul Clements · Rick Kazman

Software Architecture in Practice

Third Edition

The SEI Series in Software Engineering

Software Engineering Institute | Carnegie Mellon

CERT-RMM, VERSION 1.1
CERT® Resilience Management Model
A Maturity Model for Managing Operational Resilience
Richard A. Caralli
Julia H. Allen
David W. White
CMMI-DEV, VERSION 1.3
CMMI for Development
Guidelines for Process Integration and Product Improvement
Third Edition
Mary Beth Chrissis
Mike Konrad
Sandy Shrum
TSP
Leading a Development Team
Watts S. Humphrey
Documenting Software Architectures
Views and Beyond
SECOND EDITION
Reflections on Management
How to Manage Your Software Projects, Your Teams, Your Boss, and Yourself
Watts S. Humphrey
with William R. Thomas

Addison-Wesley

Visit **informit.com/sei** for a complete list of available products.

The **SEI Series in Software Engineering** represents is a collaborative undertaking of the Carnegie Mellon Software Engineering Institute (SEI) and Addison-Wesley to develop and publish books on software engineering and related topics. The common goal of the SEI and Addison-Wesley is to provide the most current information on these topics in a form that is easily usable by practitioners and students.

Books in the series describe frameworks, tools, methods, and technologies designed to help organizations, teams, and individuals improve their technical or management capabilities. Some books describe processes and practices for developing higher-quality software, acquiring programs for complex systems, or delivering services more effectively. Other books focus on software and system architecture and product-line development. Still others, from the SEI's CERT Program, describe technologies and practices needed to manage software and network security risk. These and all books in the series address critical problems in software engineering for which practical solutions are available.

Software Architecture in Practice

Third Edition

Len Bass
Paul Clements
Rick Kazman

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Software Engineering Institute | Carnegie Mellon

The SEI Series in Software Engineering

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

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

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact international@pearsoned.com.

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data

Bass, Len.
Software architecture in practice / Len Bass, Paul Clements, Rick Kazman.—3rd ed.
p. cm.—(SEI series in software engineering)
Includes bibliographical references and index.
ISBN 978-0-321-81573-6 (hardcover : alk. paper) 1. Software architecture. 2. System design. I. Clements, Paul, 1955– II. Kazman, Rick. III. Title.
QA76.754.B37 2012
005.1—dc23
2012023744

ISBN-13: 978-0-321-81573-6
ISBN-10: 0-321-81573-4

Text printed in the United States on recycled paper at Courier in Westford, Massachusetts.
Fifth printing, September 2015

Contents

PART FOUR ARCHITECTURE AND BUSINESS 435

This page intentionally left blank

Preface

I should have no objection to go over the same life from its beginning to the end: requesting only the advantage authors have, of correcting in a [third] edition the faults of the first [two].
— Benjamin Franklin

It has been a decade since the publication of the second edition of this book. During that time, the field of software architecture has broadened its focus from being primarily internally oriented—How does one design, evaluate, and document software?—to including external impacts as well—a deeper understanding of the influences on architectures and a deeper understanding of the impact architectures have on the life cycle, organizations, and management.

The past ten years have also seen dramatic changes in the types of systems being constructed. Large data, social media, and the cloud are all areas that, at most, were embryonic ten years ago and now are not only mature but extremely influential.

We listened to some of the criticisms of the previous editions and have included much more material on patterns, reorganized the material on quality attributes, and made interoperability a quality attribute worthy of its own chapter. We also provide guidance about how you can generate scenarios and tactics for your own favorite quality attributes.

To accommodate this plethora of new material, we had to make difficult choices. In particular, this edition of the book does not include extended case studies as the prior editions did. This decision also reflects the maturing of the field, in the sense that case studies about the choices made in software architectures are more prevalent than they were ten years ago, and they are less necessary to convince readers of the importance of software architecture. The case studies from the first two editions are available, however, on the book's website, at www.informit.com/title/9780321815736. In addition, on the same website, we have slides that will assist instructors in presenting this material.

We have thoroughly reworked many of the topics covered in this edition. In particular, we realize that the methods we present—for architecture design, analysis, and documentation—are one version of how to achieve a particular goal, but there are others. This led us to separate the methods that we present in detail from their underlying theory. We now present the theory first with specific methods given as illustrations of possible realizations of the theories. The new topics in this edition include architecture-centric project management; architecture competence; requirements modeling and analysis; Agile methods; implementation and testing; the cloud; and the edge.

As with the prior editions, we firmly believe that the topics are best discussed in either reading groups or in classroom settings, and to that end we have included a collection of discussion questions at the end of each chapter. Most of these questions are open-ended, with no absolute right or wrong answers, so you, as a reader, should emphasize how you justify your answer rather than just answer the question itself.

Reader's Guide

We have structured this book into five distinct portions. Part One introduces architecture and the various contextual lenses through which it could be viewed. These are the following:

- *Technical.* What technical role does the software architecture play in the system or systems of which it's a part?
- *Project.* How does a software architecture relate to the other phases of a software development life cycle?
- *Business.* How does the presence of a software architecture affect an organization's business environment?
- *Professional.* What is the role of a software architect in an organization or a development project?

Part Two is focused on technical background. Part Two describes how decisions are made. Decisions are based on the desired quality attributes for a system, and Chapters 5–11 describe seven different quality attributes and the techniques used to achieve them. The seven are availability, interoperability, maintainability, performance, security, testability, and usability. Chapter 12 tells you how to add other quality attributes to our seven, Chapter 13 discusses patterns and tactics, and Chapter 14 discusses the various types of modeling and analysis that are possible.

Part Three is devoted to how a software architecture is related to the other portions of the life cycle. Of special note is how architecture can be used in Agile projects. We discuss individually other aspects of the life cycle: requirements, design, implementation and testing, recovery and conformance, and evaluation.

Part Four deals with the business of architecting from an economic perspective, from an organizational perspective, and from the perspective of constructing a series of similar systems.

Part Five discusses several important emerging technologies and how architecture relates to these technologies.

This page intentionally left blank

Acknowledgments

We had a fantastic collection of reviewers for this edition, and their assistance helped make this a better book. Our reviewers were Muhammad Ali Babar, Felix Bachmann, Joe Batman, Phil Bianco, Jeromy Carriere, Roger Champagne, Steve Chenoweth, Viktor Clerc, Andres Diaz Pace, George Fairbanks, Rik Farenhorst, Ian Gorton, Greg Hartman, Rich Hilliard, James Ivers, John Klein, Philippe Kruchten, Phil Laplante, George Leih, Grace Lewis, John McGregor, Tommi Mikkonen, Linda Northrop, Ipek Ozkaya, Eltjo Poort, Eelco Rommes, Nick Rozanski, Jungwoo Ryoo, James Scott, Antony Tang, Arjen Uittenbogaard, Hans van Vliet, Hiroshi Wada, Rob Wojcik, Eoin Woods, and Liming Zhu.

In addition, we had significant contributions from Liming Zhu, Hong-Mei Chen, Jungwoo Ryoo, Phil Laplante, James Scott, Grace Lewis, and Nick Rozanski that helped give the book a richer flavor than one written by just the three of us.

The issue of build efficiency in Chapter 12 came from Rolf Siegers and John McDonald of Raytheon. John Klein and Eltjo Poort contributed the "abstract system clock" and "sandbox mode" tactics, respectively, for testability. The list of stakeholders in Chapter 3 is from *Documenting Software Architectures: Views and Beyond, Second Edition.* Some of the material in Chapter 28 was inspired by a talk given by Anthony Lattanze called "Organizational Design Thinking" in 2011.

Joe Batman was instrumental in the creation of the seven categories of design decisions we describe in Chapter 4. In addition, the descriptions of the security view, communications view, and exception view in Chapter 18 are based on material that Joe wrote while planning the documentation for a real system's architecture. Much of the new material on modifiability tactics was based on the work of Felix Bachmann and Rod Nord. James Ivers helped us with the security tactics.

Both Paul Clements and Len Bass have taken new positions since the last edition was published, and we thank their new respective managements (BigLever Software for Paul and NICTA for Len) for their willingness to support our work on this edition. We would also like to thank our (former) colleagues at the Software Engineering Institute for multiple contributions to the evolution of the ideas expressed in this edition.

Finally, as always, we thank our editor at Addison-Wesley, Peter Gordon, for providing guidance and support during the writing and production processes.

This page intentionally left blank

PART ONE

INTRODUCTION

What is a software architecture? What is it good for? How does it come to be? What effect does its existence have? These are the questions we answer in Part I.

Chapter 1 deals with a technical perspective on software architecture. We define it and relate it to system and enterprise architectures. We discuss how the architecture can be represented in different views to emphasize different perspectives on the architecture. We define patterns and discuss what makes a "good" architecture.

In Chapter 2, we discuss the uses of an architecture. You may be surprised that we can find so many—ranging from a vehicle for communication among stakeholders to a blueprint for implementation, to the carrier of the system's quality attributes. We also discuss how the architecture provides a reasoned basis for schedules and how it provides the foundation for training new members on a team.

Finally, in Chapter 3, we discuss the various contexts in which a software architecture exists. It exists in a technical context, in a project life-cycle context, in a business context, and in a professional context. Each of these contexts defines a role for the software architecture to play, or an influence on it. These impacts and influences define the Architecture Influence Cycle.

This page intentionally left blank

1

What Is Software Architecture?

Good judgment is usually the result of experience. And experience is frequently the result of bad judgment. But to learn from the experience of others requires those who have the experience to share the knowledge with those who follow.
—Barry LePatner

Writing (on our part) and reading (on your part) a book about software architecture, which distills the experience of many people, presupposes that

1. having a software architecture is important to the successful development of a software system and
2. there is a sufficient, and sufficiently generalizable, body of knowledge about software architecture to fill up a book.

One purpose of this book is to convince you that both of these assumptions are true, and once you are convinced, give you a basic knowledge so that you can apply it yourself.

Software systems are constructed to satisfy organizations' business goals. The architecture is a bridge between those (often abstract) business goals and the final (concrete) resulting system. While the path from abstract goals to concrete systems can be complex, the good news is that software architectures can be designed, analyzed, documented, and implemented using known techniques that will support the achievement of these business and mission goals. The complexity can be tamed, made tractable.

These, then, are the topics for this book: the design, analysis, documentation, and implementation of architectures. We will also examine the influences, principally in the form of business goals and quality attributes, which inform these activities.

In this chapter we will focus on architecture strictly from a software engineering point of view. That is, we will explore the value that a software architecture brings to a development project. (Later chapters will take a business and organizational perspective.)

1.1 What Software Architecture Is and What It Isn't

There are many definitions of software architecture, easily discoverable with a web search, but the one we like is this one:

> The software architecture of a system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both.

This definition stands in contrast to other definitions that talk about the system's "early" or "major" design decisions. While it is true that many architectural decisions are made early, not all are—especially in Agile or spiral-development projects. It's also true that very many decisions are made early that are not architectural. Also, it's hard to look at a decision and tell whether or not it's "major." Sometimes only time will tell. And since writing down an architecture is one of the architect's most important obligations, we need to know now which decisions an architecture comprises.

Structures, on the other hand, are fairly easy to identify in software, and they form a powerful tool for system design.

Let us look at some of the implications of our definition.

Architecture Is a Set of Software Structures

This is the first and most obvious implication of our definition. A structure is simply a set of elements held together by a relation. Software systems are composed of many structures, and no single structure holds claim to being *the* architecture. There are three categories of architectural structures, which will play an important role in the design, documentation, and analysis of architectures:

1. First, some structures partition systems into implementation units, which in this book we call *modules*. Modules are assigned specific computational responsibilities, and are the basis of work assignments for programming teams (Team A works on the database, Team B works on the business rules, Team C works on the user interface, etc.). In large projects, these elements (modules) are subdivided for assignment to subteams. For example, the database for a large enterprise resource planning (ERP) implementation might be so complex that its implementation is split into many parts. The structure that captures that decomposition is a kind of module structure, the *module*

decomposition structure in fact. Another kind of module structure emerges as an output of object-oriented analysis and design—class diagrams. If you aggregate your modules into layers, you've created another (and very useful) module structure. Module structures are static structures, in that they focus on the way the system's functionality is divided up and assigned to implementation teams.

2. Other structures are dynamic, meaning that they focus on the way the elements interact with each other at runtime to carry out the system's functions. Suppose the system is to be built as a set of services. The services, the infrastructure they interact with, and the synchronization and interaction relations among them form another kind of structure often used to describe a system. These services are made up of (compiled from) the programs in the various implementation units that we just described. In this book we will call runtime structures *component-and-connector* (C&C) structures. The term *component* is overloaded in software engineering. In our use, a component is always a runtime entity.
3. A third kind of structure describes the mapping from software structures to the system's organizational, developmental, installation, and execution environments. For example, modules are assigned to teams to develop, and assigned to places in a file structure for implementation, integration, and testing. Components are deployed onto hardware in order to execute. These mappings are called *allocation* structures.

Although software comprises an endless supply of structures, not all of them are architectural. For example, the set of lines of source code that contain the letter "z," ordered by increasing length from shortest to longest, is a software structure. But it's not a very interesting one, nor is it architectural. A structure is architectural if it supports reasoning about the system and the system's properties. The reasoning should be about an attribute of the system that is important to some stakeholder. These include functionality achieved by the system, the availability of the system in the face of faults, the difficulty of making specific changes to the system, the responsiveness of the system to user requests, and many others. We will spend a great deal of time in this book on the relationship between architecture and quality attributes like these.

Thus, the set of architectural structures is not fixed or limited. What is architectural is what is useful in your context for your system.

Architecture Is an Abstraction

Because architecture consists of structures and structures consist of elements[1] and relations, it follows that an architecture comprises software elements and

[1] In this book we use the term "element" when we mean either a module or a component, and don't want to distinguish.

how the elements relate to each other. This means that architecture specifically omits certain information about elements that is not useful for reasoning about the system—in particular, it omits information that has no ramifications outside of a single element. Thus, an architecture is foremost an *abstraction* of a system that selects certain details and suppresses others. In all modern systems, elements interact with each other by means of interfaces that partition details about an element into public and private parts. Architecture is concerned with the public side of this division; private details of elements—details having to do solely with internal implementation—are not architectural. Beyond just interfaces, though, the architectural abstraction lets us look at the system in terms of its elements, how they are arranged, how they interact, how they are composed, what their properties are that support our system reasoning, and so forth. This abstraction is essential to taming the complexity of a system—we simply cannot, and do not want to, deal with all of the complexity all of the time.

Every Software System Has a Software Architecture

Every system can be shown to comprise elements and relations among them to support some type of reasoning. In the most trivial case, a system is itself a single element—an uninteresting and probably non-useful architecture, but an architecture nevertheless.

Even though every system has an architecture, it does not necessarily follow that the architecture is known to anyone. Perhaps all of the people who designed the system are long gone, the documentation has vanished (or was never produced), the source code has been lost (or was never delivered), and all we have is the executing binary code. This reveals the difference between the architecture of a system and the *representation* of that architecture. Because an architecture can exist independently of its description or specification, this raises the importance of *architecture documentation*, which is described in Chapter 18, and *architecture reconstruction*, discussed in Chapter 20.

Architecture Includes Behavior

The behavior of each element is part of the architecture insofar as that behavior can be used to reason about the system. This behavior embodies how elements interact with each other, which is clearly part of our definition of architecture.

This tells us that box-and-line drawings that are passed off as architectures are in fact not architectures at all. When looking at the names of the boxes (database, graphical user interface, executive, etc.), a reader may well imagine the functionality and behavior of the corresponding elements. This mental image approaches an architecture, but it springs from the imagination of the observer's mind and relies on information that is not present. This does not mean that the exact behavior and performance of every element must be documented in all circumstances—some aspects of behavior are fine-grained and below the architect's level of concern. But to the extent that an element's behavior influences another element or influences the acceptability of the system as a whole, this behavior must be considered, and should be documented, as part of the software architecture.

Not All Architectures Are Good Architectures

The definition is indifferent as to whether the architecture for a system is a good one or a bad one. An architecture may permit or preclude a system's achievement of its behavioral, quality attribute, and life-cycle requirements. Assuming that we do not accept trial and error as the best way to choose an architecture for a system—that is, picking an architecture at random, building the system from it, and then hacking away and hoping for the best—this raises the importance of *architecture design*, which is treated in Chapter 17, and *architecture evaluation*, which we deal with in Chapter 21.

System and Enterprise Architectures

Two disciplines related to software architecture are system architecture and enterprise architecture. Both of these disciplines have broader concerns than software and affect software architecture through the establishment of constraints within which a software system must live. In both cases, the software architect for a system should be on the team that provides input into the decisions made about the system or the enterprise.

System architecture
A system's architecture is a representation of a system in which there is a mapping of functionality onto hardware and software components, a mapping of the software architecture onto the hardware architecture, and a concern for the human interaction with these components. That is, system architecture is concerned with a total system, including hardware, software, and humans.

A system architecture will determine, for example, the functionality that is assigned to different processors and the type of network that connects those processors. The software architecture on each of those processors will determine how this functionality is implemented and how the various processors interact through the exchange of messages on the network.

A description of the software architecture, as it is mapped to hardware and networking components, allows reasoning about qualities such as performance and reliability. A description of the system architecture will allow reasoning about additional qualities such as power consumption, weight, and physical footprint.

When a particular system is designed, there is frequently negotiation between the system architect and the software architect as to the distribution of functionality and, consequently, the constraints placed on the software architecture.

Enterprise architecture
Enterprise architecture is a description of the structure and behavior of an organization's processes, information flow, personnel, and organizational subunits, aligned with the organization's core goals and strategic direction. An enterprise architecture need not include information systems—clearly organizations had architectures that fit the preceding definition prior to the advent of computers—but these days, enterprise architectures for all but the smallest businesses are unthinkable without information system support. Thus, a modern enterprise architecture is concerned with how an enterprise's software systems support the business processes and goals of the enterprise. Typically included in this set of concerns is a process for deciding which systems with which functionality should be supported by an enterprise.

An enterprise architecture will specify the data model that various systems use to interact, for example. It will specify rules for how the enterprise's systems interact with external systems.

Software is only one concern of enterprise architecture. Two other common concerns addressed by enterprise architecture are how the software is used by humans to perform business processes, and the standards that determine the computational environment.

Sometimes the software infrastructure that supports communication among systems and with the external world is considered a portion of the enterprise architecture; other times, this infrastructure is considered one of the systems within an enterprise. (In either case, the architecture of that infrastructure is a *software* architecture!) These two views will result in different management structures and spheres of influence for the individuals concerned with the infrastructure.

The system and the enterprise provide environments for, and constraints on, the software architecture. The software architecture must live within the system and enterprise, and increasingly it is the focus for achieving the organization's business goals. But all three forms of architecture share important commonalities: They are concerned with major elements taken as abstractions, the relationships among the elements, and how the elements together meet the behavioral and quality goals of the thing being built.

Are these in scope for this book? Yes! (Well, no.)
System and enterprise architectures share a great deal with software architectures. All can be designed, evaluated, and documented; all answer to requirements; all are intended to satisfy stakeholders; all consist of structures, which in turn consist of elements and relationships; all have a repertoire of patterns and styles at their respective architects' disposal; and the list goes on. So to the extent that these architectures share commonalities with software architecture, they are in the scope of this book. But like all technical disciplines, each has its own specialized vocabulary and techniques, and we won't cover those. Copious other sources do.

1.2 Architectural Structures and Views

The neurologist, the orthopedist, the hematologist, and the dermatologist all have different views of the structure of a human body. Ophthalmologists, cardiologists, and podiatrists concentrate on specific subsystems. And the kinesiologist and psychiatrist are concerned with different aspects of the entire arrangement's behavior. Although these views are pictured differently and have very different properties, all are inherently related, interconnected: together they describe the architecture of the human body. Figure 1.1 shows several different views of the human body: the skeletal, the vascular, and the X-ray.

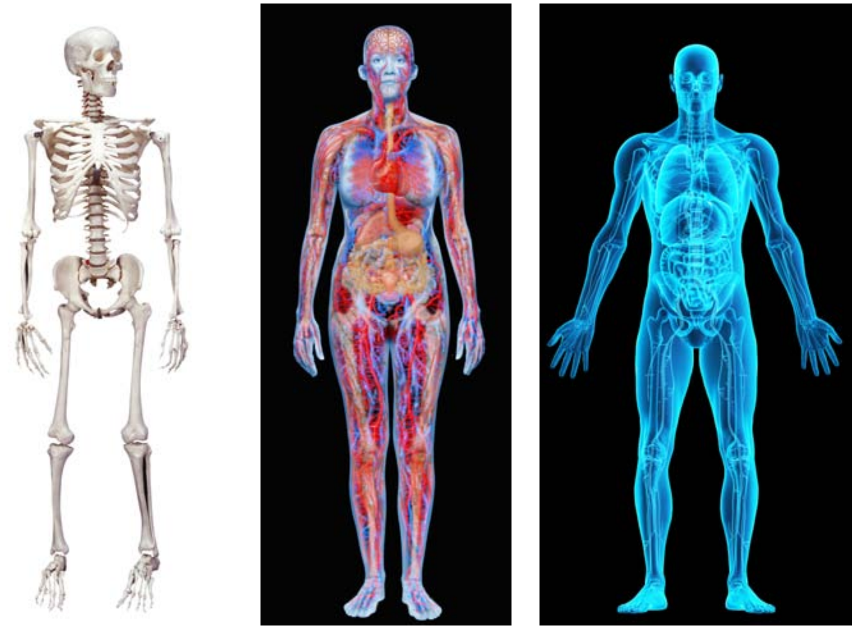


FIGURE 1.1 Physiological structures (Getty images: Brand X Pictures [skeleton], Don Farrall [woman], Mads Abildgaard [man])

So it is with software. Modern systems are frequently too complex to grasp all at once. Instead, we restrict our attention at any one moment to one (or a small number) of the software system's structures. To communicate meaningfully about an architecture, we must make clear which structure or structures we are discussing at the moment—which *view* we are taking of the architecture.

Structures and Views

We will be using the related terms *structure* and *view* when discussing architecture representation.

- A view is a representation of a coherent set of architectural elements, as written by and read by system stakeholders. It consists of a representation of a set of elements and the relations among them.
- A structure is the set of elements itself, as they exist in software or hardware.

In short, a view is a representation of a structure. For example, a module *structure* is the set of the system's modules and their organization. A module *view* is the representation of that structure, documented according to a template in a chosen notation, and used by some system stakeholders.

So: Architects design structures. They document views of those structures.

Three Kinds of Structures

As we saw in the previous section, architectural structures can be divided into three major categories, depending on the broad nature of the elements they show. These correspond to the three broad kinds of decisions that architectural design involves:

1. *Module structures* embody decisions as to how the system is to be structured as a set of code or data units that have to be constructed or procured. In any module structure, the elements are modules of some kind (perhaps classes, or layers, or merely divisions of functionality, all of which are units of implementation). Modules represent a static way of considering the system. Modules are assigned areas of functional responsibility; there is less emphasis in these structures on how the resulting software manifests itself at runtime. Module structures allow us to answer questions such as these:

    - What is the primary functional responsibility assigned to each module?
    - What other software elements is a module allowed to use?
    - What other software does it actually use and depend on?
    - What modules are related to other modules by generalization or specialization (i.e., inheritance) relationships?

    Module structures convey this information directly, but they can also be used by extension to ask questions about the impact on the system when the responsibilities assigned to each module change. In other words, examining a system's module structures—that is, looking at its module views—is an excellent way to reason about a system's modifiability.
2. *Component-and-connector structures* embody decisions as to how the system is to be structured as a set of elements that have runtime behavior (components) and interactions (connectors). In these structures, the

elements are runtime components (which are the principal units of computation and could be services, peers, clients, servers, filters, or many other types of runtime elements) and connectors (which are the communication vehicles among components, such as call-return, process synchronization operators, pipes, or others). Component-and-connector views help us answer questions such as these:

- What are the major executing components and how do they interact at runtime?
- What are the major shared data stores?
- Which parts of the system are replicated?
- How does data progress through the system?
- What parts of the system can run in parallel?
- Can the system's structure change as it executes and, if so, how?

By extension, component-and-connector views are crucially important for asking questions about the system's runtime properties such as performance, security, availability, and more.

3. *Allocation structures* embody decisions as to how the system will relate to nonsoftware structures in its environment (such as CPUs, file systems, networks, development teams, etc.). These structures show the relationship between the software elements and elements in one or more external environments in which the software is created and executed. Allocation views help us answer questions such as these:

- What processor does each software element execute on?
- In what directories or files is each element stored during development, testing, and system building?
- What is the assignment of each software element to development teams?

Structures Provide Insight

Structures play such an important role in our perspective on software architecture because of the analytical and engineering power they hold. Each structure provides a perspective for reasoning about some of the relevant quality attributes. For example:

- The module "uses" structure, which embodies what modules use what other modules, is strongly tied to the ease with which a system can be extended or contracted.
- The concurrency structure, which embodies parallelism within the system, is strongly tied to the ease with which a system can be made free of deadlock and performance bottlenecks.
- The deployment structure is strongly tied to the achievement of performance, availability, and security goals.

And so forth. Each structure provides the architect with a different insight into the design (that is, each structure can be analyzed for its ability to deliver a quality attribute). But perhaps more important, each structure presents the architect with an engineering leverage point: By designing the structures appropriately, the desired quality attributes emerge.

Scenarios, described in Chapter 4, are useful for exercising a given structure as well as its connections to other structures. For example, a software engineer wanting to make a change to the concurrency structure of a system would need to consult the concurrency and deployment views, because the affected mechanisms typically involve processes and threads, and physical distribution might involve different control mechanisms than would be used if the processes were co-located on a single machine. If control mechanisms need to be changed, the module decomposition would need to be consulted to determine the extent of the changes.

Some Useful Module Structures

Useful module structures include the following:

- *Decomposition structure.* The units are modules that are related to each other by the *is-a-submodule-of* relation, showing how modules are decomposed into smaller modules recursively until the modules are small enough to be easily understood. Modules in this structure represent a common starting point for design, as the architect enumerates what the units of software will have to do and assigns each item to a module for subsequent (more detailed) design and eventual implementation. Modules often have products (such as interface specifications, code, test plans, etc.) associated with them. The decomposition structure determines, to a large degree, the system's modifiability, by assuring that likely changes are localized. That is, changes fall within the purview of at most a few (preferably small) modules. This structure is often used as the basis for the development project's organization, including the structure of the documentation, and the project's integration and test plans. The units in this structure tend to have names that are organization-specific such as "segment" or "subsystem."
- *Uses structure.* In this important but overlooked structure, the units here are also modules, perhaps classes. The units are related by the *uses* relation, a specialized form of dependency. A unit of software uses another if the correctness of the first requires the presence of a correctly functioning version (as opposed to a stub) of the second. The uses structure is used to engineer systems that can be extended to add functionality, or from which useful functional subsets can be extracted. The ability to easily create a subset of a system allows for incremental development.

- *Layer structure.* The modules in this structure are called layers. A layer is an abstract "virtual machine" that provides a cohesive set of services through a managed interface. Layers are allowed to use other layers in a strictly managed fashion; in strictly layered systems, a layer is only allowed to use the layer immediately below. This structure is used to imbue a system with portability, the ability to change the underlying computing platform.
- *Class (or generalization) structure*. The module units in this structure are called classes. The relation is *inherits from* or *is an instance of*. This view supports reasoning about collections of similar behavior or capability (e.g., the classes that other classes inherit from) and parameterized differences. The class structure allows one to reason about reuse and the incremental addition of functionality. If any documentation exists for a project that has followed an object-oriented analysis and design process, it is typically this structure.
- *Data model.* The data model describes the static information structure in terms of data entities and their relationships. For example, in a banking system, entities will typically include Account, Customer, and Loan. Account has several attributes, such as account number, type (savings or checking), status, and current balance. A relationship may dictate that one customer can have one or more accounts, and one account is associated to one or two customers.

Some Useful C&C Structures

Component-and-connector structures show a runtime view of the system. In these structures the modules described above have all been compiled into executable forms. All component-and-connector structures are thus orthogonal to the module-based structures and deal with the dynamic aspects of a running system. The relation in all component-and-connector structures is *attachment*, showing how the components and the connectors are hooked together. (The connectors themselves can be familiar constructs such as "invokes.") Useful C&C structures include the following:

- *Service structure*. The units here are services that interoperate with each other by service coordination mechanisms such as SOAP (see Chapter 6). The service structure is an important structure to help engineer a system composed of components that may have been developed anonymously and independently of each other.
- *Concurrency structure*. This component-and-connector structure allows the architect to determine opportunities for parallelism and the locations where resource contention may occur. The units are components and the connectors are their communication mechanisms. The components are arranged into *logical threads*; a logical thread is a sequence of computations that

could be allocated to a separate physical thread later in the design process. The concurrency structure is used early in the design process to identify the requirements to manage the issues associated with concurrent execution.

Some Useful Allocation Structures

Allocation structures define how the elements from C&C or module structures map onto things that are not software: typically hardware, teams, and file systems. Useful allocation structures include these:

- *Deployment structure*. The deployment structure shows how software is assigned to hardware processing and communication elements. The elements are software elements (usually a process from a C&C view), hardware entities (processors), and communication pathways. Relations are *allocated-to*, showing on which physical units the software elements reside, and *migrates-to* if the allocation is dynamic. This structure can be used to reason about performance, data integrity, security, and availability. It is of particular interest in distributed and parallel systems.
- *Implementation structure*. This structure shows how software elements (usually modules) are mapped to the file structure(s) in the system's development, integration, or configuration control environments. This is critical for the management of development activities and build processes. (In practice, a screenshot of your development environment tool, which manages the implementation environment, often makes a very useful and sufficient diagram of your implementation view.)
- *Work assignment structure.* This structure assigns responsibility for implementing and integrating the modules to the teams who will carry it out. Having a work assignment structure be part of the architecture makes it clear that the decision about who does the work has architectural as well as management implications. The architect will know the expertise required on each team. Also, on large multi-sourced distributed development projects, the work assignment structure is the means for calling out units of functional commonality and assigning those to a single team, rather than having them implemented by everyone who needs them. This structure will also determine the major communication pathways among the teams: regular teleconferences, wikis, email lists, and so forth.

Table 1.1 summarizes these structures. The table lists the meaning of the elements and relations in each structure and tells what each might be used for.

Relating Structures to Each Other

Each of these structures provides a different perspective and design handle on a system, and each is valid and useful in its own right. Although the structures give

TABLE 1.1 Useful Architectural Structures

	Software Structure	Element Types	Relations	Useful For	Quality Attributes Affected
Module Structures	Decomposition	Module	Is a submodule of	Resource allocation and project structuring and planning; information hiding, encapsulation; configuration control	Modifiability
	Uses	Module	Uses (i.e., requires the correct presence of)	Engineering subsets, engineering extensions	"Subsetability," extensibility
	Layers	Layer	Requires the correct presence of, uses the services of, provides abstraction to	Incremental development, implementing systems on top of "virtual machines"	Portability
	Class	Class, object	Is an instance of, shares access methods of	In object-oriented design systems, factoring out commonality; planning extensions of functionality	Modifiability, extensibility
	Data model	Data entity	{one, many}-to-{one, many}, generalizes, specializes	Engineering global data structures for consistency and performance	Modifiability, performance
C&C Structures	Service	Service, ESB, registry, others	Runs concurrently with, may run concurrently with, excludes, precedes, etc.	Scheduling analysis, performance analysis	Interoperability, modifiability
	Concurrency	Processes, threads	Can run in parallel	Identifying locations where resource contention exists, or where threads may fork, join, be created, or be killed	Performance, availability
Allocation Structures	Deployment	Components, hardware elements	Allocated to, migrates to	Performance, availability, security analysis	Performance, security, availability
	Implementation	Modules, file structure	Stored in	Configuration control, integration, test activities	Development efficiency
	Work assignment	Modules, organizational units	Assigned to	Project management, best use of expertise and available resources, management of commonality	Development efficiency

different system perspectives, they are not independent. Elements of one structure will be related to elements of other structures, and we need to reason about these relations. For example, a module in a decomposition structure may be manifested as one, part of one, or several components in one of the component-and-connector structures, reflecting its runtime alter ego. In general, mappings between structures are many to many.

Figure 1.2 shows a very simple example of how two structures might relate to each other. The figure on the left shows a module decomposition view of a tiny client-server system. In this system, two modules must be implemented: The client software and the server software. The figure on the right shows a component-and-connector view of the same system. At runtime there are ten clients running and accessing the server. Thus, this little system has two modules and eleven components (and ten connectors).

Whereas the correspondence between the elements in the decomposition structure and the client-server structure is obvious, these two views are used for very different things. For example, the view on the right could be used for performance analysis, bottleneck prediction, and network traffic management, which would be extremely difficult or impossible to do with the view on the left.

(In Chapter 13 we'll learn about the map-reduce pattern, in which copies of simple, identical functionality are distributed across hundreds or thousands of processing nodes—one module for the whole system, but one component per node.)

Individual projects sometimes consider one structure dominant and cast other structures, when possible, in terms of the dominant structure. Often the dominant structure is the module decomposition structure. This is for a good

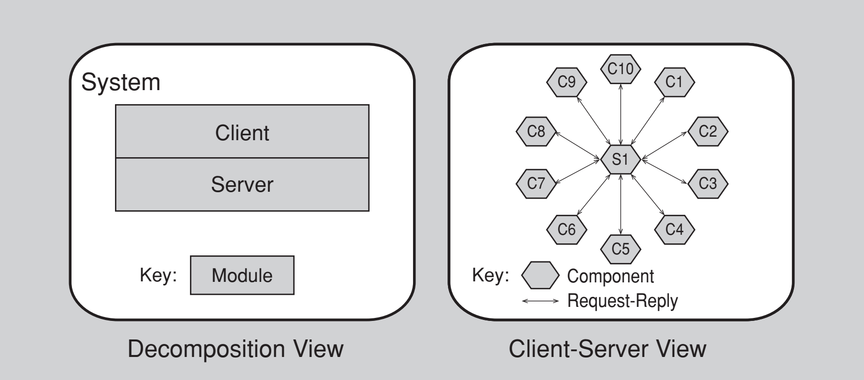


FIGURE 1.2 Two views of a client-server system

reason: it tends to spawn the project structure, because it mirrors the team structure of development. In other projects, the dominant structure might be a C&C structure that shows how the system's functionality and/or critical quality attributes are achieved.

Fewer Is Better

Not all systems warrant consideration of many architectural structures. The larger the system, the more dramatic the difference between these structures tends to be; but for small systems we can often get by with fewer. Instead of working with each of several component-and-connector structures, usually a single one will do. If there is only one process, then the process structure collapses to a single node and need not be explicitly represented in the design. If there is to be no distribution (that is, if the system is implemented on a single processor), then the deployment structure is trivial and need not be considered further. In general, design and document a structure only if doing so brings a positive return on the investment, usually in terms of decreased development or maintenance costs.

Which Structures to Choose?

We have briefly described a number of useful architectural structures, and there are many more. Which ones shall an architect choose to work on? Which ones shall the architect choose to document? Surely not all of them. Chapter 18 will treat this topic in more depth, but for now a good answer is that you should think about how the various structures available to you provide insight and leverage into the system's most important quality attributes, and then choose the ones that will play the best role in delivering those attributes.

Ask Cal

More than a decade ago I went to a customer site to do an architecture evaluation—one of the first instances of the Architecture Tradeoff Analysis Method (ATAM) that I had ever performed (you can read about the ATAM, and other architecture evaluation topics, in Chapter 21). In those early days, we were still figuring out how to make architecture evaluations repeatable and predictable, and how to guarantee useful outcomes from them. One of the ways that we ensured useful outcomes was to enforce certain preconditions on the evaluation. A precondition that we figured out rather quickly was this: if the architecture has not been documented, we will not proceed with the evaluation. The reason for this precondition was simple: we could not evaluate the architecture by reading the code—we didn't have the time for that—and we couldn't just ask the architect to sketch the architecture in real time, since that would produce vague and very likely erroneous representations.

Okay, it's not completely true to say that they had *no* architecture documentation. They did produce a single-page diagram, with a few boxes and lines. Some of those boxes were, however, clouds. Yes, they actually used a cloud as one of their icons. When I pressed them on the meaning of this icon—Was it a process? A class? A thread?—they waffled. This was not, in fact, architecture documentation. It was, at best, "marketecture."

But in those early days we had no preconditions and so we didn't stop the evaluation there. We just blithely waded in to whatever swamp we found, and we enforced nothing. As I began this evaluation, I interviewed some of the key project stakeholders: the project manager and several of the architects (this was a large project with one lead architect and several subordinates). As it happens, the lead architect was away, and so I spent my time with the subordinate architects. Every time I asked the subordinates a tough question—"How do you ensure that you will meet your latency goal along this critical execution path?" or "What are your rules for layering?"—they would answer: "Ask Cal. Cal knows that." Cal was the lead architect. Immediately I noted a risk for this system: What if Cal gets hit by a bus? What then?

In the end, because of my pestering, the architecture team did in fact produce respectable architecture documentation. About halfway through the evaluation, the project manager came up to me and shook my hand and thanked me for the great job I had done. I was dumbstruck. In my mind I hadn't done anything, at that point; the evaluation was only partially complete and I hadn't produced a single report or finding. I said that to the manager and he said: "You got those guys to document the architecture. I've never been able to get them to do that. So . . . thanks!"

If Cal had been hit by a bus or just left the company, they would have had a serious problem on their hands: all of that architectural knowledge located in one guy's head and he is no longer with the organization. In can happen. It *does* happen.

The moral of this story? An architecture that is not documented, and not communicated, may still be a good architecture, but the risks surrounding it are enormous.

—RK

1.3 Architectural Patterns

In some cases, architectural elements are composed in ways that solve particular problems. The compositions have been found useful over time, and over many different domains, and so they have been documented and disseminated. These compositions of architectural elements, called *architectural patterns*, provide packaged strategies for solving some of the problems facing a system.

An architectural pattern delineates the element types and their forms of interaction used in solving the problem. Patterns can be characterized according to the type of architectural elements they use. For example, a common module type pattern is this:

- *Layered pattern*. When the *uses* relation among software elements is strictly unidirectional, a system of layers emerges. A layer is a coherent set of related functionality. In a *strictly* layered structure, a layer can only use the services of the layer immediately below it. Many variations of this pattern, lessening the structural restriction, occur in practice. Layers are often designed as abstractions (virtual machines) that hide implementation specifics below from the layers above, engendering portability.

Common component-and-connector type patterns are these:

- *Shared-data (or repository) pattern.* This pattern comprises components and connectors that create, store, and access persistent data. The repository usually takes the form of a (commercial) database. The connectors are protocols for managing the data, such as SQL.
- *Client-server pattern.* The components are the clients and the servers, and the connectors are protocols and messages they share among each other to carry out the system’s work.

Common allocation patterns include the following:

- *Multi-tier pattern*, which describes how to distribute and allocate the components of a system in distinct subsets of hardware and software, connected by some communication medium. This pattern specializes the generic deployment (software-to-hardware allocation) structure.
- *Competence center* and *platform*, which are patterns that specialize a software system’s work assignment structure. In *competence center*, work is allocated to sites depending on the technical or domain expertise located at a site. For example, user-interface design is done at a site where usability engineering experts are located. In *platform*, one site is tasked with developing reusable core assets of a software product line (see Chapter 25), and other sites develop applications that use the core assets.

Architectural patterns will be investigated much further in Chapter 13.

1.4 What Makes a “Good” Architecture?

There is no such thing as an inherently good or bad architecture. Architectures are either more or less fit for some purpose. A three-tier layered service-oriented architecture may be just the ticket for a large enterprise’s web-based B2B system but completely wrong for an avionics application. An architecture carefully crafted to achieve high modifiability does not make sense for a throwaway prototype (and vice versa!). One of the messages of this book is that architectures can in fact be *evaluated*—one of the great benefits of paying attention to them—but only in the context of specific stated goals.

Nevertheless, there are rules of thumb that should be followed when designing most architectures. Failure to apply any of these does not automatically mean that the architecture will be fatally flawed, but it should at least serve as a warning sign that should be investigated.

We divide our observations into two clusters: process recommendations and product (or structural) recommendations. Our process recommendations are the following:

1. The architecture should be the product of a single architect or a small group of architects with an identified technical leader. This approach gives the architecture its conceptual integrity and technical consistency. This recommendation holds for Agile and open source projects as well as "traditional" ones. There should be a strong connection between the architect(s) and the development team, to avoid ivory tower designs that are impractical.
2. The architect (or architecture team) should, on an ongoing basis, base the architecture on a prioritized list of well-specified quality attribute requirements. These will inform the tradeoffs that always occur. Functionality matters less.
3. The architecture should be documented using views. The views should address the concerns of the most important stakeholders in support of the project timeline. This might mean minimal documentation at first, elaborated later. Concerns usually are related to construction, analysis, and maintenance of the system, as well as education of new stakeholders about the system.
4. The architecture should be evaluated for its ability to deliver the system's important quality attributes. This should occur early in the life cycle, when it returns the most benefit, and repeated as appropriate, to ensure that changes to the architecture (or the environment for which it is intended) have not rendered the design obsolete.
5. The architecture should lend itself to incremental implementation, to avoid having to integrate everything at once (which almost never works) as well as to discover problems early. One way to do this is to create a "skeletal" system in which the communication paths are exercised but which at first has minimal functionality. This skeletal system can be used to "grow" the system incrementally, refactoring as necessary.

Our structural rules of thumb are as follows:

1. The architecture should feature well-defined modules whose functional responsibilities are assigned on the principles of information hiding and

separation of concerns. The information-hiding modules should encapsulate things likely to change, thus insulating the software from the effects of those changes. Each module should have a well-defined interface that encapsulates or “hides” the changeable aspects from other software that uses its facilities. These interfaces should allow their respective development teams to work largely independently of each other.

2. Unless your requirements are unprecedented—possible, but unlikely—your quality attributes should be achieved using well-known architectural patterns and tactics (described in Chapter 13) specific to each attribute.
3. The architecture should never depend on a particular version of a commercial product or tool. If it must, it should be structured so that changing to a different version is straightforward and inexpensive.
4. Modules that produce data should be separate from modules that consume data. This tends to increase modifiability because changes are frequently confined to either the production or the consumption side of data. If new data is added, both sides will have to change, but the separation allows for a staged (incremental) upgrade.
5. Don’t expect a one-to-one correspondence between modules and components. For example, in systems with concurrency, there may be multiple instances of a component running in parallel, where each component is built from the same module. For systems with multiple threads of concurrency, each thread may use services from several components, each of which was built from a different module.
6. Every process should be written so that its assignment to a specific processor can be easily changed, perhaps even at runtime.
7. The architecture should feature a small number of ways for components to interact. That is, the system should do the same things in the same way throughout. This will aid in understandability, reduce development time, increase reliability, and enhance modifiability.
8. The architecture should contain a specific (and small) set of resource contention areas, the resolution of which is clearly specified and maintained. For example, if network utilization is an area of concern, the architect should produce (and enforce) for each development team guidelines that will result in a minimum of network traffic. If performance is a concern, the architect should produce (and enforce) time budgets for the major threads.

1.5 Summary

The software architecture of a system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both.

A structure is a set of elements and the relations among them.

A view is a representation of a coherent set of architectural elements, as written by and read by system stakeholders. A view is a representation of one or more structures.

There are three categories of structures:

- Module structures show how a system is to be structured as a set of code or data units that have to be constructed or procured.
- Component-and-connector structures show how the system is to be structured as a set of elements that have runtime behavior (components) and interactions (connectors).
- Allocation structures show how the system will relate to nonsoftware structures in its environment (such as CPUs, file systems, networks, development teams, etc.).

Structures represent the primary engineering leverage points of an architecture. Each structure brings with it the power to manipulate one or more quality attributes. They represent a powerful approach for creating the architecture (and later, for analyzing it and explaining it to its stakeholders). And as we will see in Chapter 18, the structures that the architect has chosen as engineering leverage points are also the primary candidates to choose as the basis for architecture documentation.

Every system has a software architecture, but this architecture may be documented and disseminated, or it may not be.

There is no such thing as an inherently good or bad architecture. Architectures are either more or less fit for some purpose.

1.6 For Further Reading

The early work of David Parnas laid much of the conceptual foundation for what became the study of software architecture. A quintessential Parnas reader would include his foundational article on information hiding [Parnas 72] as well as his works on program families [Parnas 76], the structures inherent in software systems [Parnas 74], and introduction of the uses structure to build subsets and supersets of systems [Parnas 79]. All of these papers can be found in the more easily accessible collection of his important papers [Hoffman 00].

An early paper by Perry and Wolf [Perry 92] drew an analogy between software architecture views and structures and the structures one finds in a house (plumbing, electrical, and so forth).

Software architectural patterns have been extensively catalogued in the series *Pattern-Oriented Software Architecture* [Buschmann 96] and others. Chapter 13 of this book also deals with architectural patterns.

Early papers on architectural views as used in industrial development projects are [Soni 95] and [Kruchten 95]. The former grew into a book [Hofmeister 00] that presents a comprehensive picture of using views in development and analysis. The latter grew into the Rational Unified Process, about which there is no shortage of references, both paper and online. A good one is [Kruchten 03].

Cristina Gacek and her colleagues discuss the process issues surrounding software architecture in [Gacek 95].

Garlan and Shaw's seminal work on software architecture [Garlan 93] provides many excellent examples of architectural styles (a concept similar to patterns).

In [Clements 10a] you can find an extended discussion on the difference between an architectural pattern and an architectural style. (It argues that a pattern is a context-problem-solution triple; a style is simply a condensation that focuses most heavily on the solution part.)

See [Taylor 09] for a definition of software architecture based on decisions rather than on structure.

1.7 Discussion Questions

1. Software architecture is often compared to the architecture of buildings as a conceptual analogy. What are the strong points of that analogy? What is the correspondence in buildings to software architecture structures and views? To patterns? What are the weaknesses of the analogy? When does it break down?

2. Do the architectures you've been exposed to document different structures and relations like those described in this chapter? If so, which ones? If not, why not?

3. Is there a different definition of software architecture that you are familiar with? If so, compare and contrast it with the definition given in this chapter. Many definitions include considerations like "rationale" (stating the reasons why the architecture is what it is) or how the architecture will evolve over time. Do you agree or disagree that these considerations should be part of the definition of software architecture?

4. Discuss how an architecture serves as a basis for analysis. What about decision-making? What kinds of decision-making does an architecture empower?

5. What is architecture's role in project risk reduction?

6. Find a commonly accepted definition of *system architecture* and discuss what it has in common with software architecture. Do the same for *enterprise architecture*.

7. Find a published example of an architecture. What structure or structures are shown? Given its purpose, what structure or structures *should* have been shown? What analysis does the architecture support? Critique it: What questions do you have that the representation does not answer?

8. Sailing ships have architectures, which means they have "structures" that lend themselves to reasoning about the ship's performance and other quality attributes. Look up the technical definitions for *barque*, *brig*, *cutter*, *frigate*, *ketch*, *schooner*, and *sloop*. Propose a useful set of "structures" for distinguishing and reasoning about ship architectures.

2

Why Is Software Architecture Important?

Software architecture is the set of design
decisions which, if made incorrectly, may
cause your project to be cancelled.
—Eoin Woods

If architecture is the answer, what was the question?

While Chapter 3 will cover the business importance of architecture to an enterprise, this chapter focuses on why architecture matters from a technical perspective. We will examine a baker's dozen of the most important reasons.

1. An architecture will inhibit or enable a system's driving quality attributes.
2. The decisions made in an architecture allow you to reason about and manage change as the system evolves.
3. The analysis of an architecture enables early prediction of a system's qualities.
4. A documented architecture enhances communication among stakeholders.
5. The architecture is a carrier of the earliest and hence most fundamental, hardest-to-change design decisions.
6. An architecture defines a set of constraints on subsequent implementation.
7. The architecture dictates the structure of an organization, or vice versa.
8. An architecture can provide the basis for evolutionary prototyping.
9. An architecture is the key artifact that allows the architect and project manager to reason about cost and schedule.
10. An architecture can be created as a transferable, reusable model that forms the heart of a product line.
11. Architecture-based development focuses attention on the assembly of components, rather than simply on their creation.
12. By restricting design alternatives, architecture channels the creativity of developers, reducing design and system complexity.
13. An architecture can be the foundation for training a new team member.

Even if you already believe us that architecture is important and don't need the point hammered thirteen more times, think of these thirteen points (which form the outline for this chapter) as thirteen useful ways to use architecture in a project.

2.1 Inhibiting or Enabling a System's Quality Attributes

Whether a system will be able to exhibit its desired (or required) quality attributes is substantially determined by its architecture.

This is such an important message that we've devoted all of Part 2 of this book to expounding that message in detail. Until then, keep these examples in mind as a starting point:

- If your system requires high performance, then you need to pay attention to managing the time-based behavior of elements, their use of shared resources, and the frequency and volume of inter-element communication.
- If modifiability is important, then you need to pay careful attention to assigning responsibilities to elements so that the majority of changes to the system will affect a small number of those elements. (Ideally each change will affect just a single element.)
- If your system must be highly secure, then you need to manage and protect inter-element communication and control which elements are allowed to access which information; you may also need to introduce specialized elements (such as an authorization mechanism) into the architecture.
- If you believe that scalability will be important to the success of your system, then you need to carefully localize the use of resources to facilitate introduction of higher-capacity replacements, and you must avoid hard-coding in resource assumptions or limits.
- If your projects need the ability to deliver incremental subsets of the system, then you must carefully manage intercomponent usage.
- If you want the elements from your system to be reusable in other systems, then you need to restrict inter-element coupling, so that when you extract an element, it does not come out with too many attachments to its current environment to be useful.

The strategies for these and other quality attributes are supremely architectural. But an architecture alone cannot guarantee the functionality or quality required of a system. Poor downstream design or implementation decisions can always undermine an adequate architectural design. As we like to say (*mostly* in jest): The architecture giveth and the implementation taketh away. Decisions at all stages of the life cycle—from architectural design to coding and implementation—affect system quality. Therefore, quality is not completely a function of an architectural design.

A good architecture is necessary, but not sufficient, to ensure quality. Achieving quality attributes must be considered throughout design, implementation, and deployment. No quality attribute is entirely dependent on design, nor is it entirely dependent on implementation or deployment. Satisfactory results are a matter of getting the big picture (architecture) as well as the details (implementation) correct.

For example, modifiability is determined by how functionality is divided and coupled (architectural) and by coding techniques within a module (nonarchitectural). Thus, a system is typically modifiable if changes involve the fewest possible number of distinct elements. In spite of having the ideal architecture, however, it is always possible to make a system difficult to modify by writing obscure, tangled code.

2.2 Reasoning About and Managing Change

This point is a corollary to the previous point.

Modifiability—the ease with which changes can be made to a system—is a quality attribute (and hence covered by the arguments in the previous section), but it is such an important quality that we have awarded it its own spot in the List of Thirteen. The software development community is coming to grips with the fact that roughly 80 percent of a typical software system's total cost occurs *after* initial deployment. A corollary of this statistic is that most systems that people work on are in this phase. Many programmers and software designers *never* get to work on new development; they work under the constraints of the existing architecture and the existing body of code. Virtually all software systems change over their lifetime, to accommodate new features, to adapt to new environments, to fix bugs, and so forth. But these changes are often fraught with difficulty.

Every architecture partitions possible changes into three categories: local, nonlocal, and architectural.

- A local change can be accomplished by modifying a single element. For example, adding a new business rule to a pricing logic module.
- A nonlocal change requires multiple element modifications but leaves the underlying architectural approach intact. For example, adding a new business rule to a pricing logic module, then adding new fields to the database that this new business rule requires, and then revealing the results of the rule in the user interface.
- An architectural change affects the fundamental ways in which the elements interact with each other and will probably require changes all over the system. For example, changing a system from client-server to peer-to-peer.

Obviously, local changes are the most desirable, and so an *effective* architecture is one in which the most common changes are local, and hence easy to make.

Deciding when changes are essential, determining which change paths have the least risk, assessing the consequences of proposed changes, and arbitrating sequences and priorities for requested changes all require broad insight into relationships, performance, and behaviors of system software elements. These activities are in the job description for an architect. Reasoning about the architecture and analyzing the architecture can provide the insight necessary to make decisions about anticipated changes.

2.3 Predicting System Qualities

This point follows from the previous two. Architecture not only imbues systems with qualities, but it does so in a predictable way.

Were it not possible to tell that the appropriate architectural decisions have been made (i.e., if the system will exhibit its required quality attributes) without waiting until the system is developed and deployed, then choosing an architecture would be a hopeless task—randomly making architecture selections would perform as well as any other method. Fortunately, it *is* possible to make quality predictions about a system based solely on an evaluation of its architecture. If we know that certain kinds of architectural decisions lead to certain quality attributes in a system, then we can make those decisions and rightly expect to be rewarded with the associated quality attributes. After the fact, when we examine an architecture, we can look to see if those decisions have been made, and confidently predict that the architecture will exhibit the associated qualities.

This is no different from any mature engineering discipline, where design analysis is a standard part of the development process. The earlier you can find a problem in your design, the cheaper, easier, and less disruptive it will be to fix.

Even if you don't do the quantitative analytic modeling sometimes necessary to ensure that an architecture will deliver its prescribed benefits, this principle of evaluating decisions based on their quality attribute implications is invaluable for at least spotting potential trouble spots early.

The architecture modeling and analysis techniques described in Chapter 14, as well as the architecture evaluation techniques covered in Chapter 21, allow early insight into the software product qualities made possible by software architectures.

2.4 Enhancing Communication among Stakeholders

Software architecture represents a common abstraction of a system that most, if not all, of the system's stakeholders can use as a basis for creating mutual understanding, negotiating, forming consensus, and communicating with each other. The architecture—or at least parts of it—is sufficiently abstract that most nontechnical people can understand it adequately, particularly with some coaching from the architect, and yet that abstraction can be refined into sufficiently rich technical specifications to guide implementation, integration, test, and deployment.

Each stakeholder of a software system—customer, user, project manager, coder, tester, and so on—is concerned with different characteristics of the system that are affected by its architecture. For example:

- The user is concerned that the system is fast, reliable, and available when needed.
- The customer is concerned that the architecture can be implemented on schedule and according to budget.
- The manager is worried (in addition to concerns about cost and schedule) that the architecture will allow teams to work largely independently, interacting in disciplined and controlled ways.
- The architect is worried about strategies to achieve all of those goals.

Architecture provides a common language in which different concerns can be expressed, negotiated, and resolved at a level that is intellectually manageable even for large, complex systems. Without such a language, it is difficult to understand large systems sufficiently to make the early decisions that influence both quality and usefulness. Architectural analysis, as we will see in Chapter 21, both depends on this level of communication and enhances it.

Section 3.5 covers stakeholders and their concerns in greater depth.

"What Happens When I Push This Button?" Architecture as a Vehicle for Stakeholder Communication

The project review droned on and on. The government-sponsored development was behind schedule and over budget and was large enough that these lapses were attracting congressional attention. And now the government was making up for past neglect by holding a marathon come-one-come-all review session. The contractor had recently undergone a buyout, which hadn't helped matters. It was the afternoon of the second day, and the agenda called for the software architecture to be presented. The young architect—an apprentice to the chief architect for the system—was bravely explaining how the software architecture for the massive system would enable it to meet its very demanding real-time, distributed, high-reliability requirements. He had a solid presentation and a solid architecture to present. It was sound and sensible. But the audience—about 30 government representatives who had varying roles in the management and oversight of this sticky project—was tired. Some of them were even thinking that perhaps they should have gone into real estate instead of enduring another one of these marathon let's-finally-get-it-right-this-time reviews.

The viewgraph showed, in semiformal box-and-line notation, what the major software elements were in a runtime view of the system. The names were all acronyms, suggesting no semantic meaning without explanation, which the young architect gave. The lines showed data flow, message passing, and process synchronization. The elements were internally redundant, the architect was explaining. "In the event of a failure," he began, using a laser pointer to denote one of the lines, "a restart mechanism triggers along this path when—"

"What happens when the mode select button is pushed?" interrupted one of the audience members. He was a government attendee representing the user community for this system.

"Beg your pardon?" asked the architect.

"The mode select button," he said. "What happens when you push it?"

"Um, that triggers an event in the device driver, up here," began the architect, laser-pointing. "It then reads the register and interprets the event code. If it's mode select, well, then, it signals the blackboard, which in turns signals the objects that have subscribed to that event. . . . "

"No, I mean what does the system *do,*" interrupted the questioner. "Does it reset the displays? And what happens if this occurs during a system reconfiguration?"

The architect looked a little surprised and flicked off the laser pointer. This was not an architectural question, but since he was an architect and therefore fluent in the requirements, he knew the answer. "If the command line is in setup mode, the displays will reset," he said. "Otherwise an error message will be put on the control console, but the signal will be ignored." He put the laser pointer back on. "Now, the restart mechanism that I was talking about—"

"Well, I was just wondering," said the users' delegate. "Because I see from your chart that the display console is sending signal traffic to the target location module."

"What *should* happen?" asked another member of the audience, addressing the first questioner. "Do you really want the user to get mode data during its reconfiguring?" And for the next 45 minutes, the architect watched as the audience consumed his time slot by debating what the correct behavior of the system was supposed to be in various esoteric states.

The debate was not architectural, but the architecture (and the graphical rendition of it) had sparked debate. It is natural to think of architecture as the basis for communication among some of the stakeholders besides the architects and developers: Managers, for example, use the architecture to create teams and allocate resources among them. But users? The architecture is invisible to users, after all; why should they latch on to it as a tool for understanding the system?

The fact is that they do. In this case, the questioner had sat through two days of viewgraphs all about function, operation, user interface, and testing. But it was the first slide on architecture that—even though he was tired and wanted to go home—made him realize he didn't understand something. Attendance at many architecture reviews has convinced me that seeing the system in a new way prods the mind and brings new questions to the surface. For users, architecture often serves as that new way, and the questions that a user poses will be behavioral in nature. In a memorable architecture evaluation exercise a few years ago, the user representatives were much more interested in what the system was going to do than in how it was going to do it, and naturally so. Up until that point, their only contact with the vendor had been through its marketers. The architect was the first legitimate expert on the system to whom they had access, and they didn't hesitate to seize the moment.

Of course, careful and thorough requirements specifications would ameliorate this situation, but for a variety of reasons they are not always created or available. In their absence, a specification of the architecture often serves to trigger questions and improve clarity. It is probably more prudent to recognize this reality than to resist it.

Sometimes such an exercise will reveal unreasonable requirements, whose utility can then be revisited. A review of this type that emphasizes synergy between requirements and architecture would have let the young architect in our story off the hook by giving him a place in the overall review session to address that kind of information. And the user representative wouldn't have felt like a fish out of water, asking his question at a clearly inappropriate moment.

—PCC

2.5 Carrying Early Design Decisions

Software architecture is a manifestation of the earliest design decisions about a system, and these early bindings carry enormous weight with respect to the system's remaining development, its deployment, and its maintenance life. It is also the earliest point at which these important design decisions affecting the system can be scrutinized.

Any design, in any discipline, can be viewed as a set of decisions. When painting a picture, an artist decides on the material for the canvas, on the media for recording—oil paint, watercolor, crayon—even before the picture is begun. Once the picture is begun, other decisions are immediately made: Where is the first line? What is its thickness? What is its shape? All of these early design decisions have a strong influence on the final appearance of the picture. Each decision constrains the many decisions that follow. Each decision, in isolation, might appear innocent enough, but the early ones in particular have disproportionate weight simply because they influence and constrain so much of what follows.

So it is with architecture design. An architecture design can also be viewed as a set of decisions. The early design decisions constrain the decisions that follow, and changing these decisions has enormous ramifications. Changing these early decisions will cause a ripple effect, in terms of the additional decisions that must now be changed. Yes, sometimes the architecture must be refactored or redesigned, but this is not a task we undertake lightly (because the "ripple" might turn into a tsunami).

What are these early design decisions embodied by software architecture? Consider:

- Will the system run on one processor or be distributed across multiple processors?
- Will the software be layered? If so, how many layers will there be? What will each one do?
- Will components communicate synchronously or asynchronously? Will they interact by transferring control or data or both?
- Will the system depend on specific features of the operating system or hardware?
- Will the information that flows through the system be encrypted or not?
- What operating system will we use?
- What communication protocol will we choose?

Imagine the nightmare of having to change any of these or a myriad other related decisions. Decisions like these begin to flesh out some of the structures of the architecture and their interactions. In Chapter 4, we describe seven categories of these early design decisions. In Chapters 5–11 we show the implications of these design decision categories on achieving quality attributes.

2.6 Defining Constraints on an Implementation

An implementation exhibits an architecture if it conforms to the design decisions prescribed by the architecture. This means that the implementation must be implemented as the set of prescribed elements, these elements must interact with each other in the prescribed fashion, and each element must fulfill its responsibility to the other elements as dictated by the architecture. Each of these prescriptions is a constraint on the implementer.

Element builders must be fluent in the specifications of their individual elements, but they may not be aware of the architectural tradeoffs—the architecture (or architect) simply constrains them in such a way as to meet the tradeoffs. A classic example of this phenomenon is when an architect assigns performance budget to the pieces of software involved in some larger piece of functionality. If each software unit stays within its budget, the overall transaction will meet its performance requirement. Implementers of each of the constituent pieces may not know the overall budget, only their own.

Conversely, the architects need not be experts in all aspects of algorithm design or the intricacies of the programming language—although they should certainly know enough not to design something that is difficult to build—but they are the ones responsible for establishing, analyzing, and enforcing the architectural tradeoffs.

2.7 Influencing the Organizational Structure

Not only does architecture prescribe the structure of the system being developed, but that structure becomes engraved in the structure of the development project (and sometimes the structure of the entire organization). The normal method for dividing up the labor in a large project is to assign different groups different portions of the system to construct. This is called the work-breakdown structure of a system. Because the architecture includes the broadest decomposition of the system, it is typically used as the basis for the work-breakdown structure. The work-breakdown structure in turn dictates units of planning, scheduling, and budget; interteam communication channels; configuration control and file-system organization; integration and test plans and procedures; and even project minutiae such as how the project intranet is organized and who sits with whom at the company picnic. Teams communicate with each other in terms of the interface specifications for the major elements. The maintenance activity, when launched, will also reflect the software structure, with teams formed to maintain specific structural elements from the architecture: the database, the business rules, the user interface, the device drivers, and so forth.

A side effect of establishing the work-breakdown structure is to freeze some aspects of the software architecture. A group that is responsible for one of the subsystems will resist having its responsibilities distributed across other groups. If these responsibilities have been formalized in a contractual relationship, changing responsibilities could become expensive or even litigious.

Thus, once the architecture has been agreed on, it becomes very costly—for managerial and business reasons—to significantly modify it. This is one argument (among many) for carrying out extensive analysis before settling on the software architecture for a large system—because so much depends on it.

2.8 Enabling Evolutionary Prototyping

Once an architecture has been defined, it can be analyzed and prototyped as a skeletal system. A skeletal system is one in which at least some of the infrastructure—how the elements initialize, communicate, share data, access resources, report errors, log activity, and so forth—is built before much of the system's functionality has been created. (The two can go hand in hand: build a little infrastructure to support a little end-to-end functionality; repeat until done.)

For example, systems built as plug-in architectures are skeletal systems: the plug-ins provide the actual functionality. This approach aids the development process because the system is executable early in the product's life cycle. The fidelity of the system increases as stubs are instantiated, or prototype parts are replaced with complete versions of these parts of the software. In some cases the prototype parts can be low-fidelity versions of the final functionality, or they can be *surrogates* that consume and produce data at the appropriate rates but do little else. Among other things, this approach allows potential performance problems to be identified early in the product's life cycle.

These benefits reduce the potential risk in the project. Furthermore, if the architecture is part of a family of related systems, the cost of creating a framework for prototyping can be distributed over the development of many systems.

2.9 Improving Cost and Schedule Estimates

Cost and schedule estimates are important tools for the project manager both to acquire the necessary resources and to monitor progress on the project, to know if and when a project is in trouble. One of the duties of an architect is to help the project manager create cost and schedule estimates early in the project life cycle. Although top-down estimates are useful for setting goals and apportioning budgets, cost estimations that are based on a bottom-up understanding of the system's pieces are typically more accurate than those that are based purely on top-down system knowledge.

As we have said, the organizational and work-breakdown structure of a project is almost always based on its architecture. Each team or individual responsible for a work item will be able to make more-accurate estimates for their piece than a project manager and will feel more ownership in making the estimates come true. But the best cost and schedule estimates will typically emerge from a consensus between the top-down estimates (created by the architect and project manager) and the bottom-up estimates (created by the developers). The discussion and negotiation that results from this process creates a far more accurate estimate than either approach by itself.

It helps if the requirements for a system have been reviewed and validated. The more up-front knowledge you have about the scope, the more accurate the cost and schedule estimates will be.

Chapter 22 delves into the use of architecture in project management.

2.10 Supplying a Transferable, Reusable Model

The earlier in the life cycle that reuse is applied, the greater the benefit that can be achieved. While code reuse provides a benefit, reuse of architectures provides tremendous leverage for systems with similar requirements. Not only can code be reused, but so can the requirements that led to the architecture in the first place, as well as the experience and infrastructure gained in building the reused architecture. When architectural decisions can be reused across multiple systems, all of the early-decision consequences we just described are also transferred.

A software product line or family is a set of software systems that are all built using the same set of reusable assets. Chief among these assets is the architecture that was designed to handle the needs of the entire family. Product-line architects choose an architecture (or a family of closely related architectures) that will serve all envisioned members of the product line. The architecture defines what is fixed for all members of the product line and what is variable. Software product lines represent a powerful approach to multi-system development that is showing order-of-magnitude payoffs in time to market, cost, productivity, and product quality. The power of architecture lies at the heart of the paradigm. Similar to other capital investments, the architecture for a product line becomes a developing organization's core asset. Software product lines are explained in Chapter 25.

2.11 Allowing Incorporation of Independently Developed Components

Whereas earlier software paradigms have focused on *programming* as the prime activity, with progress measured in lines of code, architecture-based development often focuses on *composing* or *assembling elements* that are likely to have been developed separately, even independently, from each other. This composition is possible because the architecture defines the elements that can be incorporated into the system. The architecture constrains possible replacements (or additions) according to how they interact with their environment, how they receive and relinquish control, what data they consume and produce, how they access data, and what protocols they use for communication and resource sharing.

In 1793, Eli Whitney's mass production of muskets, based on the principle of interchangeable parts, signaled the dawn of the industrial age. In the days before physical measurements were reliable, manufacturing interchangeable parts was a daunting notion. Today in software, until abstractions can be reliably delimited, the notion of structural interchangeability is just as daunting and just as significant.

Commercial off-the-shelf components, open source software, publicly available apps, and networked services are all modern-day software instantiations of Whitney's basic idea. Whitney's musket parts had "interfaces" (having to do with fit and durability) and so do today's interchangeable software components.

For software, the payoff can be

- Decreased time to market (it should be easier to use someone else's ready solution than build your own)
- Increased reliability (widely used software should have its bugs ironed out already)
- Lower cost (the software supplier can amortize development cost across their customer base)
- Flexibility (if the component you want to buy is not terribly special-purpose, it's likely to be available from several sources, thus increasing your buying leverage)

2.12 Restricting the Vocabulary of Design Alternatives

As useful architectural patterns are collected, it becomes clear that although software elements can be combined in more or less infinite ways, there is something to be gained by voluntarily restricting ourselves to a relatively small number of choices of elements and their interactions. By doing so we minimize the design complexity of the system we are building.

A software engineer is not an *artiste*, whose creativity and freedom are paramount. Engineering is about discipline, and discipline comes in part by *restricting* the vocabulary of alternatives to proven solutions. Advantages of this approach include enhanced reuse, more regular and simpler designs that are more easily understood and communicated, more capable analysis, shorter selection time, and greater interoperability. Architectural patterns guide the architect and focus the architect on the quality attributes of interest in large part by restricting the vocabulary of design alternatives to a relatively small number.

Properties of software design follow from the choice of an architectural pattern. Those patterns that are more desirable for a particular problem should improve the implementation of the resulting design solution, perhaps by making it easier to arbitrate conflicting design constraints, by increasing insight into poorly understood design contexts, or by helping to surface inconsistencies in requirements. We will discuss architectural patterns in more detail in Chapter 13.

2.13 Providing a Basis for Training

The architecture, including a description of how the elements interact with each other to carry out the required behavior, can serve as the first introduction to the system for new project members. This reinforces our point that one of the important uses of software architecture is to support and encourage communication among the various stakeholders. The architecture is a common reference point.

Module views are excellent for showing someone the structure of a project: Who does what, which teams are assigned to which parts of the system, and so forth. Component-and-connector views are excellent for explaining how the system is expected to work and accomplish its job.

We will discuss these views in more detail in Chapter 18.

2.14 Summary

Software architecture is important for a wide variety of technical and nontechnical reasons. Our list includes the following:

1. An architecture will inhibit or enable a system's driving quality attributes.
2. The decisions made in an architecture allow you to reason about and manage change as the system evolves.
3. The analysis of an architecture enables early prediction of a system's qualities.
4. A documented architecture enhances communication among stakeholders.
5. The architecture is a carrier of the earliest and hence most fundamental, hardest-to-change design decisions.
6. An architecture defines a set of constraints on subsequent implementation.
7. The architecture dictates the structure of an organization, or vice versa.
8. An architecture can provide the basis for evolutionary prototyping.
9. An architecture is the key artifact that allows the architect and project manager to reason about cost and schedule.
10. An architecture can be created as a transferable, reusable model that forms the heart of a product line.
11. Architecture-based development focuses attention on the assembly of components, rather than simply on their creation.
12. An architecture channels the creativity of developers, reducing design and system complexity.
13. An architecture can be the foundation for training of a new team member.

2.15 For Further Reading

Rebecca Grinter has observed architects from a sociological standpoint. In [Grinter 99] she argues eloquently that the architect's primary role is to facilitate stakeholder communication. The way she puts it is that architects enable communication among parties who would otherwise not be able to talk to each other.

The granddaddy of papers about architecture and organization is [Conway 68]. Conway's law states that "organizations which design systems . . . are constrained to produce designs which are copies of the communication structures of these organizations."

There is much about software development through composition that remains unresolved. When the components that are candidates for importation and reuse are distinct subsystems that have been built with conflicting architectural assumptions, unanticipated complications can increase the effort required to integrate their functions. David Garlan and his colleagues coined the term *architectural mismatch* to describe this situation, and their paper on it is worth reading [Garlan 95].

Paulish [Paulish 02] discusses architecture-based project management, and in particular the ways in which an architecture can help in the estimation of project cost and schedule.

2.16 Discussion Questions

1. For each of the thirteen reasons articulated in this chapter why architecture is important, take the contrarian position: Propose a set of circumstances under which architecture is not necessary to achieve the result indicated. Justify your position. (Try to come up with different circumstances for each of the thirteen.)

2. This chapter argues that architecture brings a number of tangible benefits. How would you measure the benefits, on a particular project, of each of the thirteen points?

3. Suppose you want to introduce architecture-centric practices to your organization. Your management is open to the idea, but wants to know the ROI for doing so. How would you respond?

4. Prioritize the list of thirteen points in this chapter according to some criteria meaningful to you. Justify your answer. Or, if you could choose only two or three of the reasons to promote the use of architecture in a project, which would you choose and why?

3

The Many Contexts of Software Architecture

People in London think of London as the center of the world, whereas New Yorkers think the world ends three miles outside of Manhattan.
—Toby Young

In 1976, a *New Yorker* magazine cover featured a cartoon by Saul Steinberg showing a New Yorker's view of the world. You've probably seen it; if not, you can easily find it online. Looking to the west from 9th Avenue in Manhattan, the illustration shows 10th Avenue, then the wide Hudson River, then a thin strip of completely nondescript land called "Jersey," followed by a somewhat thicker strip of land representing the entire rest of the United States. The mostly empty United States has a cartoon mountain or two here and there and a few cities haphazardly placed "out there," and is flanked by featureless "Canada" on the right and "Mexico" on the left. Beyond is the Pacific Ocean, only slightly wider than the Hudson, and beyond that lie tiny amorphous shapes for Japan and China and Russia, and that's pretty much the world from a New Yorker's perspective.

In a book about architecture, it is tempting to view architecture in the same way, as the most important piece of the software universe. And in some chapters, we unapologetically will do exactly that. But in this chapter we put software architecture in its place, showing how it supports and is informed by other critical forces and activities in the various contexts in which it plays a role.

These contexts, around which we structured this book, are as follows:

- *Technical.* What technical role does the software architecture play in the system or systems of which it's a part?
- *Project life cycle.* How does a software architecture relate to the other phases of a software development life cycle?
- *Business.* How does the presence of a software architecture affect an organization's business environment?

- *Professional.* What is the role of a software architect in an organization or a development project?

These contexts all play out throughout the book, but this chapter introduces each one. Although the contexts are unchanging, the specifics for your system may change over time. One challenge for the architect is to envision what in their context might change and to adopt mechanisms to protect the system and its development if the envisioned changes come to pass.

3.1 Architecture in a Technical Context

Architectures inhibit or enable the achievement of quality attributes, and one use of an architecture is to support reasoning about the consequences of change in the particular quality attributes important for a system at its inception.

Architectures Inhibit or Enable the Achievement of Quality Attributes

Chapter 2 listed thirteen reasons why software architecture is important and merits study. Several of those reasons deal with exigencies that go beyond the bounds of a particular development project (such as communication among stakeholders, many of whom may reside outside the project's organization). Others deal with nontechnical aspects of a project (such as the architecture's influence on a project's team structure, or its contribution to accurate budget and schedule estimation). The first three reasons in that List of Thirteen deal specifically with an architecture's technical impact on every system that uses it:

1. An architecture will inhibit or enable the achievement of a system's quality attributes.
2. You can predict many aspects of a system's qualities by studying its architecture.
3. An architecture makes it easier for you to reason about and manage change.

These are all about the architecture's effect on a system's quality attributes, although the first one states it the most explicitly. While all of the reasons enumerated in Chapter 2 are valid statements of the contribution of architecture, probably the most important reason that it warrants attention is its critical effect on quality attributes.

This is such a critical point that, with your indulgence, we'll add a few more points to the bullet list that we gave in Section 2.1. Remember? The one that started like this:

- If your system requires high performance, then you need to pay attention to managing the time-based behavior of elements, their use of shared resources, and the frequency and volume of interelement communication.

To that list, we'll add the following:

- If you care about a system's availability, you *have* to be concerned with how components take over for each other in the event of a failure, and how the system responds to a fault.
- If you care about usability, you *have* to be concerned about isolating the details of the user interface and those elements responsible for the user experience from the rest of the system, so that those things can be tailored and improved over time.
- If you care about the testability of your system, you *have* to be concerned about the testability of individual elements, which means making their state observable and controllable, plus understanding the emergent behavior of the elements working together.
- If you care about the safety of your system, you *have* to be concerned about the behavioral envelope of the elements and the emergent behavior of the elements working in concert.
- If you care about interoperability between your system and another, you *have* to be concerned about which elements are responsible for external interactions so that you can control those interactions.

These and other representations are all saying the same thing in different ways: If you care about *this* quality attribute, you have to be concerned with *these* decisions, all of which are thoroughly architectural in nature. An architecture inhibits or enables a system's quality attributes. And conversely, nothing else influences an architecture more than the quality attribute requirements it must satisfy.

If you care about architecture for no other reason, you should care about it for this one. We feel so strongly about architecture's importance with respect to achieving system quality attributes that all of Part II of this book is devoted to the topic.

Why is functionality missing from the preceding list? It is missing because the architecture mainly provides containers into which the architect places functionality. Functionality is not so much a driver for the architecture as it is a consequence of it. We return to this point in more detail in Part II.

Architectures and the Technical Environment

The technical environment that is current when an architecture is designed will influence that architecture. It might include standard industry practices or software engineering techniques prevalent in the architect's professional community. It is a brave architect who, in today's environment, does not at least consider a web-based, object-oriented, service-oriented, mobility-aware, cloud-based, social-networking-friendly design for an information system. It wasn't always so, and it won't be so ten years from now when another crop of technological trends has come to the fore.

The Swedish Ship *Vasa*

In the 1620s, Sweden and Poland were at war. The king of Sweden, Gustavus Adolphus, was determined to put a swift and favorable end to it and commissioned a new warship the likes of which had never been seen before. The *Vasa*, shown in Figure 3.1, was to be the world's most formidable instrument of war: 70 meters long, able to carry 300 soldiers, and with an astonishing 64 heavy guns mounted on two gun decks. Seeking to add overwhelming firepower to his navy to strike a decisive blow, the king insisted on stretching the *Vasa*'s armaments to the limits. Her architect, Henrik Hybertsson, was a seasoned Dutch shipbuilder with an impeccable reputation, but the *Vasa* was beyond even his broad experience. Two-gun-deck ships were rare, and none had been built of the *Vasa*'s size and armament.

Like all architects of systems that push the envelope of experience, Hybertsson had to balance many concerns. Swift time to deployment was critical, but so were performance, functionality, safety, reliability, and cost.

FIGURE 3.1 The warship. Used with permission of The Vasa Museum, Stockholm, Sweden.

He was also responsible to a variety of stakeholders. In this case, the primary customer was the king, but Hybertsson also was responsible to the crew that would sail his creation. Also like all architects, Hybertsson brought his experience with him to the task. In this case, his experience told him to design the *Vasa* as though it were a single-gun-deck ship and then extrapolate, which was in accordance with the technical environment of the day. Faced with an impossible task, Hybertsson had the good sense to die about a year before the ship was finished.

The project was completed to his specifications, however, and on Sunday morning, August 10, 1628, the mighty ship was ready. She set her sails, waddled out into Stockholm's deep-water harbor, fired her guns in salute, and promptly rolled over. Water poured in through the open gun ports, and the *Vasa* plummeted. A few minutes later her first and only voyage ended 30 meters beneath the surface. Dozens among her 150-man crew drowned.

Inquiries followed, which concluded that the ship was well built but "badly proportioned." In other words, its architecture was flawed. Today we know that Hybertsson did a poor job of balancing all of the conflicting constraints levied on him. In particular, he did a poor job of risk management and a poor job of customer management (not that anyone could have fared better). He simply acquiesced in the face of impossible requirements.

The story of the *Vasa*, although more than 375 years old, well illustrates the Architecture Influence Cycle: organization goals beget requirements, which beget an architecture, which begets a system. The architecture flows from the architect's experience and the technical environment of the day. Hybertsson suffered from the fact that neither of those were up to the task before him.

In this book, we provide three things that Hybertsson could have used:

1. Examples of successful architectural practices that work under demanding requirements, so as to help set the technical playing field of the day.
2. Methods to assess an architecture before any system is built from it, so as to mitigate the risks associated with launching unprecedented designs.
3. Techniques for incremental architecture-based development, so as to uncover design flaws before it is too late to correct them.

Our goal is to give architects another way out of their design dilemmas than the one that befell the ill-fated Dutch ship designer. Death before deployment is not nearly so admired these days.

—PCC

3.2 Architecture in a Project Life-Cycle Context

Software development processes are standard approaches for developing software systems. They impose a discipline on software engineers and, more important, teams of software engineers. They tell the members of the team what to do next. There are four dominant software development processes, which we describe in roughly the order in which they came to prominence:

1. *Waterfall*. For many years the Waterfall model dominated the field of software development. The Waterfall model organized the life cycle into a series of connected sequential activities, each with entry and exit conditions and a formalized relationship with its upstream and downstream neighbors. The process began with requirements specification, followed by design, then implementation, then integration, then testing, then installation, all followed by maintenance. Feedback paths from later to earlier steps allowed for the revision of artifacts (requirements documents, design documents, etc.) on an as-needed basis, based on the knowledge acquired in the later stage. For example, designers might push back against overly stringent requirements, which would then be reworked and flow back down. Testing that uncovered defects would trigger reimplementation (and maybe even redesign). And then the cycle continued.
2. *Iterative*. Over time the feedback paths of the Waterfall model became so pronounced that it became clear that it was better to think of software development as a series of short cycles through the steps—*some* requirements lead to *some* design, which can be implemented and tested while the next cycle's worth of requirements are being captured and designed. These cycles are called iterations, in the sense of iterating toward the ultimate software solution for the given problem. Each iteration should deliver something working and useful. The trick here is to uncover early those requirements that have the most far-reaching effect on the design; the corresponding danger is to overlook requirements that, when discovered later, will capsize the design decisions made so far. An especially well-known iterative process is called the Unified Process (originally named the Rational Unified Process, after Rational Software, which originated it). It defines four phases of each iteration: inception, elaboration, construction, and transition. A set of chosen use cases defines the goals for each iteration, and the iterations are ordered to address the greatest risks first.
3. *Agile*. The term "Agile software development" refers to a group of software development methodologies, the best known of which include Scrum, Extreme Programming, and Crystal Clear. These methodologies are all incremental and iterative. As such, one can consider some iterative

methodologies as Agile. What distinguishes Agile practices is early and frequent delivery of working software, close collaboration between developers and customers, self-organizing teams, and a focus on adaptation to changing circumstances (such as late-arriving requirements). All Agile methodologies focus on teamwork, adaptability, and close collaboration (both within the team and between team members and customers/end users). These methodologies typically eschew substantial up-front work, on the assumption that requirements *always* change, and they continue to change throughout the project's life cycle. As such, it might seem that Agile methodologies and architecture cannot happily coexist. As we will show in Chapter 15, this is not so.

4. *Model-driven development*. Model-driven development is based on the idea that humans should not be writing code in programming languages, but they should be creating models of the domain, from which code is automatically generated. Humans create a platform-independent model (PIM), which is combined with a platform-definition model (PDM) to generate running code. In this way the PIM is a pure realization of the functional requirements while the PDM addresses platform specifics and quality attributes.

All of these processes include design among their obligations, and because architecture is a special kind of design, architecture finds a home in each one. Changing from one development process to another in the middle of a project requires the architect to save useful information from the old process and determine how to integrate it into the new process.

No matter what software development process or life-cycle model you're using, there are a number of activities that are involved in creating a software architecture, using that architecture to realize a complete design, and then implementing or managing the evolution of a target system or application. The process you use will determine how often and when you revisit and elaborate each of these activities. These activities include:

1. Making a business case for the system
2. Understanding the architecturally significant requirements
3. Creating or selecting the architecture
4. Documenting and communicating the architecture
5. Analyzing or evaluating the architecture
6. Implementing and testing the system based on the architecture
7. Ensuring that the implementation conforms to the architecture

Each of these activities is covered in a chapter in Part III of this book, and described briefly below.

Making a Business Case for the System

A business case is, briefly, a justification of an organizational investment. It is a tool that helps you make business decisions by predicting how they will affect your organization. Initially, the decision will be a go/no-go for pursuing a new business opportunity or approach. After initiation, the business case is reviewed to assess the accuracy of initial estimates and then updated to examine new or alternative angles on the opportunity. By documenting the expected costs, benefits, and risks, the business case serves as a repository of the business and marketing data. In this role, management uses the business case to determine possible courses of action.

Knowing the business goals for the system—Chapter 16 will show you how to elicit and capture them in a systematic way—is also critical in the creation of a business case for a system.

Creating a business case is broader than simply assessing the market need for a system. It is an important step in shaping and constraining any future requirements. How much should the product cost? What is its targeted market? What is its targeted time to market and lifetime? Will it need to interface with other systems? Are there system limitations that it must work within?

These are all questions about which the system's architects have specialized knowledge; they must contribute to the answers. These questions cannot be decided solely by an architect, but if an architect is not consulted in the creation of the business case, the organization may be unable to achieve its business goals. Typically, a business case is created prior to the initiation of a project, but it also may be revisited during the course of the project for the organization to determine whether to continue making investments in the project. If the circumstances assumed in the initial version of the business case change, the architect may be called upon to establish how the system will change to reflect the new set of circumstances.

Understanding the Architecturally Significant Requirements

There are a variety of techniques for eliciting requirements from the stakeholders. For example, object-oriented analysis uses use cases and scenarios to embody requirements. Safety-critical systems sometimes use more rigorous approaches, such as finite-state-machine models or formal specification languages. In Part II of this book, which covers quality attributes, we introduce a collection of quality attribute scenarios that aid in the brainstorming, discussion, and capture of quality attribute requirements for a system.

One fundamental decision with respect to the system being built is the extent to which it is a variation on other systems that have been constructed. Because it is a rare system these days that is not similar to other systems, requirements elicitation techniques involve understanding these prior systems' characteristics. We discuss the architectural implications of software product lines in Chapter 25.

Another technique that helps us understand requirements is the creation of prototypes. Prototypes may help to model and explore desired behavior, design the user interface, or analyze resource utilization. This helps to make the system "real" in the eyes of its stakeholders and can quickly build support for the project and catalyze decisions on the system's design and the design of its user interface.

Creating or Selecting the Architecture

In the landmark book *The Mythical Man-Month*, Fred Brooks argues forcefully and eloquently that conceptual integrity is the key to sound system design and that conceptual integrity can only be had by a small number of minds coming together to design the system's architecture. We firmly believe this as well. Good architecture almost never results as an emergent phenomenon.

Chapters 5–12 and 17 will provide practical techniques that will aid you in creating an architecture to achieve its behavioral and quality requirements.

Documenting and Communicating the Architecture

For the architecture to be effective as the backbone of the project's design, it must be communicated clearly and unambiguously to all of the stakeholders. Developers must understand the work assignments that the architecture requires of them, testers must understand the task structure that the architecture imposes on them, management must understand the scheduling implications it contains, and so forth.

Toward this end, the architecture's documentation should be informative, unambiguous, and readable by many people with varied backgrounds. Architectural documentation should also be *minimal* and aimed at the stakeholders who will use it; we are no fans of documentation for documentation's sake. We discuss the documentation of architectures and provide examples of good documentation practices in Chapter 18. We will also discuss keeping the architecture up to date when there is a change in something on which the architecture documentation depends.

Analyzing or Evaluating the Architecture

In any design process there will be multiple candidate designs considered. Some will be rejected immediately. Others will contend for primacy. Choosing among these competing designs in a rational way is one of the architect's greatest challenges.

Evaluating an architecture for the qualities that it supports is essential to ensuring that the system constructed from that architecture satisfies its stakeholders' needs. Analysis techniques to evaluate the quality attributes that an architecture imparts to a system have become much more widespread in the past decade. Scenario-based techniques provide one of the most general and effective approaches for evaluating an architecture. The most mature methodological approach is found in the Architecture Tradeoff Analysis Method (ATAM) of Chapter 21, while the economic implications of architectural decisions are explored in Chapter 23.

Implementing and Testing the System Based on the Architecture

If the architect designs and analyzes a beautiful, conceptually sound architecture which the implementers then ignore, what was the point? If architecture is important enough to devote the time and effort of your best minds to, then it is just as important to keep the developers faithful to the structures and interaction protocols constrained by the architecture. Having an explicit and well-communicated architecture is the first step toward ensuring *architectural conformance*. Having an environment or infrastructure that actively assists developers in creating and maintaining the architecture (as opposed to just the code) is better.

There are many reasons why developers might not be faithful to the architecture: It might not have been properly documented and disseminated. It might be too confusing. It might be that the architect has not built ground-level support for the architecture (particularly if it presents a different way of "doing business" than the developers are used to), and so the developers resist it. Or the developers may sincerely want to implement the architecture but, being human, they occasionally slip up. This is not to say that the architecture should not change, but it should not change purely on the basis of the whims of the developers, because they may not have the overall picture.

Ensuring That the Implementation Conforms to the Architecture

Finally, when an architecture is created and used, it goes into a maintenance phase. Vigilance is required to ensure that the actual architecture and its representation remain faithful to each other during this phase. And when they do get significantly out of sync, effort must be expended to either fix the implementation or update the architectural documentation.

Although work in this area is still relatively immature, it has been an area of intense activity in recent years. Chapter 20 will present the current state of recovering an architecture from an existing system and ensuring that it conforms to the specified architecture.

3.3 Architecture in a Business Context

Architectures and systems are not constructed frivolously. They serve some business purposes, although as mentioned before, these purposes may change over time.

Architectures and Business Goals

Systems are created to satisfy the business goals of one or more organizations. Development organizations want to make a profit, or capture market, or stay in business, or help their customers do their jobs better, or keep their staff gainfully employed, or make their stockholders happy, or a little bit of each. Customers have their own goals for acquiring a system, usually involving some aspect of making their lives easier or more productive. Other organizations involved in a project's life cycle, such as subcontractors or government regulatory agencies, have their own goals dealing with the system.

Architects need to understand who the vested organizations are and what their goals are. Many of these goals will have a profound influence on the architecture.

Many business goals will be manifested as quality attribute requirements. In fact, every quality attribute—such as a user-visible response time or platform flexibility or ironclad security or any of a dozen other needs—should originate from some higher purpose that can be described in terms of added value. If we ask, for example, "*Why* do you want this system to have a really fast response time?" we might hear that this will differentiate the product from its competition and let the developing organization capture market share.

Some business goals, however, will not show up in the form of requirements. We know of one software architect who was informed by his manager that the architecture should include a database. The architect was perplexed, because the requirements for the system really didn't warrant a database and the architect's design had nicely avoided putting one in, thereby simplifying the design and lowering the cost of the product. The architect was perplexed, that is, until the manager reminded the architect that the company's database department was currently overstaffed and underworked. They needed something to do! The architect put in the database, and all was well. That kind of business goal—keeping staff gainfully employed—is not likely to show up in any requirements document, but if the architect had failed to meet it, the manager would have considered the architecture as unacceptable, just as the customer would have if it failed to provide a key piece of functionality.

Still other business goals have no effect on the architecture whatsoever. A business goal to lower costs might be realized by asking employees to work from home, or turn the office thermostats down in the winter, or using less paper in the printers. Chapter 16 will deal with uncovering business goals and the requirements they lead to.

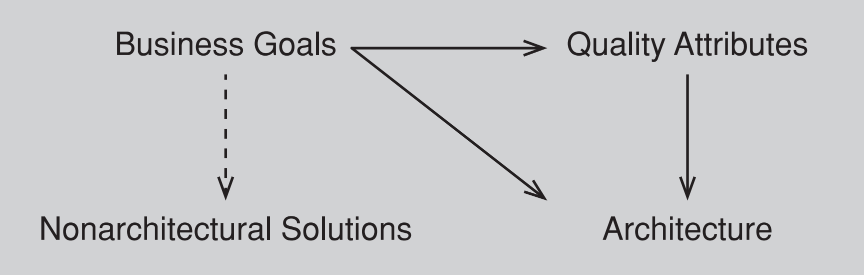


FIGURE 3.2 Some business goals may lead to quality attribute requirements (which lead to architectures), or lead directly to architectural decisions, or lead to nonarchitectural solutions.

Figure 3.2 illustrates the major points from the preceding discussion. In the figure, the arrows mean "leads to." The solid arrows highlight the relationships of most interest to us.

Architectures and the Development Organization

A development organization contributes many of the business goals that influence an architecture. For example, if the organization has an abundance of experienced and idle programmers skilled in peer-to-peer communications, then a peer-to-peer architecture might be the approach supported by management. If not, it may well be rejected. This would support the business goal, perhaps left implicit, of not wanting to hire new staff or lay off existing staff, or not wanting to invest significantly in the retraining of existing staff.

More generally, an organization often has an investment in assets, such as existing architectures and the products based on them. The foundation of a development project may be that the proposed system is the next in a sequence of similar systems, and the cost estimates assume a high degree of asset reuse and a high degree of skill and productivity from the programmers.

Additionally, an organization may wish to make a long-term business investment in an infrastructure to pursue strategic goals and may view the proposed system as one means of financing and extending that infrastructure. For example, an organization may decide that it wants to develop a reputation for supporting solutions based on cloud computing or service-oriented architecture or high-performance real-time computing. This long-term goal would be supported, in part, by infrastructural investments that will affect the developing organization: a cloud-computing group needs to be hired or grown, infrastructure needs to be purchased, or perhaps training needs to be planned.

Finally, the organizational structure can shape the software architecture, and vice versa. Organizations are often organized around technology and application concepts: a database group, a networking group, a business rules team, a user-interface group. So the explicit identification of a distinct subsystem in the architecture will frequently lead to the creation of a group with the name of the subsystem. Furthermore, if the user-interface team frequently needs to communicate with the business rules team, these teams will need to either be co-located or they will need some regular means of communicating and coordinating.

3.4 Architecture in a Professional Context

What do architects do? How do you become an architect? In this section we talk about the many facets of being an architect that go beyond what you learned in a programming or software engineering course.

You probably know by now that architects need more than just technical skills. Architects need to explain to one stakeholder or another the chosen priorities of different properties, and why particular stakeholders are not having all of their expectations fulfilled. To be an effective architect, then, you will need diplomatic, negotiation, and communication skills.

You will perform many activities beyond directly producing an architecture. These activities, which we call *duties*, form the backbone of individual architecture competence. We surveyed the broad body of information aimed at architects (such as websites, courses, books, and position descriptions for architects), as well as practicing architects, and duties are but one aspect. Writers about architects also speak of skills and knowledge. For example, architects need the ability to communicate ideas clearly and need to have up-to-date knowledge about (for example) patterns, or database platforms, or web services standards.

Duties, skills, and knowledge form a triad on which architecture competence rests. You will need to be involved in supporting management and dealing with customers. You will need to manage a diverse workload and be able to switch contexts frequently. You will need to know business considerations. You will need to be a leader in the eyes of developers and management. In Chapter 24 we examine at length the architectural competence of organizations and people.

Architects' Background and Experience

We are all products of our experiences, architects included. If you have had good results using a particular architectural approach, such as three-tier client-server or publish-subscribe, chances are that you will try that same approach on a new development effort. Conversely, if your experience with an approach was disastrous, you may be reluctant to try it again.

Architectural choices may also come from your education and training, exposure to successful architectural patterns, or exposure to systems that have worked particularly poorly or particularly well. You may also wish to experiment with an architectural pattern or technique learned from a book (such as this one) or a training course.

Why do we mention this? Because you (and your organization) must be aware of this influence, so that you can manage it to the best of your abilities. This may mean that you will critically examine proposed architectural solutions, to ensure that they are not simply the path of least resistance. It may mean that you will take training courses in interesting new technologies. It may mean that you will invest in exploratory projects, to "test the water" of a new technology. Each of these steps is a way to proactively manage your background and experience.

3.5 Stakeholders

Many people and organizations are interested in a software system. We call these entities *stakeholders.* A stakeholder is anyone who has a stake in the success of the system: the customer, the end users, the developers, the project manager, the maintainers, and even those who market the system, for example. But stakeholders, despite all having a shared stake in the success of the system, typically have different specific concerns that they wish the system to guarantee or optimize. These concerns are as diverse as providing a certain behavior at runtime, performing well on a particular piece of hardware, being easy to customize, achieving short time to market or low cost of development, gainfully employing programmers who have a particular specialty, or providing a broad range of functions. Figure 3.3 shows the architect receiving a few helpful stakeholder "suggestions."

You will need to know and understand the nature, source, and priority of constraints on the project as early as possible. Therefore, you must identify and actively engage the stakeholders to solicit their needs and expectations. Early engagement of stakeholders allows you to understand the constraints of the task, manage expectations, negotiate priorities, and make tradeoffs. Architecture evaluation (covered in Part III of this book) and iterative prototyping are two means for you to achieve stakeholder engagement.

Having an acceptable system involves appropriate performance, reliability, availability, platform compatibility, memory utilization, network usage, security, modifiability, usability, and interoperability with other systems as well as behavior. All of these qualities, and others, affect how the delivered system is viewed by its eventual recipients, and so such quality attributes will be demanded by one or more of the system's stakeholders.

The underlying problem, of course, is that each stakeholder has different concerns and goals, some of which may be contradictory. It is a rare requirements

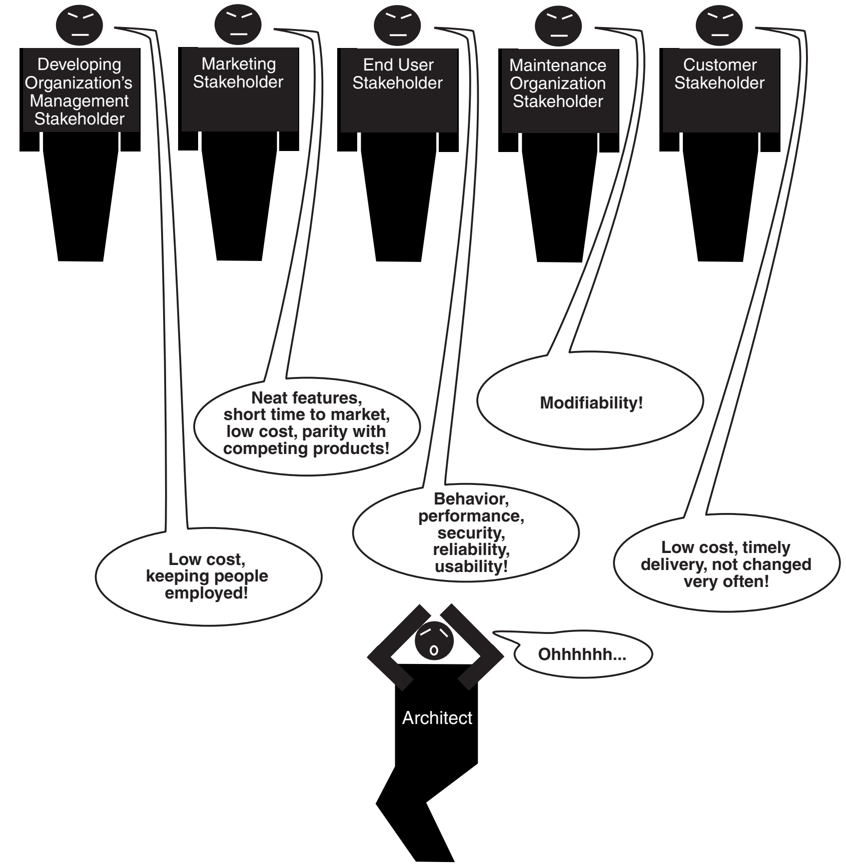

Developing Organization's Management Stakeholder
Marketing Stakeholder
End User Stakeholder
Maintenance Organization Stakeholder
Customer Stakeholder
Neat features, short time to market, low cost, parity with competing products!
Modifiability!
Behavior, performance, security, reliability, usability!
Low cost, keeping people employed!
Low cost, timely delivery, not changed very often!
Ohhhhhh...
Architect

FIGURE 3.3 Influence of stakeholders on the architect

document that does a good job of capturing all of a system's quality requirements in testable detail (a property is testable if it is falsifiable; "make the system easy to use" is not falsifiable but "deliver audio packets with no more than 10 ms. jitter" is falsifiable). The architect often has to fill in the blanks—the quality attribute requirements that have not been explicitly stated—and mediate the conflicts that frequently emerge.

Therefore, one of the best pieces of advice we can give to architects is this: Know your stakeholders. Talk to them, engage them, listen to them, and put yourself in their shoes. Table 3.1 enumerates a set of stakeholders. Notice the remarkable variety and length of this set, but remember that not every stakeholder named in this list may play a role in every system, and one person may play many roles.

TABLE 3.1 Stakeholders for a System and Their Interests

Name	Description	Interest in Architecture
Analyst	Responsible for analyzing the architecture to make sure it meets certain critical quality attribute requirements. Analysts are often specialized; for instance, performance analysts, safety analysts, and security analysts may have well-defined positions in a project.	Analyzing satisfaction of quality attribute requirements of the system based on its architecture.
Architect	Responsible for the development of the architecture and its documentation. Focus and responsibility is on the system.	Negotiating and making tradeoffs among competing requirements and design approaches. A vessel for recording design decisions. Providing evidence that the architecture satisfies its requirements.
Business Manager	Responsible for the functioning of the business/organizational entity that owns the system. Includes managerial/executive responsibility, responsibility for defining business processes, etc.	Understanding the ability of the architecture to meet business goals.
Conformance Checker	Responsible for assuring conformance to standards and processes to provide confidence in a product's suitability.	Basis for conformance checking, for assurance that implementations have been faithful to the architectural prescriptions.
Customer	Pays for the system and ensures its delivery. The customer often speaks for or represents the end user, especially in a government acquisition context.	Assuring required functionality and quality will be delivered; gauging progress; estimating cost; and setting expectations for what will be delivered, when, and for how much.
Database Administrator	Involved in many aspects of the data stores, including database design, data analysis, data modeling and optimization, installation of database software, and monitoring and administration of database security.	Understanding how data is created, used, and updated by other architectural elements, and what properties the data and database must have for the overall system to meet its quality goals.
Deployer	Responsible for accepting the completed system from the development effort and deploying it, making it operational, and fulfilling its allocated business function.	Understanding the architectural elements that are delivered and to be installed at the customer or end user's site, and their overall responsibility toward system function.
Designer	Responsible for systems and/or software design downstream of the architecture, applying the architecture to meet specific requirements of the parts for which they are responsible.	Resolving resource contention and establishing performance and other kinds of runtime resource consumption budgets. Understanding how their part will communicate and interact with other parts of the system.
Evaluator	Responsible for conducting a formal evaluation of the architecture (and its documentation) against some clearly defined criteria.	Evaluating the architecture's ability to deliver required behavior and quality attributes.

Name	Description	Interest in Architecture
Implementer	Responsible for the development of specific elements according to designs, requirements, and the architecture.	Understanding inviolable constraints and exploitable freedoms on development activities.
Integrator	Responsible for taking individual components and integrating them, according to the architecture and system designs.	Producing integration plans and procedures, and locating the source of integration failures.
Maintainer	Responsible for fixing bugs and providing enhancements to the system throughout its life (including adaptation of the system for uses not originally envisioned).	Understanding the ramifications of a change.
Network Administrator	Responsible for the maintenance and oversight of computer hardware and software in a computer network. This may include the deployment, configuration, maintenance, and monitoring of network components.	Determining network loads during various use profiles, understanding uses of the network.
Product-Line Manager	Responsible for development of an entire family of products, all built using the same core assets (including the architecture).	Determining whether a potential new member of a product family is in or out of scope and, if out, by how much.
Project Manager	Responsible for planning, sequencing, scheduling, and allocating resources to develop software components and deliver components to integration and test activities.	Helping to set budget and schedule, gauging progress against established budget and schedule, identifying and resolving development-time resource contention.
Representative of External Systems	Responsible for managing a system with which this one must interoperate, and its interface with our system.	Defining the set of agreement between the systems.
System Engineer	Responsible for design and development of systems or system components in which software plays a role.	Assuring that the system environment provided for the software is sufficient.
Tester	Responsible for the (independent) test and verification of the system or its elements against the formal requirements and the architecture.	Creating tests based on the behavior and interaction of the software elements.
User	The actual end users of the system. There may be distinguished kinds of users, such as administrators, superusers, etc.	Users, in the role of reviewers, might use architecture documentation to check whether desired functionality is being delivered. Users might also use the documentation to understand what the major system elements are, which can aid them in emergency field maintenance.

3.6 How Is Architecture Influenced?

For decades, software designers have been taught to build systems based on the software's technical requirements. In the older Waterfall model, the requirements document is "tossed over the wall" into the designer's cubicle, and the designer must come forth with a satisfactory design. Requirements beget design, which begets system. In an iterative or Agile approach to development, an increment of requirements begets an increment of design, and so forth.

This vision of software development is short-sighted. In any development effort, the requirements make explicit some—but only some—of the desired properties of the final system. Not all requirements are focused directly on desired system properties; some requirements might mandate a development process or the use of a particular tool. Furthermore, the requirements specification only begins to tell the story. Failure to satisfy other constraints may render the system just as problematic as if it functioned poorly.

What do you suppose would happen if two different architects, working in two different organizations, were given the same requirements specification for a system? Do you think they would produce the same architecture or different ones?

The answer is that they would very likely produce different ones, which immediately belies the notion that requirements *determine* architecture. Other factors are at work.

A software architecture is a result of business and social influences, as well as technical ones. The existence of an architecture in turn affects the technical, business, and social environments that subsequently influence future architectures. In particular, each of the contexts for architecture that we just covered—technical, project, business, and professional—plays a role in influencing an architect and the architecture, as shown in Figure 3.4.

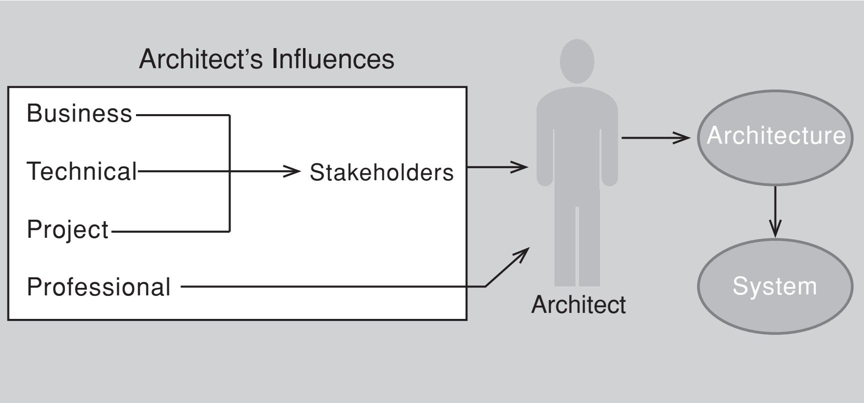


FIGURE 3.4 Influences on the architect

An architect designing a system for which the real-time deadlines are tight will make one set of design choices; the same architect, designing a similar system in which the deadlines can be easily satisfied, will make different choices. And the same architect, designing a non-real-time system, is likely to make quite different choices still. Even with the same requirements, hardware, support software, and human resources available, an architect designing a system today is likely to design a different system than might have been designed five years ago.

3.7 What Do Architectures Influence?

The story about contexts influencing architectures has a flip side. It turns out that architectures have an influence on the very factors that influence them. Specifically, the existence of an architecture affects the technical, project, business, and professional contexts that subsequently influence future architectures.

Here is how the cycle works:

- *Technical context.* The architecture can affect stakeholder requirements for the next system by giving the customer the opportunity to receive a system (based on the same architecture) in a more reliable, timely, and economical manner than if the subsequent system were to be built from scratch, and typically with fewer defects. A customer may in fact be willing to relax some of their requirements to gain these economies. Shrink-wrapped software has clearly affected people's requirements by providing solutions that are not tailored to any individual's precise needs but are instead inexpensive and (in the best of all possible worlds) of high quality. Software product lines have the same effect on customers who cannot be so flexible with their requirements.
- *Project context.* The architecture affects the structure of the developing organization. An architecture prescribes a structure for a system; as we will see, it particularly prescribes the units of software that must be implemented (or otherwise obtained) and integrated to form the system. These units are the basis for the development project's structure. Teams are formed for individual software units; and the development, test, and integration activities all revolve around the units. Likewise, schedules and budgets allocate resources in chunks corresponding to the units. If a company becomes adept at building families of similar systems, it will tend to invest in each team by nurturing each area of expertise. Teams become embedded in the organization's structure. This is feedback from the architecture to the developing organization. In any design undertaken by the organization at large, these groups have a strong voice in the system's decomposition, pressuring for the continued existence of the portions they control.

- *Business context.* The architecture can affect the business goals of the developing organization. A successful system built from an architecture can enable a company to establish a foothold in a particular market segment—think of the iPhone or Android app platforms as examples. The architecture can provide opportunities for the efficient production and deployment of similar systems, and the organization may adjust its goals to take advantage of its newfound expertise to plumb the market. This is feedback from the system to the developing organization and the systems it builds.
- *Professional context.* The process of system building will affect the architect's experience with subsequent systems by adding to the corporate experience base. A system that was successfully built around a particular technical approach will make the architect more inclined to build systems using the same approach in the future. On the other hand, architectures that fail are less likely to be chosen for future projects.

These and other feedback mechanisms form what we call the Architecture Influence Cycle, or AIC, illustrated in Figure 3.5, which depicts the influences of the culture and business of the development organization on the software architecture. That architecture is, in turn, a primary determinant of the properties of the developed system or systems. But the AIC is also based on a recognition that shrewd organizations can take advantage of the organizational and experiential effects of developing an architecture and can use those effects to position their business strategically for future projects.

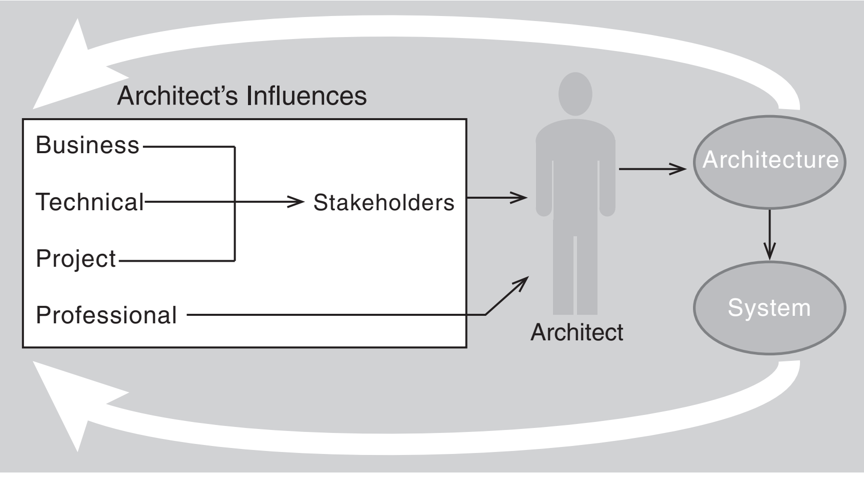


FIGURE 3.5 Architecture Influence Cycle

3.8 Summary

Architectures exist in four different contexts.

1. *Technical.* The technical context includes the achievement of quality attribute requirements. We spend Part II discussing how to do this. The technical context also includes the current technology. The cloud (discussed in Chapter 26) and mobile computing (discussed in Chapter 27) are important current technologies.
2. *Project life cycle.* Regardless of the software development methodology you use, you must make a business case for the system, understand the architecturally significant requirements, create or select the architecture, document and communicate the architecture, analyze or evaluate the architecture, implement and test the system based on the architecture, and ensure that the implementation conforms to the architecture.
3. *Business.* The system created from the architecture must satisfy the business goals of a wide variety of stakeholders, each of whom has different expectations for the system. The architecture is also influenced by and influences the structure of the development organization.
4. *Professional.* You must have certain skills and knowledge to be an architect, and there are certain duties that you must perform as an architect. These are influenced not only by coursework and reading but also by your experiences.

An architecture has some influences that lead to its creation, and its existence has an impact on the architect, the organization, and, potentially, the industry. We call this cycle the Architecture Influence Cycle.

3.9 For Further Reading

The product line framework produced by the Software Engineering Institute includes a discussion of business cases from which we drew [SEI 12].

The SEI has also published a case study of Celsius Tech that includes an example of how organizations and customers change over time [Brownsword 96].

Several other SEI reports discuss how to find business goals and the business goals that have been articulated by certain organizations [Kazman 05, Clements 10b].

Ruth Malan and Dana Bredemeyer provide a description of how an architect can build buy-in within an organization [Malan 00].

3.10 Discussion Questions

1. Enumerate six different software systems used by your organization. For each of these systems:

   a. What are the contextual influences?
   b. Who are the stakeholders?
   c. How do these systems reflect or impact the organizational structure?

2. What kinds of business goals have driven the construction of the following:

   a. The World Wide Web
   b. Amazon's EC2 cloud infrastructure
   c. Google's Android platform

3. What mechanisms are available to improve your skills and knowledge? What skills are you lacking?

4. Describe a system you are familiar with and place it into the AIC. Specifically, identify the forward and reverse influences on contextual factors.

PART TWO

QUALITY ATTRIBUTES

In Part II, we provide the technical foundations for you to design or analyze an architecture to achieve particular quality attributes. We do not discuss design or analysis processes here; we cover those topics in Part III. It is impossible, however, to understand how to improve the performance of a design, for example, without understanding something about performance.

In Chapter 4 we describe how to specify a quality attribute requirement and motivate design techniques called tactics to enable you to achieve a particular quality attribute requirement. We also enumerate seven categories of design decisions. These are categories of decisions that are universally important, and so we provide material to help an architect focus on these decisions. In Chapter 4, we describe these categories, and in each of the following chapters devoted to a particular quality attribute—Chapters 5–11—we use those categories to develop a checklist that tells you how to focus your attention on the important aspects associated with that quality attribute. Many of the items in our checklists may seem obvious, but the purpose of a checklist is to help ensure the completeness of your design and analysis process.

In addition to providing a treatment of seven specific quality attributes (availability, interoperability, modifiability, performance, security, testability, and usability), we also describe how you can generate the material provided in Chapters 5–11 for other quality attributes that we have not covered.

Architectural patterns provide known solutions to a number of common problems in design. In Chapter 13, we present some of the most important patterns and discuss the relationship between patterns and tactics.

Being able to analyze a design for a particular quality attribute is a key skill that you as an architect will need to acquire. In Chapter 14, we discuss modeling techniques for some of the quality attributes.

This page intentionally left blank

4

Understanding Quality Attributes

> *Between stimulus and response, there is a space. In that space is our power to choose our response. In our response lies our growth and our freedom.*
>
> — Viktor E. Frankl, *Man's Search for Meaning*

As we have seen in the Architecture Influence Cycle (in Chapter 3), many factors determine the qualities that must be provided for in a system's architecture. These qualities go beyond functionality, which is the basic statement of the system's capabilities, services, and behavior. Although functionality and other qualities are closely related, as you will see, functionality often takes the front seat in the development scheme. This preference is shortsighted, however. Systems are frequently redesigned not because they are functionally deficient—the replacements are often functionally identical—but because they are difficult to maintain, port, or scale; or they are too slow; or they have been compromised by hackers. In Chapter 2, we said that architecture was the first place in software creation in which quality requirements could be addressed. It is the mapping of a system's functionality onto software structures that determines the architecture's support for qualities. In Chapters 5–11 we discuss how various qualities are supported by architectural design decisions. In Chapter 17 we show how to integrate all of the quality attribute decisions into a single design.

We have been using the term "quality attribute" loosely, but now it is time to define it more carefully. A quality attribute (QA) is a measurable or testable property of a system that is used to indicate how well the system satisfies the needs of its stakeholders. You can think of a quality attribute as measuring the "goodness" of a product along some dimension of interest to a stakeholder.

In this chapter our focus is on understanding the following:

- How to express the qualities we want our architecture to provide to the system or systems we are building from it

- How to achieve those qualities
- How to determine the design decisions we might make with respect to those qualities

This chapter provides the context for the discussion of specific quality attributes in Chapters 5–11.

4.1 Architecture and Requirements

Requirements for a system come in a variety of forms: textual requirements, mockups, existing systems, use cases, user stories, and more. Chapter 16 discusses the concept of an *architecturally significant requirement*, the role such requirements play in architecture, and how to identify them. No matter the source, all requirements encompass the following categories:

1. *Functional requirements.* These requirements state what the system must do, and how it must behave or react to runtime stimuli.
2. *Quality attribute requirements.* These requirements are qualifications of the functional requirements or of the overall product. A qualification of a functional requirement is an item such as how fast the function must be performed, or how resilient it must be to erroneous input. A qualification of the overall product is an item such as the time to deploy the product or a limitation on operational costs.
3. *Constraints.* A constraint is a design decision with zero degrees of freedom. That is, it's a design decision that's already been made. Examples include the requirement to use a certain programming language or to reuse a certain existing module, or a management fiat to make your system service oriented. These choices are arguably in the purview of the architect, but external factors (such as not being able to train the staff in a new language, or having a business agreement with a software supplier, or pushing business goals of service interoperability) have led those in power to dictate these design outcomes.

What is the "response" of architecture to each of these kinds of requirements?

1. Functional requirements are satisfied by assigning an appropriate sequence of responsibilities throughout the design. As we will see later in this chapter, assigning responsibilities to architectural elements is a fundamental architectural design decision.
2. Quality attribute requirements are satisfied by the various structures designed into the architecture, and the behaviors and interactions of the elements that populate those structures. Chapter 17 will show this approach in more detail.

3. Constraints are satisfied by accepting the design decision and reconciling it with other affected design decisions.

4.2 Functionality

Functionality is the ability of the system to do the work for which it was intended. Of all of the requirements, functionality has the strangest relationship to architecture.

First of all, functionality does not determine architecture. That is, given a set of required functionality, there is no end to the architectures you could create to satisfy that functionality. At the very least, you could divide up the functionality in any number of ways and assign the subpieces to different architectural elements.

In fact, if functionality were the only thing that mattered, you wouldn't have to divide the system into pieces at all; a single monolithic blob with no internal structure would do just fine. Instead, we design our systems as structured sets of cooperating architectural elements—modules, layers, classes, services, databases, apps, threads, peers, tiers, and on and on—to make them understandable and to support a variety of other purposes. Those "other purposes" are the other quality attributes that we'll turn our attention to in the remaining sections of this chapter, and the remaining chapters of Part II.

But although functionality is independent of any particular structure, functionality is achieved by assigning responsibilities to architectural elements, resulting in one of the most basic of architectural structures.

Although responsibilities can be allocated arbitrarily to any modules, software architecture constrains this allocation when other quality attributes are important. For example, systems are frequently divided so that several people can cooperatively build them. The architect's interest in functionality is in how it interacts with and constrains other qualities.

4.3 Quality Attribute Considerations

Just as a system's functions do not stand on their own without due consideration of other quality attributes, neither do quality attributes stand on their own; they pertain to the functions of the system. If a functional requirement is "When the user presses the green button, the Options dialog appears," a performance QA annotation might describe how quickly the dialog will appear; an availability QA annotation might describe how often this function will fail, and how quickly it will be repaired; a usability QA annotation might describe how easy it is to learn this function.

Functional Requirements

After more than 15 years of writing and discussing the distinction between functional requirements and quality requirements, the definition of functional requirements still eludes me. Quality attribute requirements are well defined: performance has to do with the timing behavior of the system, modifiability has to do with the ability of the system to support changes in its behavior or other qualities after initial deployment, availability has to do with the ability of the system to survive failures, and so forth.

Function, however, is much more slippery. An international standard (ISO 25010) defines functional suitability as “the capability of the software product to provide functions which meet stated and implied needs when the software is used under specified conditions.” That is, functionality is the ability to provide functions. One interpretation of this definition is that functionality describes what the system does and quality describes how well the system does its function. That is, qualities are attributes of the system and function is the purpose of the system.

This distinction breaks down, however, when you consider the nature of some of the “function.” If the function of the software is to control engine behavior, how can the function be correctly implemented without considering timing behavior? Is the ability to control access through requiring a user name/password combination not a function even though it is not the purpose of any system?

I like much better the use of the word “responsibility” to describe computations that a system must perform. Questions such as “What are the timing constraints on that set of responsibilities?”, “What modifications are anticipated with respect to that set of responsibilities?”, and “What class of users is allowed to execute that set of responsibilities?” make sense and are actionable.

The achievement of qualities induces responsibility; think of the user name/password example just mentioned. Further, one can identify responsibilities as being associated with a particular set of requirements.

So does this mean that the term “functional requirement” shouldn’t be used? People have an understanding of the term, but when precision is desired, we should talk about sets of specific responsibilities instead.

Paul Clements has long ranted against the careless use of the term “nonfunctional,” and now it’s my turn to rant against the careless use of the term “functional”—probably equally ineffectually.

—LB

Quality attributes have been of interest to the software community at least since the 1970s. There are a variety of published taxonomies and definitions, and many of them have their own research and practitioner communities. From an architect's perspective, there are three problems with previous discussions of system quality attributes:

1. The definitions provided for an attribute are not testable. It is meaningless to say that a system will be "modifiable." Every system may be modifiable with respect to one set of changes and not modifiable with respect to another. The other quality attributes are similar in this regard: a system may be robust with respect to some faults and brittle with respect to others. And so forth.
2. Discussion often focuses on which quality a particular concern belongs to. Is a system failure due to a denial-of-service attack an aspect of availability, an aspect of performance, an aspect of security, or an aspect of usability? All four attribute communities would claim ownership of a system failure due to a denial-of-service attack. All are, to some extent, correct. But this doesn't help us, as architects, understand and create architectural solutions to manage the attributes of concern.
3. Each attribute community has developed its own vocabulary. The performance community has "events" arriving at a system, the security community has "attacks" arriving at a system, the availability community has "failures" of a system, and the usability community has "user input." All of these may actually refer to the same occurrence, but they are described using different terms.

A solution to the first two of these problems (untestable definitions and overlapping concerns) is to use *quality attribute scenarios* as a means of characterizing quality attributes (see the next section). A solution to the third problem is to provide a discussion of each attribute—concentrating on its underlying concerns—to illustrate the concepts that are fundamental to that attribute community.

There are two categories of quality attributes on which we focus. The first is those that describe some property of the system at runtime, such as availability, performance, or usability. The second is those that describe some property of the development of the system, such as modifiability or testability.

Within complex systems, quality attributes can never be achieved in isolation. The achievement of any one will have an effect, sometimes positive and sometimes negative, on the achievement of others. For example, almost every quality attribute negatively affects performance. Take portability. The main technique for achieving portable software is to isolate system dependencies, which introduces overhead into the system's execution, typically as process or procedure boundaries, and this hurts performance. Determining the design that satisfies all of the quality attribute requirements is partially a matter of making the appropriate tradeoffs; we discuss design in Chapter 17. Our purpose here is to provide the context for discussing each quality attribute. In particular, we focus on how quality attributes can be specified, what architectural decisions will enable the achievement of particular quality attributes, and what questions about quality attributes will enable the architect to make the correct design decisions.

4.4 Specifying Quality Attribute Requirements

A quality attribute requirement should be unambiguous and testable. We use a common form to specify all quality attribute requirements. This has the advantage of emphasizing the commonalities among all quality attributes. It has the disadvantage of occasionally being a force-fit for some aspects of quality attributes.

Our common form for quality attribute expression has these parts:

- *Stimulus.* We use the term "stimulus" to describe an event arriving at the system. The stimulus can be an event to the performance community, a user operation to the usability community, or an attack to the security community. We use the same term to describe a motivating action for developmental qualities. Thus, a stimulus for modifiability is a request for a modification; a stimulus for testability is the completion of a phase of development.
- *Stimulus source.* A stimulus must have a source—it must come from somewhere. The source of the stimulus may affect how it is treated by the system. A request from a trusted user will not undergo the same scrutiny as a request by an untrusted user.
- *Response.* How the system should respond to the stimulus must also be specified. The response consists of the responsibilities that the system (for runtime qualities) or the developers (for development-time qualities) should perform in response to the stimulus. For example, in a performance scenario, an event arrives (the stimulus) and the system should process that event and generate a response. In a modifiability scenario, a request for a modification arrives (the stimulus) and the developers should implement the modification—without side effects—and then test and deploy the modification.
- *Response measure.* Determining whether a response is satisfactory—whether the requirement is satisfied—is enabled by providing a response measure. For performance this could be a measure of latency or throughput; for modifiability it could be the labor or wall clock time required to make, test, and deploy the modification.

These four characteristics of a scenario are the heart of our quality attribute specifications. But there are two more characteristics that are important: environment and artifact.

- *Environment.* The environment of a requirement is the set of circumstances in which the scenario takes place. The environment acts as a qualifier on the stimulus. For example, a request for a modification that arrives after the code has been frozen for a release may be treated differently than one that arrives before the freeze. A failure that is the fifth successive failure of a component may be treated differently than the first failure of that component.

- *Artifact.* Finally, the artifact is the portion of the system to which the requirement applies. Frequently this is the entire system, but occasionally specific portions of the system may be called out. A failure in a data store may be treated differently than a failure in the metadata store. Modifications to the user interface may have faster response times than modifications to the middleware.

To summarize how we specify quality attribute requirements, we capture them formally as six-part scenarios. While it is common to omit one or more of these six parts, particularly in the early stages of thinking about quality attributes, knowing that all parts are there forces the architect to consider whether each part is relevant.

In summary, here are the six parts:

1. *Source of stimulus.* This is some entity (a human, a computer system, or any other actuator) that generated the stimulus.
2. *Stimulus.* The stimulus is a condition that requires a response when it arrives at a system.
3. *Environment.* The stimulus occurs under certain conditions. The system may be in an overload condition or in normal operation, or some other relevant state. For many systems, "normal" operation can refer to one of a number of modes. For these kinds of systems, the environment should specify in which mode the system is executing.
4. *Artifact.* Some artifact is stimulated. This may be a collection of systems, the whole system, or some piece or pieces of it.
5. *Response.* The response is the activity undertaken as the result of the arrival of the stimulus.
6. *Response measure.* When the response occurs, it should be measurable in some fashion so that the requirement can be tested.

We distinguish *general* quality attribute scenarios (which we call "general scenarios" for short)—those that are system independent and can, potentially, pertain to any system—from *concrete* quality attribute scenarios (concrete scenarios)—those that are specific to the particular system under consideration.

We can characterize quality attributes as a collection of general scenarios. Of course, to translate these generic attribute characterizations into requirements for a particular system, the general scenarios need to be made system specific. Detailed examples of these scenarios will be given in Chapters 5–11. Figure 4.1 shows the parts of a quality attribute scenario that we have just discussed. Figure 4.2 shows an example of a general scenario, in this case for availability.

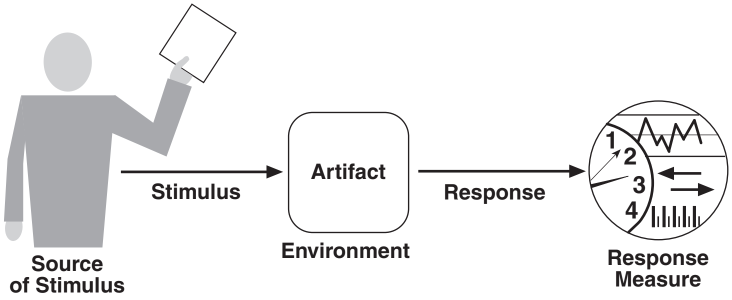


FIGURE 4.1 The parts of a quality attribute scenario

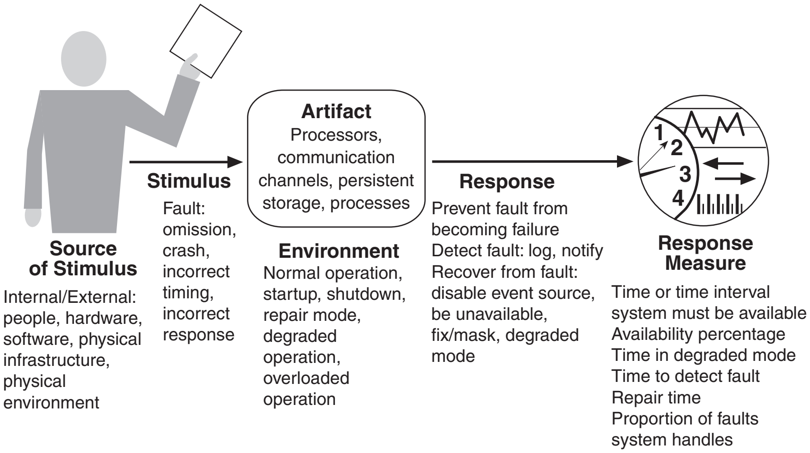

Source of Stimulus
Internal/External: people, hardware, software, physical infrastructure, physical environment
Stimulus
Fault: omission, crash, incorrect timing, incorrect response
Artifact
Processors, communication channels, persistent storage, processes
Environment
Normal operation, startup, shutdown, repair mode, degraded operation, overloaded operation
Response
Prevent fault from becoming failure
Detect fault: log, notify
Recover from fault: disable event source, be unavailable, fix/mask, degraded mode
1
2
3
4
Response Measure
Time or time interval system must be available
Availability percentage
Time in degraded mode
Time to detect fault
Repair time
Proportion of faults system handles

FIGURE 4.2 A general scenario for availability

4.5 Achieving Quality Attributes through Tactics

The quality attribute requirements specify the responses of the system that, with a bit of luck and a dose of good planning, realize the goals of the business. We now turn to the techniques an architect can use to *achieve* the required quality attributes. We call these techniques *architectural tactics.* A tactic is a design decision that influences the achievement of a quality attribute response—tactics directly affect the system's response to some stimulus. Tactics impart portability to one design, high performance to another, and integrability to a third.

Not My Problem

One time I was doing an architecture analysis on a complex system created by and for Lawrence Livermore National Laboratory. If you visit their website (www.llnl.gov) and try to figure out what Livermore Labs does, you will see the word "security" mentioned over and over. The lab focuses on nuclear security, international and domestic security, and environmental and energy security. Serious stuff . . .

Keeping this emphasis in mind, I asked them to describe the quality attributes of concern for the system that I was analyzing. I'm sure you can imagine my surprise when security wasn't mentioned once! The system stakeholders mentioned performance, modifiability, evolvability, interoperability, configurability, and portability, and one or two more, but the word security never passed their lips.

Being a good analyst, I questioned this seemingly shocking and obvious omission. Their answer was simple and, in retrospect, straightforward: "We don't care about it. Our systems are not connected to any external network and we have barbed-wire fences and guards with machine guns." Of course, *someone* at Livermore Labs was very interested in security. But it was clearly not the software architects.

—RK

The focus of a tactic is on a single quality attribute response. Within a tactic, there is no consideration of tradeoffs. Tradeoffs must be explicitly considered and controlled by the designer. In this respect, tactics differ from architectural patterns, where tradeoffs are built into the pattern. (We visit the relation between tactics and patterns in Chapter 14. Chapter 13 explains how sets of tactics for a quality attribute can be constructed, which are the steps we used to produce the set in this book.)

A system design consists of a collection of decisions. Some of these decisions help control the quality attribute responses; others ensure achievement of system functionality. We represent the relationship between stimulus, tactics, and response in Figure 4.3. The tactics, like design patterns, are design techniques that architects have been using for years. Our contribution is to isolate, catalog, and describe them. We are not inventing tactics here, we are just capturing what architects do in practice.

Why do we do this? There are three reasons:

1. Design patterns are complex; they typically consist of a bundle of design decisions. But patterns are often difficult to apply as is; architects need to modify and adapt them. By understanding the role of tactics, an architect can more easily assess the options for augmenting an existing pattern to achieve a quality attribute goal.

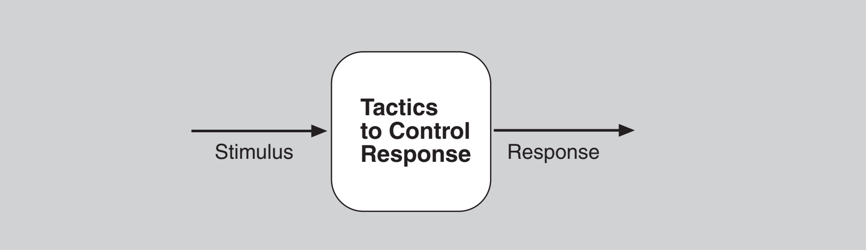


FIGURE 4.3 Tactics are intended to control responses to stimuli.

2. If no pattern exists to realize the architect's design goal, tactics allow the architect to construct a design fragment from "first principles." Tactics give the architect insight into the properties of the resulting design fragment.
3. By cataloging tactics, we provide a way of making design more systematic within some limitations. Our list of tactics does not provide a taxonomy. We only provide a categorization. The tactics will overlap, and you frequently will have a choice among multiple tactics to improve a particular quality attribute. The choice of which tactic to use depends on factors such as tradeoffs among other quality attributes and the cost to implement. These considerations transcend the discussion of tactics for particular quality attributes. Chapter 17 provides some techniques for choosing among competing tactics.

The tactics that we present can and should be refined. Consider performance: *Schedule resources* is a common performance tactic. But this tactic needs to be refined into a specific scheduling strategy, such as shortest-job-first, round-robin, and so forth, for specific purposes. *Use an intermediary* is a modifiability tactic. But there are multiple types of intermediaries (layers, brokers, and proxies, to name just a few). Thus there are refinements that a designer will employ to make each tactic concrete.

In addition, the application of a tactic depends on the context. Again considering performance: *Manage sampling rate* is relevant in some real-time systems but not in all real-time systems and certainly not in database systems.

4.6 Guiding Quality Design Decisions

Recall that one can view an architecture as the result of applying a collection of design decisions. What we present here is a systematic categorization of these decisions so that an architect can focus attention on those design dimensions likely to be most troublesome.

The seven categories of design decisions are

1. Allocation of responsibilities
2. Coordination model
3. Data model
4. Management of resources
5. Mapping among architectural elements
6. Binding time decisions
7. Choice of technology

These categories are not the only way to classify architectural design decisions, but they do provide a rational division of concerns. These categories might overlap, but it's all right if a particular decision exists in two different categories, because the concern of the architect is to ensure that every important decision is considered. Our categorization of decisions is partially based on our definition of software architecture in that many of our categories relate to the definition of structures and the relations among them.

Allocation of Responsibilities

Decisions involving allocation of responsibilities include the following:

- Identifying the important responsibilities, including basic system functions, architectural infrastructure, and satisfaction of quality attributes.
- Determining how these responsibilities are allocated to non-runtime and runtime elements (namely, modules, components, and connectors).

Strategies for making these decisions include functional decomposition, modeling real-world objects, grouping based on the major modes of system operation, or grouping based on similar quality requirements: processing frame rate, security level, or expected changes.

In Chapters 5–11, where we apply these design decision categories to a number of important quality attributes, the checklists we provide for the allocation of responsibilities category is derived systematically from understanding the stimuli and responses listed in the general scenario for that QA.

Coordination Model

Software works by having elements interact with each other through designed mechanisms. These mechanisms are collectively referred to as a coordination model. Decisions about the coordination model include these:

- Identifying the elements of the system that must coordinate, or are prohibited from coordinating.
- Determining the properties of the coordination, such as timeliness, currency, completeness, correctness, and consistency.
- Choosing the communication mechanisms (between systems, between our system and external entities, between elements of our system) that realize those properties. Important properties of the communication mechanisms include stateful versus stateless, synchronous versus asynchronous, guaranteed versus nonguaranteed delivery, and performance-related properties such as throughput and latency.

Data Model

Every system must represent artifacts of system-wide interest—data—in some internal fashion. The collection of those representations and how to interpret them is referred to as the data model. Decisions about the data model include the following:

- Choosing the major data abstractions, their operations, and their properties. This includes determining how the data items are created, initialized, accessed, persisted, manipulated, translated, and destroyed.
- Compiling metadata needed for consistent interpretation of the data.
- Organizing the data. This includes determining whether the data is going to be kept in a relational database, a collection of objects, or both. If both, then the mapping between the two different locations of the data must be determined.

Management of Resources

An architect may need to arbitrate the use of shared resources in the architecture. These include hard resources (e.g., CPU, memory, battery, hardware buffers, system clock, I/O ports) and soft resources (e.g., system locks, software buffers, thread pools, and non-thread-safe code).

Decisions for management of resources include the following:

- Identifying the resources that must be managed and determining the limits for each.
- Determining which system element(s) manage each resource.
- Determining how resources are shared and the arbitration strategies employed when there is contention.
- Determining the impact of saturation on different resources. For example, as a CPU becomes more heavily loaded, performance usually just degrades fairly steadily. On the other hand, when you start to run out of memory, at

some point you start paging/swapping intensively and your performance suddenly crashes to a halt.

Mapping among Architectural Elements

An architecture must provide two types of mappings. First, there is mapping between elements in different types of architecture structures—for example, mapping from units of development (modules) to units of execution (threads or processes). Next, there is mapping between software elements and environment elements—for example, mapping from processes to the specific CPUs where these processes will execute.

Useful mappings include these:

- The mapping of modules and runtime elements to each other—that is, the runtime elements that are created from each module; the modules that contain the code for each runtime element.
- The assignment of runtime elements to processors.
- The assignment of items in the data model to data stores.
- The mapping of modules and runtime elements to units of delivery.

Binding Time Decisions

Binding time decisions introduce allowable ranges of variation. This variation can be bound at different times in the software life cycle by different entities—from design time by a developer to runtime by an end user. A binding time decision establishes the scope, the point in the life cycle, and the mechanism for achieving the variation.

The decisions in the other six categories have an associated binding time decision. Examples of such binding time decisions include the following:

- For allocation of responsibilities, you can have build-time selection of modules via a parameterized makefile.
- For choice of coordination model, you can design runtime negotiation of protocols.
- For resource management, you can design a system to accept new peripheral devices plugged in at runtime, after which the system recognizes them and downloads and installs the right drivers automatically.
- For choice of technology, you can build an app store for a smartphone that automatically downloads the version of the app appropriate for the phone of the customer buying the app.

When making binding time decisions, you should consider the costs to implement the decision and the costs to make a modification after you have implemented the decision. For example, if you are considering changing platforms at some time after code time, you can insulate yourself from the effects caused by porting your system to another platform at some cost. Making this decision depends on the costs incurred by having to modify an early binding compared to the costs incurred by implementing the mechanisms involved in the late binding.

Choice of Technology

Every architecture decision must eventually be realized using a specific technology. Sometimes the technology selection is made by others, before the intentional architecture design process begins. In this case, the chosen technology becomes a constraint on decisions in each of our seven categories. In other cases, the architect must choose a suitable technology to realize a decision in every one of the categories.

Choice of technology decisions involve the following:

- Deciding which technologies are available to realize the decisions made in the other categories.
- Determining whether the available tools to support this technology choice (IDEs, simulators, testing tools, etc.) are adequate for development to proceed.
- Determining the extent of internal familiarity as well as the degree of external support available for the technology (such as courses, tutorials, examples, and availability of contractors who can provide expertise in a crunch) and deciding whether this is adequate to proceed.
- Determining the side effects of choosing a technology, such as a required coordination model or constrained resource management opportunities.
- Determining whether a new technology is compatible with the existing technology stack. For example, can the new technology run on top of or alongside the existing technology stack? Can it communicate with the existing technology stack? Can the new technology be monitored and managed?

4.7 Summary

Requirements for a system come in three categories:

1. *Functional.* These requirements are satisfied by including an appropriate set of responsibilities within the design.
2. *Quality attribute.* These requirements are satisfied by the structures and behaviors of the architecture.
3. *Constraints.* A constraint is a design decision that’s already been made.

To express a quality attribute requirement, we use a quality attribute scenario. The parts of the scenario are these:

1. Source of stimulus
2. Stimulus
3. Environment
4. Artifact
5. Response
6. Response measure

An architectural tactic is a design decision that affects a quality attribute response. The focus of a tactic is on a single quality attribute response. Architectural patterns can be seen as "packages" of tactics.

The seven categories of architectural design decisions are these:

1. Allocation of responsibilities
2. Coordination model
3. Data model
4. Management of resources
5. Mapping among architectural elements
6. Binding time decisions
7. Choice of technology

4.8 For Further Reading

Philippe Kruchten [Kruchten 04] provides another categorization of design decisions.

Pena [Pena 87] uses categories of Function/Form/Economy/Time as a way of categorizing design decisions.

Binding time and mechanisms to achieve different types of binding times are discussed in [Bachmann 05].

Taxonomies of quality attributes can be found in [Boehm 78], [McCall 77], and [ISO 11].

Arguments for viewing architecture as essentially independent from function can be found in [Shaw 95].

4.9 Discussion Questions

1. What is the relationship between a use case and a quality attribute scenario? If you wanted to add quality attribute information to a use case, how would you do it?

2. Do you suppose that the set of tactics for a quality attribute is finite or infinite? Why?

3. Discuss the choice of programming language (an example of choice of technology) and its relation to architecture in general, and the design decisions in the other six categories? For instance, how can certain programming languages enable or inhibit the choice of particular coordination models?

4. We will be using the automatic teller machine as an example throughout the chapters on quality attributes. Enumerate the set of responsibilities that an automatic teller machine should support and propose an initial design to accommodate that set of responsibilities. Justify your proposal.

5. Think about the screens that your favorite automatic teller machine uses. What do those screens tell you about binding time decisions reflected in the architecture?

6. Consider the choice between synchronous and asynchronous communication (a choice in the coordination mechanism category). What quality attribute requirements might lead you to choose one over the other?

7. Consider the choice between stateful and stateless communication (a choice in the coordination mechanism category). What quality attribute requirements might lead you to choose one over the other?

8. Most peer-to-peer architecture employs late binding of the topology. What quality attributes does this promote or inhibit?

5

Availability

With James Scott

Ninety percent of life is just showing up.
—Woody Allen

Availability refers to a property of software that it is there and ready to carry out its task when you need it to be. This is a broad perspective and encompasses what is normally called reliability (although it may encompass additional considerations such as downtime due to periodic maintenance). In fact, availability builds upon the concept of reliability by adding the notion of recovery—that is, when the system breaks, it repairs itself. Repair may be accomplished by various means, which we'll see in this chapter. More precisely, Avižienis and his colleagues have defined dependability:

> Dependability is the ability to avoid failures that are more frequent and more severe than is acceptable.

Our definition of availability as an aspect of dependability is this: "Availability refers to the ability of a system to mask or repair faults such that the cumulative service outage period does not exceed a required value over a specified time interval." These definitions make the concept of failure subject to the judgment of an external agent, possibly a human. They also subsume concepts of reliability, confidentiality, integrity, and any other quality attribute that involves a concept of unacceptable failure.

Availability is closely related to security. A denial-of-service attack is explicitly designed to make a system fail—that is, to make it unavailable. Availability is also closely related to performance, because it may be difficult to tell when a system has failed and when it is simply being outrageously slow to respond. Finally, availability is closely allied with safety, which is concerned with keeping the system from entering a hazardous state and recovering or limiting the damage when it does.

Fundamentally, availability is about minimizing service outage time by mitigating faults. Failure implies visibility to a system or human observer in the environment. That is, a failure is the deviation of the system from its specification, where the deviation is externally visible. One of the most demanding tasks in building a high-availability, fault-tolerant system is to understand the nature of the failures that can arise during operation (see the sidebar "Planning for Failure"). Once those are understood, mitigation strategies can be designed into the software.

A failure's cause is called a fault. A fault can be either internal or external to the system under consideration. Intermediate states between the occurrence of a fault and the occurrence of a failure are called errors. Faults can be prevented, tolerated, removed, or forecast. In this way a system becomes "resilient" to faults.

Among the areas with which we are concerned are how system faults are detected, how frequently system faults may occur, what happens when a fault occurs, how long a system is allowed to be out of operation, when faults or failures may occur safely, how faults or failures can be prevented, and what kinds of notifications are required when a failure occurs.

Because a system failure is observable by users, the time to repair is the time until the failure is no longer observable. This may be a brief delay in the response time or it may be the time it takes someone to fly to a remote location in the Andes to repair a piece of mining machinery (as was recounted to us by a person responsible for repairing the software in a mining machine engine). The notion of "observability" can be a tricky one: the Stuxnet virus, as an example, went unobserved for a very long time even though it was doing damage. In addition, we are often concerned with the level of capability that remains when a failure has occurred—a degraded operating mode.

The distinction between faults and failures allows discussion of automatic repair strategies. That is, if code containing a fault is executed but the system is able to recover from the fault without any deviation from specified behavior being observable, there is no failure.

The availability of a system can be calculated as the probability that it will provide the specified services within required bounds over a specified time interval. When referring to hardware, there is a well-known expression used to derive steady-state availability:

$$\frac{MTBF}{(MTBF + MTTR)}$$

where *MTBF* refers to the mean time between failures and *MTTR* refers to the mean time to repair. In the software world, this formula should be interpreted to mean that when thinking about availability, you should think about what will make your system fail, how likely that is to occur, and that there will be some time required to repair it.

From this formula it is possible to calculate probabilities and make claims like "99.999 percent availability," or a 0.001 percent probability that the system will not be operational when needed. Scheduled downtimes (when the system is intentionally taken out of service) may not be considered when calculating availability, because the system is deemed "not needed" then; of course, this depends on the specific requirements for the system, often encoded in service-level agreements (SLAs). This arrangement may lead to seemingly odd situations where the system is down and users are waiting for it, but the downtime is scheduled and so is not counted against any availability requirements.

In operational systems, faults are detected and correlated prior to being reported and repaired. Fault correlation logic will categorize a fault according to its severity (critical, major, or minor) and service impact (service-affecting or non-service-affecting) in order to provide the system operator with timely and accurate system status and allow for the appropriate repair strategy to be employed. The repair strategy may be automated or may require manual intervention.

The availability provided by a computer system or hosting service is frequently expressed as a service-level agreement. This SLA specifies the availability level that is guaranteed and, usually, the penalties that the computer system or hosting service will suffer if the SLA is violated. The SLA that Amazon provides for its EC2 cloud service is

> AWS will use commercially reasonable efforts to make Amazon EC2 available with an Annual Uptime Percentage [defined elsewhere] of at least 99.95% during the Service Year. In the event Amazon EC2 does not meet the Annual Uptime Percentage commitment, you will be eligible to receive a Service Credit as described below.

Table 5.1 provides examples of system availability requirements and associated threshold values for acceptable system downtime, measured over observation periods of 90 days and one year. The term *high availability* typically refers to designs targeting availability of 99.999 percent ("5 nines") or greater. By definition or convention, only unscheduled outages contribute to system downtime.

TABLE 5.1 System Availability Requirements

Availability	Downtime/90 Days	Downtime/Year
99.0%	21 hours, 36 minutes	3 days, 15.6 hours
99.9%	2 hours, 10 minutes	8 hours, 0 minutes, 46 seconds
99.99%	12 minutes, 58 seconds	52 minutes, 34 seconds
99.999%	1 minute, 18 seconds	5 minutes, 15 seconds
99.9999%	8 seconds	32 seconds

Planning for Failure

When designing a high-availability or safety-critical system, it's tempting to say that failure is not an option. It's a catchy phrase, but it's a lousy design philosophy. In fact, failure is not only an option, it's almost inevitable. What will make your system safe and available is planning for the occurrence of failure or (more likely) failures, and handling them with aplomb. The first step is to understand what kinds of failures your system is prone to, and what the consequences of each will be. Here are three well-known techniques for getting a handle on this.

Hazard analysis

Hazard analysis is a technique that attempts to catalog the hazards that can occur during the operation of a system. It categorizes each hazard according to its severity. For example, the DO-178B standard used in the aeronautics industry defines these failure condition levels in terms of their effects on the aircraft, crew, and passengers:

- *Catastrophic.* This kind of failure may cause a crash. This failure represents the loss of critical function required to safely fly and land aircraft.
- *Hazardous.* This kind of failure has a large negative impact on safety or performance, or reduces the ability of the crew to operate the aircraft due to physical distress or a higher workload, or causes serious or fatal injuries among the passengers.
- *Major.* This kind of failure is significant, but has a lesser impact than a Hazardous failure (for example, leads to passenger discomfort rather than injuries) or significantly increases crew workload to the point where safety is affected.
- *Minor.* This kind of failure is noticeable, but has a lesser impact than a Major failure (for example, causing passenger inconvenience or a routine flight plan change).
- *No effect.* This kind of failure has no impact on safety, aircraft operation, or crew workload.

Other domains have their own categories and definitions. Hazard analysis also assesses the probability of each hazard occurring. Hazards for which the product of cost and probability exceed some threshold are then made the subject of mitigation activities.

Fault tree analysis

Fault tree analysis is an analytical technique that specifies a state of the system that negatively impacts safety or reliability, and then analyzes the system's context and operation to find all the ways that the undesired state could occur. The technique uses a graphic construct (the fault tree) that helps identify all sequential and parallel sequences of contributing faults that will result in the occurrence of the undesired state, which is listed at the top of the tree (the "top event"). The contributing faults might be hardware failures, human errors, software errors, or any other pertinent events that can lead to the undesired state.

Figure 5.1, taken from a NASA handbook on fault tree analysis, shows a very simple fault tree for which the top event is failure of component D. It shows that component D can fail if A fails *and* either B or C fails.

The symbols that connect the events in a fault tree are called gate symbols, and are taken from Boolean logic diagrams. Figure 5.2 illustrates the notation.

A fault tree lends itself to static analysis in various ways. For example, a "minimal cut set" is the smallest combination of events along the bottom of the tree that together can cause the top event. The set of minimal cut sets shows all the ways the bottom events can combine to cause the overarching failure. Any singleton minimal cut set reveals a single point of failure, which should be carefully scrutinized. Also, the probabilities of various contributing failures can be combined to come up with a probability of the top event occurring. Dynamic analysis occurs when the order of contributing failures matters. In this case, techniques such as Markov analysis can be used to calculate probability of failure over different failure sequences.

Fault trees aid in system design, but they can also be used to diagnose failures at runtime. If the top event has occurred, then (assuming the fault tree model is complete) one or more of the contributing failures has occurred, and the fault tree can be used to track it down and initiate repairs.

Failure Mode, Effects, and Criticality Analysis (FMECA) catalogs the kinds of failures that systems of a given type are prone to, along with how severe the effects of each one can be. FMECA relies on the history of

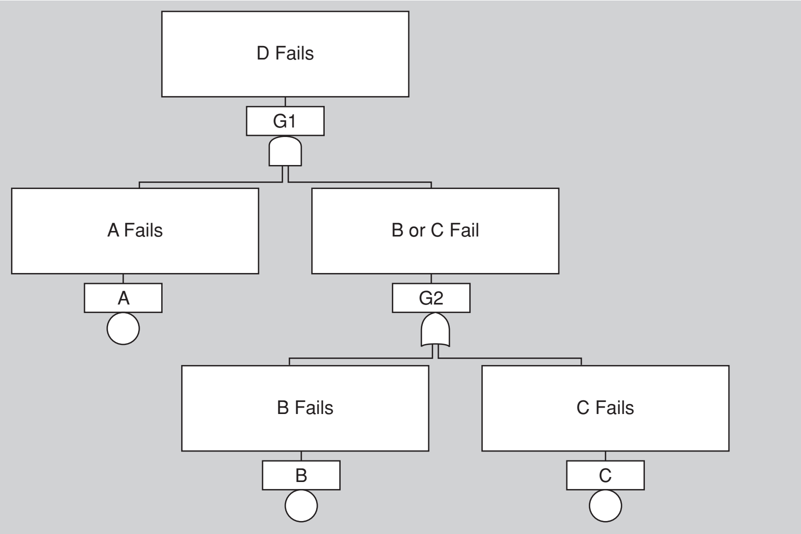
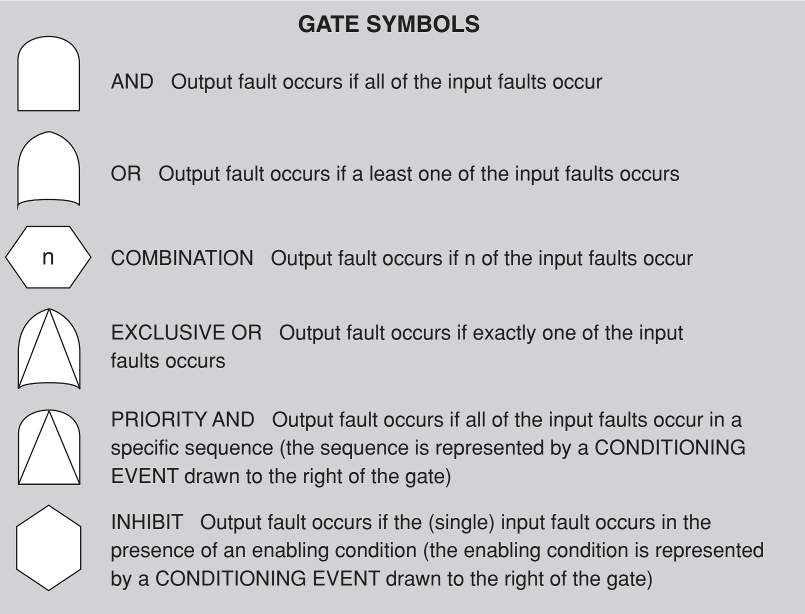


FIGURE 5.1 A simple fault tree. D fails if A fails and either B or C fails.

failure of similar systems in the past. Table 5.2, also taken from the NASA handbook, shows the data for a system of redundant amplifiers. Historical data shows that amplifiers fail most often when there is a short circuit or the circuit is left open, but there are several other failure modes as well (lumped together as “Other”).



AND Output fault occurs if all of the input faults occur

OR Output fault occurs if a least one of the input faults occurs

COMBINATION Output fault occurs if n of the input faults occur

EXCLUSIVE OR Output fault occurs if exactly one of the input faults occurs

PRIORITY AND Output fault occurs if all of the input faults occur in a specific sequence (the sequence is represented by a CONDITIONING EVENT drawn to the right of the gate)

INHIBIT Output fault occurs if the (single) input fault occurs in the presence of an enabling condition (the enabling condition is represented by a CONDITIONING EVENT drawn to the right of the gate)

FIGURE 5.2 Fault tree gate symbols

TABLE 5.2 Failure Probabilities and Effects

				Effects	
Component	**Failure Probability**	**Failure Mode**	**% Failures by Mode**	**Critical**	**Noncritical**
A	1×10^{-3}	Open	90		X
		Short	5	X (5×10^{-5})	
		Other	5	X (5×10^{-5})	
B	1×10^{-3}	Open	90		X
		Short	5	X (5×10^{-5})	
		Other	5	X (5×10^{-5})	

Adding up the critical column gives us the probability of a critical system failure: $5 \times 10^{-5} + 5 \times 10^{-5} + 5 \times 10^{-5} + 5 \times 10^{-5} = 2 \times 10^{-4}$.

These techniques, and others, are only as good as the knowledge and experience of the people who populate their respective data structures. One of the worst mistakes you can make, according to the NASA handbook, is to let form take priority over substance. That is, don't let safety engineering become a matter of just filling out the tables. Instead, keep pressing to find out what else can go wrong, and then plan for it.

5.1 Availability General Scenario

From these considerations we can now describe the individual portions of an availability general scenario. These are summarized in Table 5.3:

- *Source of stimulus*. We differentiate between internal and external origins of faults or failure because the desired system response may be different.
- *Stimulus*. A fault of one of the following classes occurs:
  - *Omission*. A component fails to respond to an input.
  - *Crash*. The component repeatedly suffers omission faults.
  - *Timing*. A component responds but the response is early or late.
  - *Response*. A component responds with an incorrect value.
- *Artifact.* This specifies the resource that is required to be highly available, such as a processor, communication channel, process, or storage.
- *Environment*. The state of the system when the fault or failure occurs may also affect the desired system response. For example, if the system has already seen some faults and is operating in other than normal mode, it may be desirable to shut it down totally. However, if this is the first fault observed, some degradation of response time or function may be preferred.
- *Response*. There are a number of possible reactions to a system fault. First, the fault must be detected and isolated (correlated) before any other response is possible. (One exception to this is when the fault is prevented before it occurs.) After the fault is detected, the system must recover from it. Actions associated with these possibilities include logging the failure, notifying selected users or other systems, taking actions to limit the damage caused by the fault, switching to a degraded mode with either less capacity or less function, shutting down external systems, or becoming unavailable during repair.
- *Response measure*. The response measure can specify an availability percentage, or it can specify a time to detect the fault, time to repair the fault, times or time intervals during which the system must be available, or the duration for which the system must be available.

Figure 5.3 shows a concrete scenario generated from the general scenario: The heartbeat monitor determines that the server is nonresponsive during normal operations. The system informs the operator and continues to operate with no downtime.

TABLE 5.3 Availability General Scenario

Portion of Scenario	Possible Values
Source	Internal/external: people, hardware, software, physical infrastructure, physical environment
Stimulus	Fault: omission, crash, incorrect timing, incorrect response
Artifact	Processors, communication channels, persistent storage, processes
Environment	Normal operation, startup, shutdown, repair mode, degraded operation, overloaded operation
Response	Prevent the fault from becoming a failure Detect the fault: ▪ Log the fault ▪ Notify appropriate entities (people or systems) Recover from the fault: ▪ Disable source of events causing the fault ▪ Be temporarily unavailable while repair is being effected ▪ Fix or mask the fault/failure or contain the damage it causes ▪ Operate in a degraded mode while repair is being effected
Response Measure	Time or time interval when the system must be available Availability percentage (e.g., 99.999%) Time to detect the fault Time to repair the fault Time or time interval in which system can be in degraded mode Proportion (e.g., 99%) or rate (e.g., up to 100 per second) of a certain class of faults that the system prevents, or handles without failing

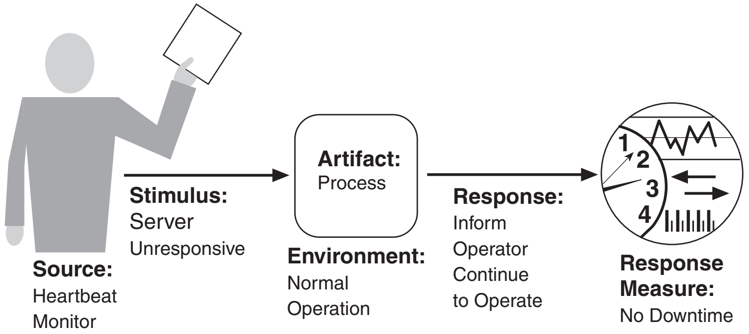


FIGURE 5.3 Sample concrete availability scenario

5.2 Tactics for Availability

A failure occurs when the system no longer delivers a service that is consistent with its specification; this failure is observable by the system's actors. A fault (or combination of faults) has the potential to cause a failure. Availability tactics, therefore, are designed to enable a system to endure system faults so that a service being delivered by the system remains compliant with its specification. The tactics we discuss in this section will keep faults from becoming failures or at least bound the effects of the fault and make repair possible. We illustrate this approach in Figure 5.4.

Availability tactics may be categorized as addressing one of three categories: fault detection, fault recovery, and fault prevention. The tactics categorization for availability is shown in Figure 5.5 (on the next page). Note that it is often the case that these tactics will be provided for you by a software infrastructure, such as a middleware package, so your job as an architect is often one of choosing and assessing (rather than implementing) the right availability tactics and the right combination of tactics.

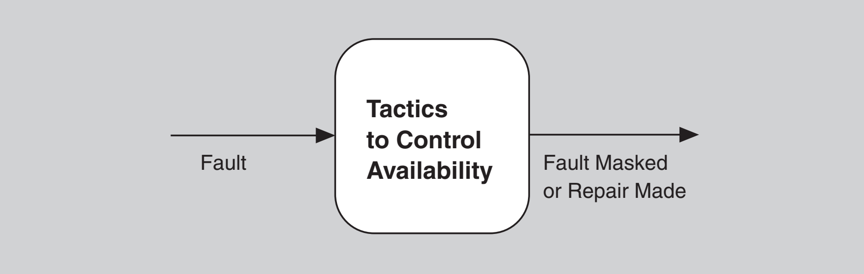


FIGURE 5.4 Goal of availability tactics

Detect Faults

Before any system can take action regarding a fault, the presence of the fault must be detected or anticipated. Tactics in this category include the following:

- *Ping/echo* refers to an asynchronous request/response message pair exchanged between nodes, used to determine reachability and the round-trip delay through the associated network path. But the echo also determines that the pinged component is alive and responding correctly. The ping is often sent by a system monitor. Ping/echo requires a time threshold to be set; this threshold tells the pinging component how long to wait for the echo before considering the pinged component to have failed ("timed out"). Standard implementations of ping/echo are available for nodes interconnected via IP.

- *Monitor*. A monitor is a component that is used to monitor the state of health of various other parts of the system: processors, processes, I/O, memory, and so on. A system monitor can detect failure or congestion in the network or other shared resources, such as from a denial-of-service attack. It orchestrates software using other tactics in this category to detect

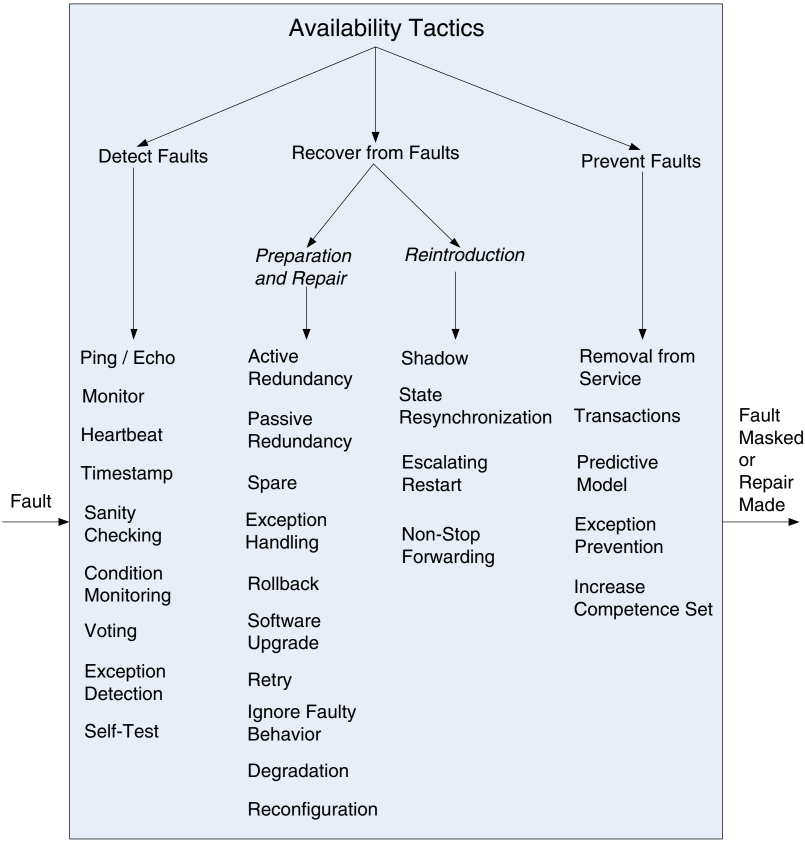


FIGURE 5.5 Availability tactics

malfunctioning components. For example, the system monitor can initiate self-tests, or be the component that detects faulty time stamps or missed heartbeats.[1]

- *Heartbeat* is a fault detection mechanism that employs a periodic message exchange between a system monitor and a process being monitored. A special case of heartbeat is when the process being monitored periodically resets the watchdog timer in its monitor to prevent it from expiring and thus signaling a fault. For systems where scalability is a concern, transport and processing overhead can be reduced by piggybacking heartbeat messages on to other control messages being exchanged between the process being monitored and the distributed system controller. The big difference between heartbeat and ping/echo is who holds the responsibility for initiating the health check—the monitor or the component itself.
- *Time stamp.* This tactic is used to detect incorrect sequences of events, primarily in distributed message-passing systems. A time stamp of an event can be established by assigning the state of a local clock to the event immediately after the event occurs. Simple sequence numbers can also be used for this purpose, if time information is not important.
- *Sanity checking* checks the validity or reasonableness of specific operations or outputs of a component. This tactic is typically based on a knowledge of the internal design, the state of the system, or the nature of the information under scrutiny. It is most often employed at interfaces, to examine a specific information flow.
- *Condition monitoring* involves checking conditions in a process or device, or validating assumptions made during the design. By monitoring conditions, this tactic prevents a system from producing faulty behavior. The computation of checksums is a common example of this tactic. However, the monitor must itself be simple (and, ideally, provable) to ensure that it does not introduce new software errors.
- *Voting.* The most common realization of this tactic is referred to as triple modular redundancy (TMR), which employs three components that do the same thing, each of which receives identical inputs, and forwards their output to voting logic, used to detect any inconsistency among the three output states. Faced with an inconsistency, the voter reports a fault. It must also decide what output to use. It can let the majority rule, or choose some computed average of the disparate outputs. This tactic depends critically on the voting logic, which is usually realized as a simple, rigorously reviewed and tested singleton so that the probability of error is low.

[1] When the detection mechanism is implemented using a counter or timer that is periodically reset, this specialization of system monitor is referred to as a "watchdog." During nominal operation, the process being monitored will periodically reset the watchdog counter/timer as part of its signal that it's working correctly; this is sometimes referred to as "petting the watchdog."

- *Replication* is the simplest form of voting; here, the components are exact clones of each other. Having multiple copies of identical components can be effective in protecting against random failures of hardware, but this cannot protect against design or implementation errors, in hardware or software, because there is no form of diversity embedded in this tactic.
- *Functional redundancy* is a form of voting intended to address the issue of common-mode failures (design or implementation faults) in hardware or software components. Here, the components must always give the same output given the same input, but they are diversely designed and diversely implemented.
- *Analytic redundancy* permits not only diversity among components' private sides, but also diversity among the components' inputs and outputs. This tactic is intended to tolerate specification errors by using separate requirement specifications. In embedded systems, analytic redundancy also helps when some input sources are likely to be unavailable at times. For example, avionics programs have multiple ways to compute aircraft altitude, such as using barometric pressure, the radar altimeter, and geometrically using the straight-line distance and look-down angle of a point ahead on the ground. The voter mechanism used with analytic redundancy needs to be more sophisticated than just letting majority rule or computing a simple average. It may have to understand which sensors are currently reliable or not, and it may be asked to produce a higher-fidelity value than any individual component can, by blending and smoothing individual values over time.

- *Exception detection* refers to the detection of a system condition that alters the normal flow of execution. The exception detection tactic can be further refined:

  - *System exceptions* will vary according to the processor hardware architecture employed and include faults such as divide by zero, bus and address faults, illegal program instructions, and so forth.
  - The *parameter fence* tactic incorporates an *a priori* data pattern (such as 0xDEADBEEF) placed immediately after any variable-length parameters of an object. This allows for runtime detection of overwriting the memory allocated for the object's variable-length parameters.
  - *Parameter typing* employs a base class that defines functions that add, find, and iterate over type-length-value (TLV) formatted message parameters. Derived classes use the base class functions to implement functions that provide parameter typing according to each parameter's structure. Use of strong typing to build and parse messages results in higher availability than implementations that simply treat messages as byte buckets. Of course, all design involves tradeoffs. When you employ strong typing, you typically trade higher availability against ease of evolution.

  - *Timeout* is a tactic that raises an exception when a component detects that it or another component has failed to meet its timing constraints. For example, a component awaiting a response from another component can raise an exception if the wait time exceeds a certain value.

- *Self-test.* Components (or, more likely, whole subsystems) can run procedures to test themselves for correct operation. Self-test procedures can be initiated by the component itself, or invoked from time to time by a system monitor. These may involve employing some of the techniques found in condition monitoring, such as checksums.

Recover from Faults

Recover-from-faults tactics are refined into *preparation-and-repair* tactics and *reintroduction* tactics. The latter are concerned with reintroducing a failed (but rehabilitated) component back into normal operation.

Preparation-and-repair tactics are based on a variety of combinations of retrying a computation or introducing redundancy. They include the following:

- *Active redundancy (hot spare).* This refers to a configuration where all of the nodes (active or redundant spare) in a protection group[2] receive and process identical inputs in parallel, allowing the redundant spare(s) to maintain synchronous state with the active node(s). Because the redundant spare possesses an identical state to the active processor, it can take over from a failed component in a matter of milliseconds. The simple case of one active node and one redundant spare node is commonly referred to as 1+1 (“one plus one”) redundancy. Active redundancy can also be used for facilities protection, where active and standby network links are used to ensure highly available network connectivity.
- *Passive redundancy (warm spare).* This refers to a configuration where only the active members of the protection group process input traffic; one of their duties is to provide the redundant spare(s) with periodic state updates. Because the state maintained by the redundant spares is only loosely coupled with that of the active node(s) in the protection group (with the looseness of the coupling being a function of the checkpointing mechanism employed between active and redundant nodes), the redundant nodes are referred to as warm spares. Depending on a system’s availability requirements, passive redundancy provides a solution that achieves a balance between the more highly available but more compute-intensive (and expensive) active redundancy tactic and the less available but significantly less complex cold spare tactic (which is also significantly cheaper). (For an

[2] A protection group is a group of processing nodes where one or more nodes are “active,” with the remaining nodes in the protection group serving as redundant spares.

example of implementing passive redundancy, see the section on code templates in Chapter 19.)

- *Spare (cold spare).* Cold sparing refers to a configuration where the redundant spares of a protection group remain out of service until a fail-over occurs, at which point a power-on-reset procedure is initiated on the redundant spare prior to its being placed in service. Due to its poor recovery performance, cold sparing is better suited for systems having only high-reliability (MTBF) requirements as opposed to those also having high-availability requirements.
- *Exception handling.* Once an exception has been detected, the system must handle it in some fashion. The easiest thing it can do is simply to crash, but of course that's a terrible idea from the point of availability, usability, testability, and plain good sense. There are much more productive possibilities. The mechanism employed for exception handling depends largely on the programming environment employed, ranging from simple function return codes (error codes) to the use of exception classes that contain information helpful in fault correlation, such as the name of the exception thrown, the origin of the exception, and the cause of the exception thrown. Software can then use this information to mask the fault, usually by correcting the cause of the exception and retrying the operation.
- *Rollback.* This tactic permits the system to revert to a previous known good state, referred to as the "rollback line"—rolling back time—upon the detection of a failure. Once the good state is reached, then execution can continue. This tactic is often combined with active or passive redundancy tactics so that after a rollback has occurred, a standby version of the failed component is promoted to active status. Rollback depends on a copy of a previous good state (a checkpoint) being available to the components that are rolling back. Checkpoints can be stored in a fixed location and updated at regular intervals, or at convenient or significant times in the processing, such as at the completion of a complex operation.
- *Software upgrade* is another preparation-and-repair tactic whose goal is to achieve in-service upgrades to executable code images in a non-service-affecting manner. This may be realized as a function patch, a class patch, or a hitless in-service software upgrade (ISSU). A function patch is used in procedural programming and employs an incremental linker/loader to store an updated software function into a pre-allocated segment of target memory. The new version of the software function will employ the entry and exit points of the deprecated function. Also, upon loading the new software function, the symbol table must be updated and the instruction cache invalidated. The class patch tactic is applicable for targets executing object-oriented code, where the class definitions include a back-door mechanism that enables the runtime addition of member data and functions. Hitless in-service software upgrade leverages the active redundancy or passive

redundancy tactics to achieve non-service-affecting upgrades to software and associated schema. In practice, the function patch and class patch are used to deliver bug fixes, while the hitless in-service software upgrade is used to deliver new features and capabilities.

- *Retry.* The retry tactic assumes that the fault that caused a failure is transient and retrying the operation may lead to success. This tactic is used in networks and in server farms where failures are expected and common. There should be a limit on the number of retries that are attempted before a permanent failure is declared.
- *Ignore faulty behavior.* This tactic calls for ignoring messages sent from a particular source when we determine that those messages are spurious. For example, we would like to ignore the messages of an external component launching a denial-of-service attack by establishing Access Control List filters, for example.
- The *degradation* tactic maintains the most critical system functions in the presence of component failures, dropping less critical functions. This is done in circumstances where individual component failures gracefully reduce system functionality rather than causing a complete system failure.
- *Reconfiguration* attempts to recover from component failures by reassigning responsibilities to the (potentially restricted) resources left functioning, while maintaining as much functionality as possible.

Reintroduction is where a failed component is reintroduced after it has been corrected. Reintroduction tactics include the following:

- The *shadow* tactic refers to operating a previously failed or in-service upgraded component in a "shadow mode" for a predefined duration of time prior to reverting the component back to an active role. During this duration its behavior can be monitored for correctness and it can repopulate its state incrementally.
- *State resynchronization* is a reintroduction partner to the active redundancy and passive redundancy preparation-and-repair tactics. When used alongside the active redundancy tactic, the state resynchronization occurs organically, because the active and standby components each receive and process identical inputs in parallel. In practice, the states of the active and standby components are periodically compared to ensure synchronization. This comparison may be based on a cyclic redundancy check calculation (checksum) or, for systems providing safety-critical services, a message digest calculation (a one-way hash function). When used alongside the passive redundancy (warm spare) tactic, state resynchronization is based solely on periodic state information transmitted from the active component(s) to the standby component(s), typically via checkpointing. A special case of this tactic is found in stateless services, whereby any resource can handle a request from another (failed) resource.

- *Escalating restart* is a reintroduction tactic that allows the system to recover from faults by varying the granularity of the component(s) restarted and minimizing the level of service affected. For example, consider a system that supports four levels of restart, as follows. The lowest level of restart (call it Level 0), and hence having the least impact on services, employs passive redundancy (warm spare), where all child threads of the faulty component are killed and recreated. In this way, only data associated with the child threads is freed and reinitialized. The next level of restart (Level 1) frees and reinitializes all unprotected memory (protected memory would remain untouched). The next level of restart (Level 2) frees and reinitializes all memory, both protected and unprotected, forcing all applications to reload and reinitialize. And the final level of restart (Level 3) would involve completely reloading and reinitializing the executable image and associated data segments. Support for the escalating restart tactic is particularly useful for the concept of graceful degradation, where a system is able to degrade the services it provides while maintaining support for mission-critical or safety-critical applications.
- *Non-stop forwarding* (NSF) is a concept that originated in router design. In this design functionality is split into two parts: supervisory, or control plane (which manages connectivity and routing information), and data plane (which does the actual work of routing packets from sender to receiver). If a router experiences the failure of an active supervisor, it can continue forwarding packets along known routes—with neighboring routers—while the routing protocol information is recovered and validated. When the control plane is restarted, it implements what is sometimes called "graceful restart," incrementally rebuilding its routing protocol database even as the data plane continues to operate.

Prevent Faults

Instead of detecting faults and then trying to recover from them, what if your system could prevent them from occurring in the first place? Although this sounds like some measure of clairvoyance might be required, it turns out that in many cases it is possible to do just that.[3]

- *Removal from service.* This tactic refers to temporarily placing a system component in an out-of-service state for the purpose of mitigating potential system failures. One example involves taking a component of a system out of service and resetting the component in order to scrub latent faults (such

[3] These tactics deal with runtime means to prevent faults from occurring. Of course, an excellent way to prevent faults—at least in the system you're building, if not in systems that your system must interact with—is to produce high-quality code. This can be done by means of code inspections, pair programming, solid requirements reviews, and a host of other good engineering practices.

as memory leaks, fragmentation, or soft errors in an unprotected cache) before the accumulation of faults affects service (resulting in system failure). Another term for this tactic is *software rejuvenation*.

- *Transactions*. Systems targeting high-availability services leverage transactional semantics to ensure that asynchronous messages exchanged between distributed components are *atomic*, *consistent*, *isolated*, and *durable*. These four properties are called the "ACID properties." The most common realization of the transactions tactic is "two-phase commit" (a.k.a. 2PC) protocol. This tactic prevents race conditions caused by two processes attempting to update the same data item.
- *Predictive model*. A predictive model, when combined with a monitor, is employed to monitor the state of health of a system process to ensure that the system is operating within its nominal operating parameters, and to take corrective action when conditions are detected that are predictive of likely future faults. The operational performance metrics monitored are used to predict the onset of faults; examples include session establishment rate (in an HTTP server), threshold crossing (monitoring high and low water marks for some constrained, shared resource), or maintaining statistics for process state (in service, out of service, under maintenance, idle), message queue length statistics, and so on.
- *Exception prevention.* This tactic refers to techniques employed for the purpose of preventing system exceptions from occurring. The use of exception classes, which allows a system to transparently recover from system exceptions, was discussed previously. Other examples of exception prevention include abstract data types, such as smart pointers, and the use of wrappers to prevent faults, such as dangling pointers and semaphore access violations from occurring. Smart pointers prevent exceptions by doing bounds checking on pointers, and by ensuring that resources are automatically deallocated when no data refers to it. In this way resource leaks are avoided.
- *Increase competence set.* A program's competence set is the set of states in which it is "competent" to operate. For example, the state when the denominator is zero is outside the competence set of most divide programs. When a component raises an exception, it is signaling that it has discovered itself to be outside its competence set; in essence, it doesn't know what to do and is throwing in the towel. Increasing a component's competence set means designing it to handle more cases—faults—as part of its normal operation. For example, a component that assumes it has access to a shared resource might throw an exception if it discovers that access is blocked. Another component might simply wait for access, or return immediately with an indication that it will complete its operation on its own the next time it does have access. In this example, the second component has a larger competence set than the first.

5.3 A Design Checklist for Availability

Table 5.4 is a checklist to support the design and analysis process for availability.

TABLE 5.4 Checklist to Support the Design and Analysis Process for Availability

Category	Checklist
Allocation of Responsibilities	Determine the system responsibilities that need to be highly available. Within those responsibilities, ensure that additional responsibilities have been allocated to detect an omission, crash, incorrect timing, or incorrect response. Additionally, ensure that there are responsibilities to do the following: ▪ Log the fault ▪ Notify appropriate entities (people or systems) ▪ Disable the source of events causing the fault ▪ Be temporarily unavailable ▪ Fix or mask the fault/failure ▪ Operate in a degraded mode
Coordination Model	Determine the system responsibilities that need to be highly available. With respect to those responsibilities, do the following: ▪ Ensure that coordination mechanisms can detect an omission, crash, incorrect timing, or incorrect response. Consider, for example, whether guaranteed delivery is necessary. Will the coordination work under conditions of degraded communication? ▪ Ensure that coordination mechanisms enable the logging of the fault, notification of appropriate entities, disabling of the source of the events causing the fault, fixing or masking the fault, or operating in a degraded mode. ▪ Ensure that the coordination model supports the replacement of the artifacts used (processors, communications channels, persistent storage, and processes). For example, does replacement of a server allow the system to continue to operate? ▪ Determine if the coordination will work under conditions of degraded communication, at startup/shutdown, in repair mode, or under overloaded operation. For example, how much lost information can the coordination model withstand and with what consequences?
Data Model	Determine which portions of the system need to be highly available. Within those portions, determine which data abstractions, along with their operations or their properties, could cause a fault of omission, a crash, incorrect timing behavior, or an incorrect response. For those data abstractions, operations, and properties, ensure that they can be disabled, be temporarily unavailable, or be fixed or masked in the event of a fault. For example, ensure that write requests are cached if a server is temporarily unavailable and performed when the server is returned to service.

Category	Checklist
Mapping among Architectural Elements	Determine which artifacts (processors, communication channels, persistent storage, or processes) may produce a fault: omission, crash, incorrect timing, or incorrect response. Ensure that the mapping (or remapping) of architectural elements is flexible enough to permit the recovery from the fault. This may involve a consideration of the following: ▪ Which processes on failed processors need to be reassigned at runtime ▪ Which processors, data stores, or communication channels can be activated or reassigned at runtime ▪ How data on failed processors or storage can be served by replacement units ▪ How quickly the system can be reinstalled based on the units of delivery provided ▪ How to (re)assign runtime elements to processors, communication channels, and data stores ▪ When employing tactics that depend on redundancy of functionality, the mapping from modules to redundant components is important. For example, it is possible to write one module that contains code appropriate for both the active component and backup components in a protection group.
Resource Management	Determine what critical resources are necessary to continue operating in the presence of a fault: omission, crash, incorrect timing, or incorrect response. Ensure there are sufficient remaining resources in the event of a fault to log the fault; notify appropriate entities (people or systems); disable the source of events causing the fault; be temporarily unavailable; fix or mask the fault/failure; operate normally, in startup, shutdown, repair mode, degraded operation, and overloaded operation. Determine the availability time for critical resources, what critical resources must be available during specified time intervals, time intervals during which the critical resources may be in a degraded mode, and repair time for critical resources. Ensure that the critical resources are available during these time intervals. For example, ensure that input queues are large enough to buffer anticipated messages if a server fails so that the messages are not permanently lost.

continues

TABLE 5.4 Checklist to Support the Design and Analysis Process for Availability, *continued*

Category	Checklist
Binding Time	Determine how and when architectural elements are bound. If late binding is used to alternate between components that can themselves be sources of faults (e.g., processes, processors, communication channels), ensure the chosen availability strategy is sufficient to cover faults introduced by all sources. For example: ▪ If late binding is used to switch between artifacts such as processors that will receive or be the subject of faults, will the chosen fault detection and recovery mechanisms work for all possible bindings? ▪ If late binding is used to change the definition or tolerance of what constitutes a fault (e.g., how long a process can go without responding before a fault is assumed), is the recovery strategy chosen sufficient to handle all cases? For example, if a fault is flagged after 0.1 milliseconds, but the recovery mechanism takes 1.5 seconds to work, that might be an unacceptable mismatch. ▪ What are the availability characteristics of the late binding mechanism itself? Can it fail?
Choice of Technology	Determine the available technologies that can (help) detect faults, recover from faults, or reintroduce failed components. Determine what technologies are available that help the response to a fault (e.g., event loggers). Determine the availability characteristics of chosen technologies themselves: What faults can they recover from? What faults might they introduce into the system?

5.4 Summary

Availability refers to the ability of the system to be available for use, especially after a fault occurs. The fault must be recognized (or prevented) and then the system must respond in some fashion. The response desired will depend on the criticality of the application and the type of fault and can range from “ignore it” to “keep on going as if it didn’t occur.”

Tactics for availability are categorized into detect faults, recover from faults and prevent faults. Detection tactics depend, essentially, on detecting signs of life from various components. Recovery tactics are some combination of retrying an operation or maintaining redundant data or computations. Prevention tactics depend either on removing elements from service or utilizing mechanisms to limit the scope of faults.

All of the availability tactics involve the coordination model because the coordination model must be aware of faults that occur to generate an appropriate response.

5.5 For Further Reading

Patterns for availability:

- You can find patterns for fault tolerance in [Hanmer 07].

Tactics for availability, overall:

- A more detailed discussion of some of the availability tactics in this chapter is given in [Scott 09]. This is the source of much of the material in this chapter.
- The Internet Engineering Task Force has promulgated a number of standards supporting availability tactics. These standards include non-stop forwarding [IETF 04], ping/echo ICMPv6 [IETF 06b], echo request/response), and MPLS (LSP Ping) networks [IETF 06a].

Tactics for availability, fault detection:

- The parameter fence tactic was first used (to our knowledge) in the Control Data Series computers of the late 1960s.
- Triple modular redundancy (TMR), part of the voting tactic, was developed in the early 1960s by Lyons [Lyons 62].
- The fault detection tactic of voting is based on the fundamental contributions to automata theory by Von Neumann, who demonstrated how systems having a prescribed reliability could be built from unreliable components [Von Neumann 56].

Tactics for availability, fault recovery:

- Standards-based realizations of active redundancy exist for protecting network links (i.e., facilities) at both the physical layer [Bellcore 99, Telcordia 00] and the network/link layer [IETF 05].
- Exception handlinghas been written about by [Powel Douglass 99]. Software can then use this information to mask the fault, usually by correcting the cause of the exception and retrying the operation.
- [Morelos-Zaragoza 06] and [Schneier 96] have written about the comparison of state during resynchronization.
- Some examples of how a system can degrade through use (degradation) are given in [Nygard 07].
- [Utas 05] has written about escalating restart.

- Mountains of papers have been written about parameter typing, but [Utas 05] writes about it in the context of availability (as opposed to bug prevention, its usual context).
- Hardware engineers often use preparation-and-repair tactics. Examples include error detection and correction (EDAC) coding, forward error correction (FEC), and temporal redundancy. EDAC coding is typically used to protect control memory structures in high-availability distributed real-time embedded systems [Hamming 80]. Conversely, FEC coding is typically employed to recover from physical-layer errors occurring on external network links Morelos-Zaragoza 06]. Temporal redundancy involves sampling spatially redundant clock or data lines at time intervals that exceed the pulse width of any transient pulse to be tolerated, and then voting out any defects detected [Mavis 02].

Tactics for availability, fault prevention:

- Parnas and Madey have written about increasing an element's competence set [Parnas 95].
- The ACID properties, important in the transactions tactic, were introduced by Gray in the 1970s and discussed in depth in [Gray 93].

Analysis:

- Fault tree analysis dates from the early 1960s, but the granddaddy of resources for it is the U.S. Nuclear Regulatory Commission's "Fault Tree Handbook," published in 1981 [Vesely 81]. NASA's 2002 "Fault Tree Handbook with Aerospace Applications" [Vesely 02] is an updated comprehensive primer of the NRC handbook, and the source for the notation used in this chapter. Both are available online as downloadable PDF files.

5.6 Discussion Questions

1. Write a set of concrete scenarios for availability using each of the possible responses in the general scenario.
2. Write a concrete availability scenario for the software for a (hypothetical) pilotless passenger aircraft.
3. Write a concrete availability scenario for a program like Microsoft Word.
4. Redundancy is often cited as a key strategy for achieving high availability. Look at the tactics presented in this chapter and decide how many of them exploit some form of redundancy and how many do not.
5. How does availability trade off against modifiability? How would you make a change to a system that is required to have "24/7" availability (no scheduled or unscheduled downtime, ever)?

6. Create a fault tree for an automatic teller machine. Include faults dealing with hardware component failure, communications failure, software failure, running out of supplies, user errors, and security attacks. How would you modify your automatic teller machine design to accommodate these faults?

7. Consider the fault detection tactics (ping/echo, heartbeat, system monitor, voting, and exception detection). What are the performance implications of using these tactics?

This page intentionally left blank

6

Interoperability

With Liming Zhu

> *The early bird (A) arrives and catches worm (B), pulling string (C) and shooting off pistol (D). Bullet (E) bursts balloon (F), dropping brick (G) on bulb (H) of atomizer (I) and shooting perfume (J) on sponge (K). As sponge gains in weight, it lowers itself and pulls string (L), raising end of board (M). Cannon ball (N) drops on nose of sleeping gentleman. String tied to cannon ball releases cork (O) of vacuum bottle (P) and ice water falls on sleeper's face to assist the cannon ball in its good work.*
>
> —Rube Goldberg, instructions for "a simple alarm clock"

Interoperability is about the degree to which two or more systems can usefully exchange meaningful information via interfaces in a particular context. The definition includes not only having the ability to exchange data (syntactic interoperability) but also having the ability to correctly interpret the data being exchanged (semantic interoperability). A system cannot be interoperable in isolation. Any discussion of a system's interoperability needs to identify with whom, with what, and under what circumstances—hence, the need to include the context.

Interoperability is affected by the systems expected to interoperate. If we already know the interfaces of external systems with which our system will interoperate, then we can design that knowledge into the system. Or we can design our system to interoperate in a more generic fashion, so that the identity and the services that another system provides can be bound later in the life cycle, at build time or runtime.

Like all quality attributes, interoperability is not a yes-or-no proposition but has shades of meaning. There are several characterizing frameworks for interoperability, all of which seem to define five levels of interoperability "maturity" (see the "For Further Reading" section at the end of this chapter for a pointer). The lowest level signifies systems that do not share data at all, or do not do so with any success. The highest level signifies systems that work together seamlessly, never make any mistakes interpreting each other's communications, and share the same underlying semantic model of the world in which they work.

"Exchanging Information via Interfaces"

Interoperability, as we said, is about two or more systems exchanging information via interfaces.

At this point, we need to clarify two critical concepts central to this discussion and emphasize that we are taking a broad view of each.

The first is what it means to "exchange information." This can mean something as simple as program A calling program B with some parameters. However, two systems (or parts of a system) can exchange information even if they never communicate directly with each other. Did you ever have a conversation like the following in junior high school? "Charlene said that Kim told her that Trevor heard that Heather wants to come to your party." Of course, junior high school protocol would preclude the possibility of responding directly to Heather. Instead, your response (if you like Heather) might be, "Cool," which would make its way back through Charlene, Kim, and Trevor. You and Heather exchanged information, but never talked to each other. (We hope you got to talk to each other at the party.)

Entities can exchange information in even less direct ways. If I have an idea of a program's behavior, and I design my program to work assuming that behavior, the two programs have also exchanged information—just not at runtime.

One of the more infamous software disasters in history occurred when an antimissile system failed to intercept an incoming ballistic rocket in Operation Desert Storm in 1991, resulting in 28 fatalities. One of the missile's software components "expected" to be shut down and restarted periodically, so it could recalibrate its orientation framework from a known initial point. The software had been running for some 100 hours when the missile was launched, and calculation errors had accumulated to the point where the software component's idea of its orientation had wandered hopelessly away from truth.

Systems (or components within systems) often have or embody expectations about the behaviors of its "information exchange" partners. The assumption of everything interacting with the errant component in the preceding example was that its accuracy did *not* degrade over time. The result was a system of parts that did not work together correctly to solve the problem they were supposed to.

The second concept we need to stress is what we mean by "interface." Once again, we mean something beyond the simple case—a syntactic description of a component's programs and the type and number of their parameters, most commonly realized as an API. That's necessary for interoperability—heck, it's necessary if you want your software to compile successfully—but it's not sufficient. To illustrate this concept, we'll use another "conversation" analogy. Has your partner or spouse ever come home, slammed the door, and when you ask what's wrong, replied "Nothing!"? If so, then you should be able to appreciate the keen difference between syntax and semantics and the role of expectations in understanding how an entity behaves. Because we want interoperable systems and components, and not simply ones that compile together nicely, we require a higher bar for interfaces than just a statement of syntax. By "interface," we mean the set of assumptions that you can safely make about an entity. For example, it's a safe assumption that whatever's wrong with your spouse/partner, it's not "Nothing," and you know that because that "interface" extends *way* beyond just the words they say. And it's also a safe assumption that nothing about our missile component's accuracy degradation over time was in its API, and yet that was a critical part of its interface.

—PCC

Here are some of the reasons you might want systems to interoperate:

- Your system provides a service to be used by a collection of unknown systems. These systems need to interoperate with your system even though you may know nothing about them. An example is a service such as Google Maps.
- You are constructing capabilities from existing systems. For example, one of the existing systems is responsible for sensing its environment, another one is responsible for processing the raw data, a third is responsible for interpreting the data, and a final one is responsible for producing and distributing a representation of what was sensed. An example is a traffic sensing system where the input comes from individual vehicles, the raw data is processed into common units of measurement, is interpreted and fused, and traffic congestion information is broadcast.

These examples highlight two important aspects of interoperability:

1. *Discovery.* The consumer of a service must discover (possibly at runtime, possibly prior to runtime) the location, identity, and the interface of the service.
2. *Handling of the response.* There are three distinct possibilities:
   - The service reports back to the requester with the response.
   - The service sends its response on to another system.
   - The service broadcasts its response to any interested parties.

These elements, discovery and disposition of response, along with management of interfaces, govern our discussion of scenarios and tactics for interoperability.

Systems of Systems

If you have a group of systems that are interoperating to achieve a joint purpose, you have what is called a *system of systems* (SoS). An SoS is an arrangement of systems that results when independent and useful systems are integrated into a larger system that delivers unique capabilities. Table 6.1 shows a categorization of SoSs.

TABLE 6.1 Taxonomy of Systems of Systems*

Directed	SoS objectives, centralized management, funding, and authority for the overall SoS are in place. Systems are subordinated to the SoS.
Acknowledged	SoS objectives, centralized management, funding, and authority in place. However, systems retain their own management, funding, and authority in parallel with the SoS.
Collaborative	There are no overall objectives, centralized management, authority, responsibility, or funding at the SoS level. Systems voluntarily work together to address shared or common interests.
Virtual	Like collaborative, but systems don't know about each other.

* *The taxonomy shown is an extension of work done by Mark Maier in 1998.*

In directed and acknowledged SoSs, there is a deliberate attempt to create an SoS. The key difference is that in the former, there is SoS-level management that exercises control over the constituent systems, while in the latter, the constituent systems retain a high degree of autonomy in their own evolution. Collaborative and virtual systems of systems are more ad hoc, absent an overarching authority or source of funding and, in the case of a virtual SoS, even absent the knowledge about the scope and membership of the SoS.

The collaborative case is quite common. Consider the Google Maps example from the introduction. Google is the manager and funding authority for the map service. Each use of the maps in an application (an SoS) has its own management and funding authority, and there is no overall management of all of the applications that use Google Maps. The various organizations involved in the applications collaborate (either explicitly or implicitly) to enable the applications to work correctly.

A virtual SoS involves large systems and is much more ad hoc. For example, there are over 3,000 electric companies in the U.S. electric grid, each state has a public utility commission that oversees the utility companies operating in its state, and the federal Department of Energy provides some level of policy guidance. Many of the systems within the electric grid must interoperate, but there is no management authority for the overall system.

6.1 Interoperability General Scenario

The following are the portions of an interoperability general scenario:

- *Source of stimulus*. A system that initiates a request.
- *Stimulus*. A request to exchange information among systems.
- *Artifacts*. The systems that wish to interoperate.
- *Environment*. The systems that wish to interoperate are discovered at runtime or are known prior to runtime.
- *Response*. The request to interoperate results in the exchange of information. The information is understood by the receiving party both syntactically and semantically. Alternatively, the request is rejected and appropriate entities are notified. In either case, the request may be logged.
- *Response measure*. The percentage of information exchanges correctly processed or the percentage of information exchanges correctly rejected.

Figure 6.1 gives an example: Our vehicle information system sends our current location to the traffic monitoring system. The traffic monitoring system combines our location with other information, overlays this information on a Google Map, and broadcasts it. Our location information is correctly included with a probability of 99.9%.

Table 6.2 presents the possible values for each portion of an interoperability scenario.

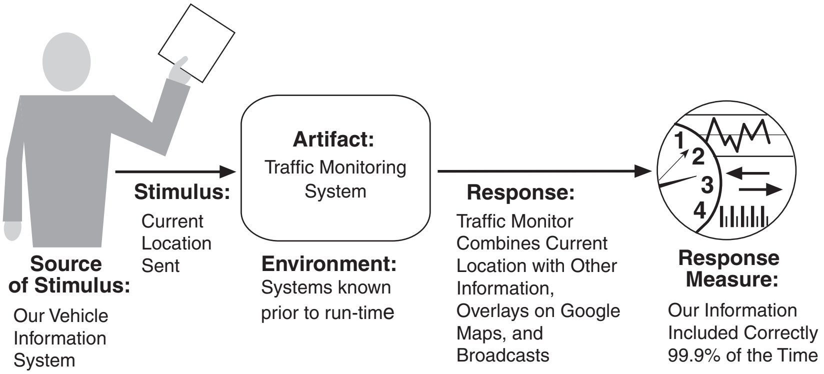

Artifact:
Traffic Monitoring System
Stimulus:
Current Location Sent
Source of Stimulus:
Our Vehicle Information System
Environment:
Systems known prior to run-time
Response:
Traffic Monitor Combines Current Location with Other Information, Overlays on Google Maps, and Broadcasts
1
2
3
4
Response Measure:
Our Information Included Correctly 99.9% of the Time

FIGURE 6.1 Sample concrete interoperability scenario

TABLE 6.2 General Interoperability Scenario

Portion of Scenario	Possible Values
Source	A system initiates a request to interoperate with another system.
Stimulus	A request to exchange information among system(s).
Artifact	The systems that wish to interoperate.
Environment	System(s) wishing to interoperate are discovered at runtime or known prior to runtime.
Response	One or more of the following: ▪ The request is (appropriately) rejected and appropriate entities (people or systems) are notified. ▪ The request is (appropriately) accepted and information is exchanged successfully. ▪ The request is logged by one or more of the involved systems.
Response Measure	One or more of the following: ▪ Percentage of information exchanges correctly processed ▪ Percentage of information exchanges correctly rejected

SOAP vs. REST

If you want to allow web-based applications to interoperate, you have two major off-the-shelf technology options today: (1) WS* and SOAP (which once stood for "Simple Object Access Protocol," but that acronym is no longer blessed) and (2) REST (which stands for "Representation State Transfer," and therefore is sometimes spelled ReST). How can we compare these technologies? What is each good for? What are the road hazards you need to be aware of? This is a bit of an apples-and-oranges comparison, but I will try to sketch the landscape.

SOAP is a protocol specification for XML-based information that distributed applications can use to exchange information and hence interoperate. It is most often accompanied by a set of SOA middleware interoperability standards and compliant implementations, referred to (collectively) as WS*. SOAP and WS* together define many standards, including the following:

- *An infrastructure for service composition.* SOAP can employ the Business Process Execution Language (BPEL) as a way to let developers express business processes that are implemented as WS* services.
- *Transactions.* There are several web-service standards for ensuring that transactions are properly managed: WS-AT, WS-BA, WS-CAF, and WS-Transaction.
- *Service discovery.* The Universal Description, Discovery and Integration (UDDI) language enables businesses to publish service listings and discover each other.

- *Reliability.* SOAP, by itself, does not ensure reliable message delivery. Applications that require such guarantees must use services compliant with SOAP's reliability standard: WS-Reliability.

SOAP is quite general and has its roots in a remote procedure call (RPC) model of interacting applications, although other models are certainly possible. SOAP has a simple type system, comparable to that found in the major programming languages. SOAP relies on HTTP and RPC for message transmission, but it could, in theory, be implemented on top of any communication protocol. SOAP does not mandate a service's method names, addressing model, or procedural conventions. Thus, choosing SOAP buys little actual interoperability between applications—it is just an information exchange standard. The interacting applications need to agree on *how to interpret* the payload, which is where you get semantic interoperability.

REST, on the other hand, is a client-server-based architectural style that is structured around a small set of create, read, update, delete (CRUD) operations (called POST, GET, PUT, DELETE respectively in the REST world) and a single addressing scheme (based on a URI, or uniform resource identifier). REST imposes few constraints on an architecture: SOAP offers completeness; REST offers simplicity.

REST is about state and state transfer and views the web (and the services that service-oriented systems can string together) as a huge network of information that is accessible by a single URI-based addressing scheme. There is no notion of type and hence no type checking in REST—it is up to the applications to get the semantics of interaction right.

Because REST interfaces are so simple and general, any HTTP client can talk to any HTTP server, using the REST operations (POST, GET, PUT, DELETE) with no further configuration. That buys you syntactic interoperability, but of course there must be organization-level agreement about what these programs actually do and what information they exchange. That is, semantic interoperability is not guaranteed between services just because both have REST interfaces.

REST, on top of HTTP, is meant to be self-descriptive and in the best case is a stateless protocol. Consider the following example, in REST, of a phone book service that allows someone to look up a person, given some unique identifier for that person:

```
http://www.XYZdirectory.com/phonebook/UserInfo/99999
```

The same simple lookup, implemented in SOAP, would be specified as something like the following:

```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap=http://www.w3.org/2001/
     12/soap-envelope
 soap:encodingStyle="http://www.w3.org/2001/12/
     soap-encoding">
   <soap:Body pb="http://www.XYZdirectory.com/
     phonebook">
```

```
      <pb:GetUserInfo>
        <pb:UserIdentifier>99999</pb:UserIdentifier>
      </pb:GetUserInfo>
    </soap:Body>
  </soap:Envelope>
```

One aspect of the choice between SOAP and REST is whether you want to accept the complexity and restrictions of SOAP+WSDL (the Web Services Description Language) to get more standardized interoperability or if you want to avoid the overhead by using REST, but perhaps benefit from less standardization. What are the other considerations?

A message exchange in REST has somewhat fewer characters than a message exchange in SOAP. So one of the tradeoffs in the choice between REST and SOAP is the size of the individual messages. For systems exchanging a large number of messages, another tradeoff is between performance (favoring REST) and structured messages (favoring SOAP).

The decision to implement WS* or REST will depend on aspects such as the quality of service (QoS) required—WS* implementation has greater support for security, availability, and so on—and type of functionality. A RESTful implementation, because of its simplicity, is more appropriate for read-only functionality, typical of mashups, where there are minimal QoS requirements and concerns.

OK, so if you are building a service-based system, how do you choose? The truth is, you don't have to make a single choice, once and for all time; each technology is reasonably easy to use, at least for simple applications. And each has its strengths and weaknesses. Like everything else in architecture, it's all about the tradeoffs; your decision will likely hinge on the way those tradeoffs affect your system in your context.

—RK

6.2 Tactics for Interoperability

Figure 6.2 shows the goal of the set of interoperability tactics.

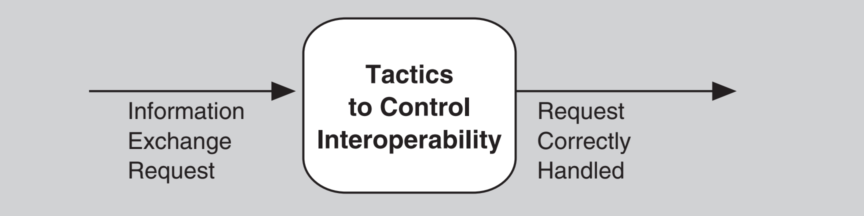

Information Exchange Request
Tactics to Control Interoperability
Request Correctly Handled

FIGURE 6.2 Goal of interoperability tactics

We identify two categories of interoperability tactics: locate and manage interfaces.

Locate

There is only one tactic in this category: *discover service*. It is used when the systems that interoperate must be discovered at runtime.

- *Discover service*. Locate a service through searching a known directory service. (By "service," we simply mean a set of capabilities that is accessible via some kind of interface.) There may be multiple levels of indirection in this location process—that is, a known location points to another location that in turn can be searched for the service. The service can be located by type of service, by name, by location, or by some other attribute.

Manage Interfaces

Managing interfaces consists of two tactics: *orchestrate* and *tailor interface*.

- *Orchestrate*. Orchestrate is a tactic that uses a control mechanism to coordinate and manage and sequence the invocation of particular services (which could be ignorant of each other). Orchestration is used when the interoperating systems must interact in a complex fashion to accomplish a complex task; orchestration "scripts" the interaction. Workflow engines are an example of the use of the orchestrate tactic. The mediator design pattern can serve this function for simple orchestration. Complex orchestration can be specified in a language such as BPEL.
- *Tailor interface*. Tailor interface is a tactic that adds or removes capabilities to an interface. Capabilities such as translation, adding buffering, or smoothing data can be added. Capabilities may be removed as well. An example of removing capabilities is to hide particular functions from untrusted users. The decorator pattern is an example of the tailor interface tactic.

The enterprise service bus that underlies many service-oriented architectures combines both of the manage interface tactics.

Figure 6.3 shows a summary of the tactics to achieve interoperability.

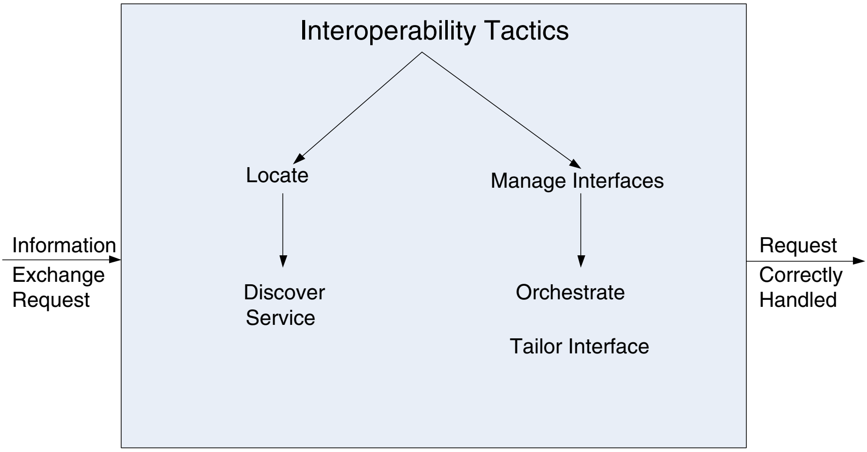


FIGURE 6.3 Summary of interoperability tactics

Why Standards Are Not Enough to Guarantee Interoperability
By Grace Lewis

Developer of System A needs to exchange product data with System B. Developer A finds that there is an existing WS* web service interface for sending product data that among other fields contains price expressed in XML Schema as a decimal with two fraction digits. Developer A writes code to interact with the web service and the system works perfectly. However, after two weeks of operation, there is a huge discrepancy between the totals reported by System A and the totals reported by System B. After conversations between the two developers, they discover that System B expected to receive a price that included tax and System A was sending it without tax.

This is a simple example of why standards are not enough. The systems exchanged data perfectly because they both agreed that the price was a decimal with two fractions digits expressed in XML Schema and the message was sent via SOAP over HTTP (syntax)—standards used in the implementation of WS* web services—but they did not agree on whether the price included tax or not (semantics).

Of course, the only realistic approach to getting diverse applications to share information is by reaching agreements on the structure and function of the information to be shared. These agreements are often reflected in standards that provide a common interface that multiple vendors and application builders support. Standards have indeed been instrumental in achieving a significant level of interoperability that we rely on in almost every domain. However, while standards are useful and in many ways indispensable, expectations of what can be achieved through standards are unrealistic. Here are some of the challenges that organizations face related to standards and interoperability:

1. Ideally, every implementation of a standard should be identical and thus completely interoperable with any other implementation. However, this is far from reality. Standards, when incorporated into products, tools, and services, undergo customizations and extensions because every vendor wants to create a unique selling point as a competitive advantage.
2. Standards are often deliberately open-ended and provide extension points. The actual implementation of these extension points is left to the discretion of implementers, leading to proprietary implementations.
3. Standards, like any technology, have a life cycle of their own and evolve over time in compatible and noncompatible ways. Deciding when to adopt a new or revised standard is a critical decision for organizations. Committing to a new standard that is not ready or eventually not adopted by the community is a big risk for organizations. On the other hand, waiting too long may also become a problem, which can lead to unsupported products, incompatibilities, and workarounds, because everyone else is using the standard.
4. Within the software community, there are as many bad standards as there are engineers with opinions. Bad standards include underspecified, overspecified, inconsistently specified, unstable, or irrelevant standards.
5. It is quite common for standards to be championed by competing organizations, resulting in conflicting standards due to overlap or mutual exclusion.
6. For new and rapidly emerging domains, the argument often made is that standardization will be destructive because it will hinder flexibility: premature standardization will force the use of an inadequate approach and lead to abandoning other presumably better approaches. So what do organizations do in the meantime?

What these challenges illustrate is that because of the way in which standards are usually created and evolved, we cannot let standards drive our architectures. We need to architect systems first and then decide which standards can support desired system requirements and qualities. This approach allows standards to change and evolve without affecting the overall architecture of the system.

I once heard someone in a keynote address say that “The nice thing about standards is that there are so many to choose from.”

6.3 A Design Checklist for Interoperability

Table 6.3 is a checklist to support the design and analysis process for interoperability.

TABLE 6.3 Checklist to Support the Design and Analysis Process for Interoperability

Category	Checklist
Allocation of Responsibilities	Determine which of your system responsibilities will need to interoperate with other systems. Ensure that responsibilities have been allocated to detect a request to interoperate with known or unknown external systems. Ensure that responsibilities have been allocated to carry out the following tasks: ▪ Accept the request ▪ Exchange information ▪ Reject the request ▪ Notify appropriate entities (people or systems) ▪ Log the request (for interoperability in an untrusted environment, logging for nonrepudiation is essential)
Coordination Model	Ensure that the coordination mechanisms can meet the critical quality attribute requirements. Considerations for performance include the following: ▪ Volume of traffic on the network both created by the systems under your control and generated by systems not under your control ▪ Timeliness of the messages being sent by your systems ▪ Currency of the messages being sent by your systems ▪ Jitter of the messages' arrival times ▪ Ensure that all of the systems under your control make assumptions about protocols and underlying networks that are consistent with the systems not under your control.
Data Model	Determine the syntax and semantics of the major data abstractions that may be exchanged among interoperating systems. Ensure that these major data abstractions are consistent with data from the interoperating systems. (If your system's data model is confidential and must not be made public, you may have to apply transformations to and from the data abstractions of systems with which yours interoperates.)
Mapping among Architectural Elements	For interoperability, the critical mapping is that of components to processors. Beyond the necessity of making sure that components that communicate externally are hosted on processors that can reach the network, the primary considerations deal with meeting the security, availability, and performance requirements for the communication. These will be dealt with in their respective chapters.

Category	Checklist
Resource Management	Ensure that interoperation with another system (accepting a request and/or rejecting a request) can never exhaust critical system resources (e.g., can a flood of such requests cause service to be denied to legitimate users?). Ensure that the resource load imposed by the communication requirements of interoperation is acceptable. Ensure that if interoperation requires that resources be shared among the participating systems, an adequate arbitration policy is in place.
Binding Time	Determine the systems that may interoperate, and when they become known to each other. For each system over which you have control: ▪ Ensure that it has a policy for dealing with binding to both known and unknown external systems. ▪ Ensure that it has mechanisms in place to reject unacceptable bindings and to log such requests. ▪ In the case of late binding, ensure that mechanisms will support the discovery of relevant new services or protocols, or the sending of information using chosen protocols.
Choice of Technology	For any of your chosen technologies, are they “visible” at the interface boundary of a system? If so, what interoperability effects do they have? Do they support, undercut, or have no effect on the interoperability scenarios that apply to your system? Ensure the effects they have are acceptable. Consider technologies that are designed to support interoperability, such as web services. Can they be used to satisfy the interoperability requirements for the systems under your control?

6.4 Summary

Interoperability refers to the ability of systems to usefully exchange information. These systems may have been constructed with the intention of exchanging information, they may be existing systems that are desired to exchange information, or they may provide general services without knowing the details of the systems that wish to utilize those services.

The general scenario for interoperability provides the details of these different cases. In any interoperability case, the goal is to intentionally exchange information or reject the request to exchange information.

Achieving interoperability involves the relevant systems locating each other and then managing the interfaces so that they can exchange information.

6.5 For Further Reading

An SEI report gives a good overview of interoperability, and it highlights some of the “maturity frameworks” for interoperability [Brownsword 04].

The various WS* services are being developed under the auspices of the World Wide Web Consortium (W3C) and can be found at www.w3.org/2002/ws.

Systems of systems are of particular interest to the U.S. Department of Defense. An engineering guide can be found at [ODUSD 08].

6.6 Discussion Questions

1. Find a web service mashup. Write several concrete interoperability scenarios for this system.
2. What is the relationship between interoperability and the other quality attributes highlighted in this book? For example, if two systems fail to exchange information properly, could a security flaw result? What other quality attributes seem strongly related (at least potentially) to interoperability?
3. Is a service-oriented system a system of systems? If so, describe a service-oriented system that is directed, one that is acknowledged, one that is collaborative, and one that is virtual.
4. Universal Description, Discovery, and Integration (UDDI) was touted as a discovery service, but commercial support for UDDI is being withdrawn. Why do you suppose this is? Does it have anything to do with the quality attributes delivered or not delivered by UDDI solutions?
5. Why has the importance of orchestration grown in recent years?
6. If you are a technology producer, what are the advantages and disadvantages of adhering to interoperability standards? Why would a producer not adhere to a standard?
7. With what other systems will an automatic teller machine need to interoperate? How would you change your automatic teller system design to accommodate these other systems?

7

Modifiability

Adapt or perish, now as ever, is nature's inexorable imperative.
—H.G. Wells

Change happens.

Study after study shows that most of the cost of the typical software system occurs after it has been initially released. If change is the only constant in the universe, then software change is not only constant but ubiquitous. Changes happen to add new features, to change or even retire old ones. Changes happen to fix defects, tighten security, or improve performance. Changes happen to enhance the user's experience. Changes happen to embrace new technology, new platforms, new protocols, new standards. Changes happen to make systems work together, even if they were never designed to do so.

Modifiability is about change, and our interest in it centers on the cost and risk of making changes. To plan for modifiability, an architect has to consider four questions:

- *What can change?* A change can occur to any aspect of a system: the functions that the system computes, the platform (the hardware, operating system, middleware), the environment in which the system operates (the systems with which it must interoperate, the protocols it uses to communicate with the rest of the world), the qualities the system exhibits (its performance, its reliability, and even its future modifications), and its capacity (number of users supported, number of simultaneous operations).
- *What is the likelihood of the change?* One cannot plan a system for all potential changes—the system would never be done, or if it was done it would be far too expensive and would likely suffer quality attribute problems in other dimensions. Although anything *might* change, the architect has to make the tough decisions about which changes are likely, and hence which changes are to be supported, and which are not.

- *When is the change made and who makes it?* Most commonly in the past, a change was made to source code. That is, a developer had to make the change, which was tested and then deployed in a new release. Now, however, the question of when a change is made is intertwined with the question of who makes it. An end user changing the screen saver is clearly making a change to one of the aspects of the system. Equally clear, it is not in the same category as changing the system so that it can be used over the web rather than on a single machine. Changes can be made to the implementation (by modifying the source code), during compile (using compile-time switches), during build (by choice of libraries), during configuration setup (by a range of techniques, including parameter setting), or during execution (by parameter settings, plugins, etc.). A change can also be made by a developer, an end user, or a system administrator.
- *What is the cost of the change?* Making a system more modifiable involves two types of cost:
  - The cost of introducing the mechanism(s) to make the system more modifiable
  - The cost of making the modification using the mechanism(s)

For example, the simplest mechanism for making a change is to wait for a change request to come in, then change the source code to accommodate the request. The cost of introducing the mechanism is zero; the cost of exercising it is the cost of changing the source code and revalidating the system. At the other end of the spectrum is an application generator, such as a user interface builder. The builder takes as input a description of the designer user interface produced through direct manipulation techniques and produces (usually) source code. The cost of introducing the mechanism is the cost of constructing the UI builder, which can be substantial. The cost of using the mechanism is the cost of producing the input to feed the builder (cost can be substantial or negligible), the cost of running the builder (approximately zero), and then the cost of whatever testing is performed on the result (usually much less than usual).

For N similar modifications, a simplified justification for a change mechanism is that

$$\begin{gathered} N \times \text{Cost of making the change without the mechanism} \leq \\ \text{Cost of installing the mechanism} + \\ (N \times \text{Cost of making the change using the mechanism}). \end{gathered}$$

N is the anticipated number of modifications that will use the modifiability mechanism, but N is a prediction. If fewer changes than expected come in, then an expensive modification mechanism may not be warranted. In addition, the cost of creating the modifiability mechanism could be applied elsewhere—in adding functionality, in improving the performance, or even in nonsoftware investments such as buying tech stocks. Also, the equation does not take time into account. It might be cheaper in the long run to build a sophisticated change-handling mechanism, but you might not be able to wait for that.

7.1 Modifiability General Scenario

From these considerations, we can see the portions of the modifiability general scenario:

- *Source of stimulus.* This portion specifies who makes the change: the developer, a system administrator, or an end user.
- *Stimulus*. This portion specifies the change to be made. A change can be the addition of a function, the modification of an existing function, or the deletion of a function. (For this categorization, we regard fixing a defect as changing a function, which presumably wasn't working correctly as a result of the defect.) A change can also be made to the qualities of the system: making it more responsive, increasing its availability, and so forth. The capacity of the system may also change. Accommodating an increasing number of simultaneous users is a frequent requirement. Finally, changes may happen to accommodate new technology of some sort, the most common of which is porting the system to a different type of computer or communication network.
- *Artifact*. This portion specifies what is to be changed: specific components or modules, the system's platform, its user interface, its environment, or another system with which it interoperates.
- *Environment*. This portion specifies when the change can be made: design time, compile time, build time, initiation time, or runtime.
- *Response*. Make the change, test it, and deploy it.
- *Response measure*. All of the possible responses take time and cost money; time and money are the most common response measures. Although both sound simple to measure, they aren't. You can measure calendar time or staff time. But do you measure the time it takes for the change to wind its way through configuration control boards and approval authorities (some of whom may be outside your organization), or merely the time it takes your engineers to make the change? Cost usually means direct outlay, but it might also include opportunity cost of having your staff work on changes instead of other tasks. Other measures include the extent of the change (number of modules or other artifacts affected) or the number of new defects introduced by the change, or the effect on other quality attributes. If the change is being made by a user, you may wish to measure the efficacy of the change mechanisms provided, which somewhat overlaps with measures of usability (see Chapter 11).

Figure 7.1 illustrates a concrete modifiability scenario: The developer wishes to change the user interface by modifying the code at design time. The modifications are made with no side effects within three hours.

Table 7.1 enumerates the elements of the general scenario that characterize modifiability.

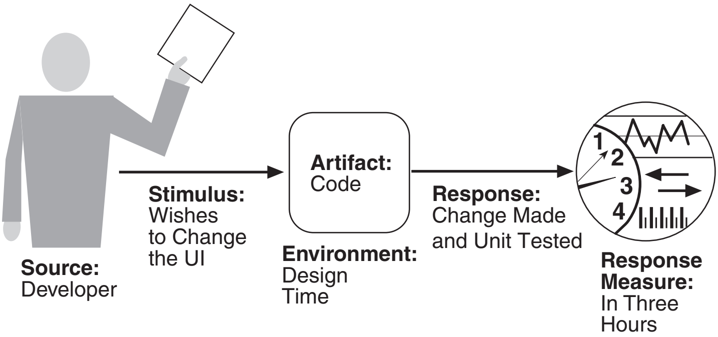


FIGURE 7.1 Sample concrete modifiability scenario

TABLE 7.1 Modifiability General Scenario

Portion of Scenario	Possible Values
Source	End user, developer, system administrator
Stimulus	A directive to add/delete/modify functionality, or change a quality attribute, capacity, or technology
Artifacts	Code, data, interfaces, components, resources, configurations, . . .
Environment	Runtime, compile time, build time, initiation time, design time
Response	One or more of the following: ▪ Make modification ▪ Test modification ▪ Deploy modification
Response Measure	Cost in terms of the following: ▪ Number, size, complexity of affected artifacts ▪ Effort ▪ Calendar time ▪ Money (direct outlay or opportunity cost) ▪ Extent to which this modification affects other functions or quality attributes ▪ New defects introduced

7.2 Tactics for Modifiability

Tactics to control modifiability have as their goal controlling the complexity of making changes, as well as the time and cost to make changes. Figure 7.2 shows this relationship.

To understand modifiability, we begin with coupling and cohesion.

Modules have responsibilities. When a change causes a module to be modified, its responsibilities are changed in some way. Generally, a change that affects one module is easier and less expensive than if it changes more than one module. However, if two modules' responsibilities overlap in some way, then a single change may well affect them both. We can measure this overlap by measuring the probability that a modification to one module will propagate to the other. This is called *coupling*, and high coupling is an enemy of modifiability.

Cohesion measures how strongly the responsibilities of a module are related. Informally, it measures the module's "unity of purpose." Unity of purpose can be measured by the change scenarios that affect a module. The cohesion of a module is the probability that a change scenario that affects a responsibility will also affect other (different) responsibilities. The higher the cohesion, the lower the probability that a given change will affect multiple responsibilities. High cohesion is good; low cohesion is bad. The definition allows for two modules with similar purposes each to be cohesive.

Given this framework, we can now identify the parameters that we will use to motivate modifiability tactics:

- *Size of a module*. Tactics that split modules will reduce the cost of making a modification to the module that is being split as long as the split is chosen to reflect the type of change that is likely to be made.

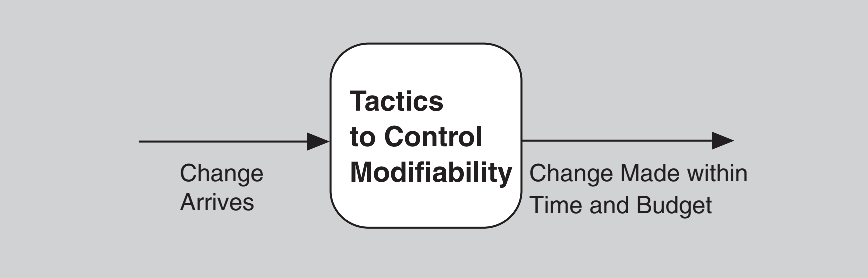

Tactics to Control Modifiability
Change Arrives
Change Made within Time and Budget

FIGURE 7.2 The goal of modifiability tactics

- *Coupling*. Reducing the strength of the coupling between two modules A and B will decrease the expected cost of any modification that affects A. Tactics that reduce coupling are those that place intermediaries of various sorts between modules A and B.
- *Cohesion*. If module A has a low cohesion, then cohesion can be improved by removing responsibilities unaffected by anticipated changes.

Finally we need to be concerned with when in the software development life cycle a change occurs. If we ignore the cost of preparing the architecture for the modification, we prefer that a change is bound as late as possible. Changes can only be successfully made (that is, quickly and at lowest cost) late in the life cycle if the architecture is suitably prepared to accommodate them. Thus the fourth and final parameter in a model of modifiability is this:

- *Binding time of modification*. An architecture that is suitably equipped to accommodate modifications late in the life cycle will, on average, cost less than an architecture that forces the same modification to be made earlier. The preparedness of the system means that some costs will be zero, or very low, for late life-cycle modifications. This, however, neglects the cost of preparing the architecture for the late binding.

Now we may understand tactics and their consequences as affecting one or more of the previous parameters: reducing the size of a module, increasing cohesion, reducing coupling, and deferring binding time. These tactics are shown in Figure 7.3.

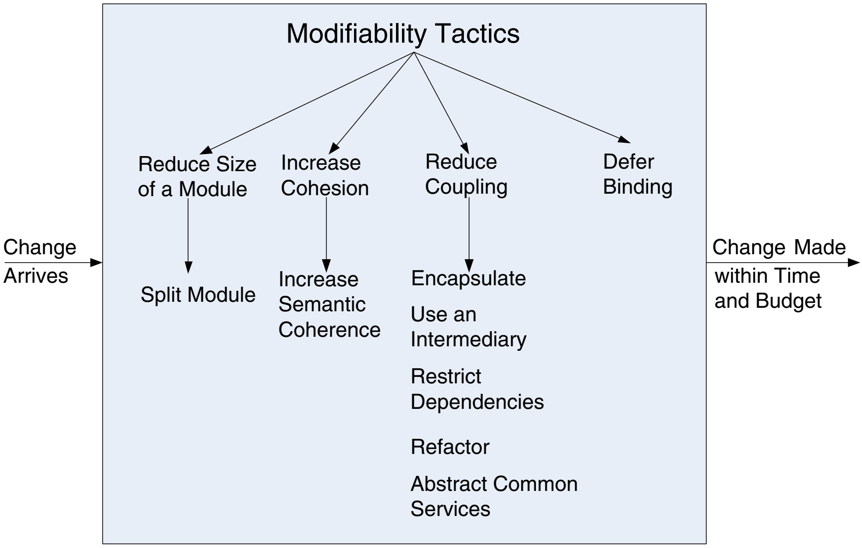


FIGURE 7.3 Modifiability tactics

Reduce the Size of a Module

- *Split module*. If the module being modified includes a great deal of capability, the modification costs will likely be high. Refining the module into several smaller modules should reduce the average cost of future changes.

Increase Cohesion

Several tactics involve moving responsibilities from one module to another. The purpose of moving a responsibility from one module to another is to reduce the likelihood of side effects affecting other responsibilities in the original module.

- *Increase semantic coherence*. If the responsibilities A and B in a module do not serve the same purpose, they should be placed in different modules. This may involve creating a new module or it may involve moving a responsibility to an existing module. One method for identifying responsibilities to be moved is to hypothesize likely changes that affect a module. If some responsibilities are not affected by these changes, then those responsibilities should probably be removed.

Reduce Coupling

We now turn to tactics that reduce the coupling between modules.

- *Encapsulate.* Encapsulation introduces an explicit interface to a module. This interface includes an application programming interface (API) and its associated responsibilities, such as "perform a syntactic transformation on an input parameter to an internal representation." Perhaps the most common modifiability tactic, encapsulation reduces the probability that a change to one module propagates to other modules. The strengths of coupling that previously went to the module now go to the interface for the module. These strengths are, however, reduced because the interface limits the ways in which external responsibilities can interact with the module (perhaps through a wrapper). The external responsibilities can now only directly interact with the module through the exposed interface (indirect interactions, however, such as dependence on quality of service, will likely remain unchanged). Interfaces designed to increase modifiability should be abstract with respect to the details of the module that are likely to change—that is, they should hide those details.
- *Use an intermediary* breaks a dependency. Given a dependency between responsibility A and responsibility B (for example, carrying out A first requires carrying out B), the dependency can be broken by using an intermediary. The type of intermediary depends on the type of dependency. For example, a publish-subscribe intermediary will remove the data producer's knowledge

of its consumers. So will a shared data repository, which separates readers of a piece of data from writers of that data. In a service-oriented architecture in which services discover each other by dynamic lookup, the directory service is an intermediary.

- *Restrict dependencies* is a tactic that restricts the modules that a given module interacts with or depends on. In practice this tactic is achieved by restricting a module's visibility (when developers cannot see an interface, they cannot employ it) and by authorization (restricting access to only authorized modules). This tactic is seen in layered architectures, in which a layer is only allowed to use lower layers (sometimes only the next lower layer) and in the use of wrappers, where external entities can only see (and hence depend on) the wrapper and not the internal functionality that it wraps.
- *Refactor* is a tactic undertaken when two modules are affected by the same change because they are (at least partial) duplicates of each other. Code refactoring is a mainstay practice of Agile development projects, as a cleanup step to make sure that teams have not produced duplicative or overly complex code; however, the concept applies to architectural elements as well. Common responsibilities (and the code that implements them) are "factored out" of the modules where they exist and assigned an appropriate home of their own. By co-locating common responsibilities—that is, making them submodules of the same parent module—the architect can reduce coupling.
- *Abstract common services*. In the case where two modules provide not-quite-the-same but similar services, it may be cost-effective to implement the services just once in a more general (abstract) form. Any modification to the (common) service would then need to occur just in one place, reducing modification costs. A common way to introduce an abstraction is by parameterizing the description (and implementation) of a module's activities. The parameters can be as simple as values for key variables or as complex as statements in a specialized language that are subsequently interpreted.

Defer Binding

Because the work of people is almost always more expensive than the work of computers, letting computers handle a change as much as possible will almost always reduce the cost of making that change. If we design artifacts with built-in flexibility, then exercising that flexibility is usually cheaper than hand-coding a specific change.

Parameters are perhaps the best-known mechanism for introducing flexibility, and that is reminiscent of the abstract common services tactic. A parameterized function $f(a, b)$ is more general than the similar function $f(a)$ that assumes $b = 0$. When we bind the value of some parameters at a different phase in the life cycle than the one in which we defined the parameters, we are applying the defer binding tactic.

In general, the later in the life cycle we can bind values, the better. However, putting the mechanisms in place to facilitate that late binding tends to be more expensive—yet another tradeoff. And so the equation on page 118 comes into play. We want to bind as late as possible, as long as the mechanism that allows it is cost-effective.

Tactics to bind values at compile time or build time include these:

- Component replacement (for example, in a build script or makefile)
- Compile-time parameterization
- Aspects

Tactics to bind values at deployment time include this:

- Configuration-time binding

Tactics to bind values at startup or initialization time include this:

- Resource files

Tactics to bind values at runtime include these:

- Runtime registration
- Dynamic lookup (e.g., for services)
- Interpret parameters
- Startup time binding
- Name servers
- Plug-ins
- Publish-subscribe
- Shared repositories
- Polymorphism

Separating building a mechanism for modifiability from using the mechanism to make a modification admits the possibility of different stakeholders being involved—one stakeholder (usually a developer) to provide the mechanism and another stakeholder (an installer, for example, or a user) to exercise it later, possibly in a completely different life-cycle phase. Installing a mechanism so that someone else can make a change to the system without having to change any code is sometimes called *externalizing* the change.

7.3 A Design Checklist for Modifiability

Table 7.2 is a checklist to support the design and analysis process for modifiability.

TABLE 7.2 Checklist to Support the Design and Analysis Process for Modifiability

Category	Checklist
Allocation of Responsibilities	Determine which changes or categories of changes are likely to occur through consideration of changes in technical, legal, social, business, and customer forces. For each potential change or category of changes: ▪ Determine the responsibilities that would need to be added, modified, or deleted to make the change. ▪ Determine what responsibilities are impacted by the change. ▪ Determine an allocation of responsibilities to modules that places, as much as possible, responsibilities that will be changed (or impacted by the change) together in the same module, and places responsibilities that will be changed at different times in separate modules.
Coordination Model	Determine which functionality or quality attribute can change at runtime and how this affects coordination; for example, will the information being communicated change at runtime, or will the communication protocol change at runtime? If so, ensure that such changes affect a small number set of modules. Determine which devices, protocols, and communication paths used for coordination are likely to change. For those devices, protocols, and communication paths, ensure that the impact of changes will be limited to a small set of modules. For those elements for which modifiability is a concern, use a coordination model that reduces coupling such as publish-subscribe, defers bindings such as enterprise service bus, or restricts dependencies such as broadcast.
Data Model	Determine which changes (or categories of changes) to the data abstractions, their operations, or their properties are likely to occur. Also determine which changes or categories of changes to these data abstractions will involve their creation, initialization, persistence, manipulation, translation, or destruction. For each change or category of change, determine if the changes will be made by an end user, a system administrator, or a developer. For those changes to be made by an end user or system administrator, ensure that the necessary attributes are visible to that user and that the user has the correct privileges to modify the data, its operations, or its properties. For each potential change or category of change: ▪ Determine which data abstractions would need to be added, modified, or deleted to make the change. ▪ Determine whether there would be any changes to the creation, initialization, persistence, manipulation, translation, or destruction of these data abstractions. ▪ Determine which other data abstractions are impacted by the change. For these additional data abstractions, determine whether the impact would be on the operations, their properties, their creation, initialization, persistence, manipulation, translation, or destruction. ▪ Ensure an allocation of data abstractions that minimizes the number and severity of modifications to the abstractions by the potential changes. Design your data model so that items allocated to each element of the data model are likely to change together.

Category	Checklist
Mapping among Architectural Elements	Determine if it is desirable to change the way in which functionality is mapped to computational elements (e.g., processes, threads, processors) at runtime, compile time, design time, or build time. Determine the extent of modifications necessary to accommodate the addition, deletion, or modification of a function or a quality attribute. This might involve a determination of the following, for example: ▪ Execution dependencies ▪ Assignment of data to databases ▪ Assignment of runtime elements to processes, threads, or processors Ensure that such changes are performed with mechanisms that utilize deferred binding of mapping decisions.
Resource Management	Determine how the addition, deletion, or modification of a responsibility or quality attribute will affect resource usage. This involves, for example: ▪ Determining what changes might introduce new resources or remove old ones or affect existing resource usage ▪ Determining what resource limits will change and how Ensure that the resources after the modification are sufficient to meet the system requirements. Encapsulate all resource managers and ensure that the policies implemented by those resource managers are themselves encapsulated and bindings are deferred to the extent possible.
Binding Time	For each change or category of change: ▪ Determine the latest time at which the change will need to be made. ▪ Choose a defer-binding mechanism (see Section 7.2) that delivers the appropriate capability at the time chosen. ▪ Determine the cost of introducing the mechanism and the cost of making changes using the chosen mechanism. Use the equation on page 118 to assess your choice of mechanism. ▪ Do not introduce so many binding choices that change is impeded because the dependencies among the choices are complex and unknown.
Choice of Technology	Determine what modifications are made easier or harder by your technology choices. ▪ Will your technology choices help to make, test, and deploy modifications? ▪ How easy is it to modify your choice of technologies (in case some of these technologies change or become obsolete)? Choose your technologies to support the most likely modifications. For example, an enterprise service bus makes it easier to change how elements are connected but may introduce vendor lock-in.

7.4 Summary

Modifiability deals with change and the cost in time or money of making a change, including the extent to which this modification affects other functions or quality attributes.

Changes can be made by developers, installers, or end users, and these changes need to be prepared for. There is a cost of preparing for change as well as a cost of making a change. The modifiability tactics are designed to prepare for subsequent changes.

Tactics to reduce the cost of making a change include making modules smaller, increasing cohesion, and reducing coupling. Deferring binding will also reduce the cost of making a change.

Reducing coupling is a standard category of tactics that includes encapsulating, using an intermediary, restricting dependencies, co-locating related responsibilities, refactoring, and abstracting common services.

Increasing cohesion is another standard tactic that involves separating responsibilities that do not serve the same purpose.

Defer binding is a category of tactics that affect build time, load time, initialization time, or runtime.

7.5 For Further Reading

Serious students of software engineering should read two early papers about designing for modifiability. The first is Edsger Dijkstra's 1968 paper about the T.H.E. operating system [Dijkstra 68], which is the first paper that talks about designing systems to be layered, and the modifiability benefits it brings. The second is David Parnas's 1972 paper that introduced the concept of information hiding [Parnas 72]. Parnas prescribed defining modules not by their functionality but by their ability to internalize the effects of changes.

The tactics that we have presented in this chapter are a variant on those introduced by [Bachmann 07].

Additional tactics for modifiability within the avionics domain can be found in [EOSAN 07], published by the European Organization for the Safety of Air Navigation.

7.6 Discussion Questions

1. Modifiability comes in many flavors and is known by many names. Find one of the IEEE or ISO standards dealing with quality attributes and

compile a list of quality attributes that refer to some form of modifiability. Discuss the differences.

2. For each quality attribute that you discovered as a result of the previous question, write a modifiability scenario that expresses it.

3. In a certain metropolitan subway system, the ticket machines accept cash but do not give change. There is a separate machine that dispenses change but does not sell tickets. In an average station there are six or eight ticket machines for every change machine. What modifiability tactics do you see at work in this arrangement? What can you say about availability?

4. For the subway system in the previous question, describe the specific form of modifiability (using a modifiability scenario) that seems to be the aim of arranging the ticket and change machines as described.

5. A *wrapper* is a common aid to modifiability. A wrapper for a component is the only element allowed to use that component; every other piece of software uses the component's services by going through the wrapper. The wrapper transforms the data or control information for the component it wraps. For example, a component may expect input using English measures but find itself in a system in which all of the other components produce metric measures. A wrapper could be employed to translate. What modifiability tactics does a wrapper embody?

6. Once an intermediary has been introduced into an architecture, some modules may attempt to circumvent it, either inadvertently (because they are not aware of the intermediary) or intentionally (for performance, for convenience, or out of habit). Discuss some architectural means to prevent inadvertent circumvention of an intermediary.

7. In some projects, *deployability* is an important quality attribute that measures how easy it is to get a new version of the system into the hands of its users. This might mean a trip to your auto dealer or transmitting updates over the Internet. It also includes the time it takes to install the update once it arrives. In projects that measure deployability separately, should the cost of a modification stop when the new version is ready to ship? Justify your answer.

8. The abstract common services tactic is intended to reduce coupling, but it also might reduce cohesion. Discuss.

9. Identify particular change scenarios for an automatic teller machine. What modifications would you make to your automatic teller machine design to accommodate these changes?

This page intentionally left blank

8

Performance

An ounce of performance is worth pounds of promises.
—Mae West

It's about time.

Performance, that is: It's about time and the software system's ability to meet timing requirements. When events occur—interrupts, messages, requests from users or other systems, or clock events marking the passage of time—the system, or some element of the system, must respond to them in time. Characterizing the events that can occur (and when they can occur) and the system or element's time-based response to those events is the essence is discussing performance.

Web-based system events come in the form of requests from users (numbering in the tens or tens of millions) via their clients such as web browsers. In a control system for an internal combustion engine, events come from the operator's controls and the passage of time; the system must control both the firing of the ignition when a cylinder is in the correct position and the mixture of the fuel to maximize power and efficiency and minimize pollution.

For a web-based system, the desired response might be expressed as number of transactions that can be processed in a minute. For the engine control system, the response might be the allowable variation in the firing time. In each case, the pattern of events arriving and the pattern of responses can be characterized, and this characterization forms the language with which to construct performance scenarios.

For much of the history of software engineering, performance has been the driving factor in system architecture. As such, it has frequently compromised the achievement of all other qualities. As the price/performance ratio of hardware continues to plummet and the cost of developing software continues to rise, other qualities have emerged as important competitors to performance.

Nevertheless, all systems have performance requirements, even if they are not expressed. For example, a word processing tool may not have any explicit performance requirement, but no doubt everyone would agree that waiting an hour (or a minute, or a second) before seeing a typed character appear on the screen is unacceptable. Performance continues to be a fundamentally important quality attribute for all software.

Performance is often linked to scalability—that is, increasing your system's capacity for work, while still performing well. Technically, scalability is making your system easy to change in a particular way, and so is a kind of modifiability. In addition, we address scalability explicitly in Chapter 12.

8.1 Performance General Scenario

A performance scenario begins with an event arriving at the system. Responding correctly to the event requires resources (including time) to be consumed. While this is happening, the system may be simultaneously servicing other events.

Concurrency

Concurrency is one of the more important concepts that an architect must understand and one of the least-taught in computer science courses. Concurrency refers to operations occurring in parallel. For example, suppose there is a thread that executes the statements

```
x := 1;
x++;
```

and another thread that executes the same statements. What is the value of *x* after both threads have executed those statements? It could be either 2 or 3. I leave it to you to figure out how the value 3 could occur—or should I say I interleave it to you?

Concurrency occurs any time your system creates a new thread, because threads, by definition, are independent sequences of control. Multitasking on your system is supported by independent threads. Multiple users are simultaneously supported on your system through the use of threads. Concurrency also occurs any time your system is executing on more than one processor, whether the processors are packaged separately or as multi-core processors. In addition, you must consider concurrency when parallel algorithms, parallelizing infrastructures such as map-reduce, or NoSQL databases are used by your system, or you utilize one of a variety of concurrent scheduling algorithms. In other words, concurrency is a tool available to you in many ways.

Concurrency, when you have multiple CPUs or wait states that can exploit it, is a good thing. Allowing operations to occur in parallel improves performance, because delays introduced in one thread allow the processor to progress on another thread. But because of the interleaving phenomenon just described (referred to as a *race condition*), concurrency must also be carefully managed by the architect.

As the example shows, race conditions can occur when there are two threads of control and there is shared state. The management of concurrency frequently comes down to managing how state is shared. One technique for preventing race conditions is to use locks to enforce sequential access to state. Another technique is to partition the state based on the thread executing a portion of code. That is, if there are two instances of *x* in our example, *x* is not shared by the two threads and there will not be a race condition.

Race conditions are one of the hardest types of bugs to discover; the occurrence of the bug is sporadic and depends on (possibly minute) differences in timing. I once had a race condition in an operating system that I could not track down. I put a test in the code so that the next time the race condition occurred, a debugging process was triggered. It took over a year for the bug to recur so that the cause could be determined.

Do not let the difficulties associated with concurrency dissuade you from utilizing this very important technique. Just use it with the knowledge that you must carefully identify critical sections in your code and ensure that race conditions will not occur in those sections.

—LB

Events can arrive in predictable patterns or mathematical distributions, or be unpredictable. An arrival pattern for events is characterized as *periodic*, *stochastic*, or *sporadic*:

- Periodic events arrive predictably at regular time intervals. For instance, an event may arrive every 10 milliseconds. Periodic event arrival is most often seen in real-time systems.
- Stochastic arrival means that events arrive according to some probabilistic distribution.
- Sporadic events arrive according to a pattern that is neither periodic nor stochastic. Even these can be characterized, however, in certain circumstances. For example, we might know that at most 600 events will occur in a minute, or that there will be at least 200 milliseconds between the arrival of any two events. (This might describe a system in which events correspond to keyboard strokes from a human user.) These are helpful characterizations, even though we don’t know when any single event will arrive.

The response of the system to a stimulus can be measured by the following:

- *Latency.* The time between the arrival of the stimulus and the system’s response to it.

- *Deadlines in processing.* In the engine controller, for example, the fuel should ignite when the cylinder is in a particular position, thus introducing a processing deadline.
- The *throughput* of the system, usually given as the number of transactions the system can process in a unit of time.
- The *jitter* of the response—the allowable variation in latency.
- The *number of events not processed* because the system was too busy to respond.

From these considerations we can now describe the individual portions of a general scenario for performance:

- *Source of stimulus.* The stimuli arrive either from external (possibly multiple) or internal sources.
- *Stimulus*. The stimuli are the event arrivals. The arrival pattern can be periodic, stochastic, or sporadic, characterized by numeric parameters.
- *Artifact*. The artifact is the system or one or more of its components.
- *Environment*. The system can be in various operational modes, such as normal, emergency, peak load, or overload.
- *Response*. The system must process the arriving events. This may cause a change in the system environment (e.g., from normal to overload mode).
- *Response measure.* The response measures are the time it takes to process the arriving events (latency or a deadline), the variation in this time (jitter), the number of events that can be processed within a particular time interval (throughput), or a characterization of the events that cannot be processed (miss rate).

The general scenario for performance is summarized in Table 8.1.

Figure 8.1 gives an example concrete performance scenario: Users initiate transactions under normal operations. The system processes the transactions with an average latency of two seconds.

TABLE 8.1 Performance General Scenario

Portion of Scenario	Possible Values
Source	Internal or external to the system
Stimulus	Arrival of a periodic, sporadic, or stochastic event
Artifact	System or one or more components in the system
Environment	Operational mode: normal, emergency, peak load, overload
Response	Process events, change level of service
Response Measure	Latency, deadline, throughput, jitter, miss rate

8.2 Tactics for Performance

The goal of performance tactics is to generate a response to an event arriving at the system within some time-based constraint. The event can be single or a stream and is the trigger to perform computation. Performance tactics control the time within which a response is generated, as illustrated in Figure 8.2.

At any instant during the period after an event arrives but before the system's response t`o it is complete, either the system is working to respond to that event or the processing is blocked for some reason. This leads to the two basic contributors to the response time: processing time (when the system is working to respond) and blocked time (when the system is unable to respond).

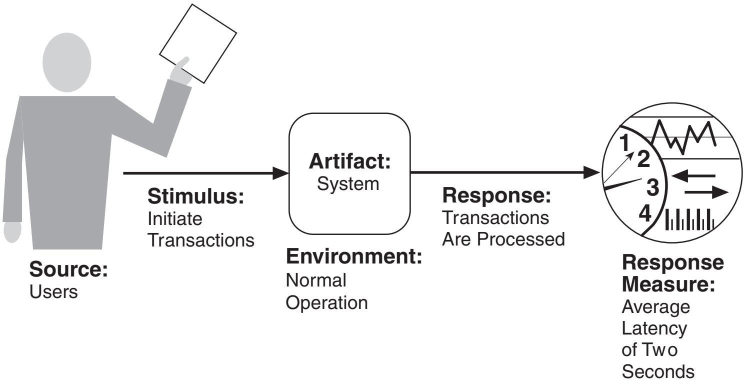


FIGURE 8.1 Sample concrete performance scenario

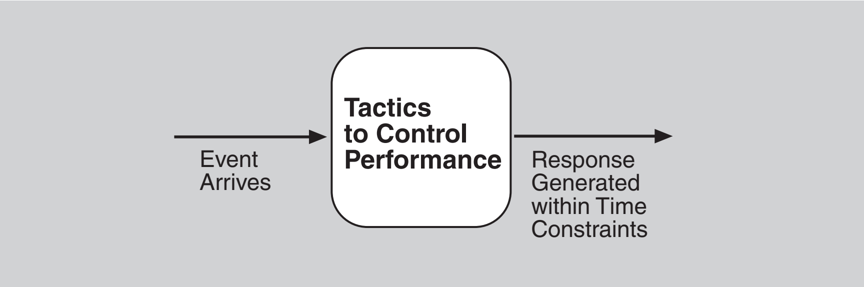


FIGURE 8.2 The goal of performance tactics

- *Processing time.* Processing consumes resources, which takes time. Events are handled by the execution of one or more components, whose time expended is a resource. Hardware resources include CPU, data stores, network communication bandwidth, and memory. Software resources include entities defined by the system under design. For example, buffers must be managed and access to critical sections[1] must be made sequential.

  For example, suppose a message is generated by one component. It might be placed on the network, after which it arrives at another component. It is then placed in a buffer; transformed in some fashion; processed according to some algorithm; transformed for output; placed in an output buffer; and sent onward to another component, another system, or some actor. Each of these steps consumes resources and time and contributes to the overall latency of the processing of that event.

  Different resources behave differently as their utilization approaches their capacity—that is, as they become saturated. For example, as a CPU becomes more heavily loaded, performance usually degrades fairly steadily. On the other hand, when you start to run out of memory, at some point the page swapping becomes overwhelming and performance crashes suddenly.

- *Blocked time.* A computation can be blocked because of contention for some needed resource, because the resource is unavailable, or because the computation depends on the result of other computations that are not yet available:

  - *Contention for resources.* Many resources can only be used by a single client at a time. This means that other clients must wait for access to those resources. Figure 8.2 shows events arriving at the system. These events may be in a single stream or in multiple streams. Multiple streams vying for the same resource or different events in the same stream vying for the same resource contribute to latency. The more contention for a resource, the more likelihood of latency being introduced.
  - *Availability of resources.* Even in the absence of contention, computation cannot proceed if a resource is unavailable. Unavailability may be caused by the resource being offline or by failure of the component or for some other reason. In any case, you must identify places where resource unavailability might cause a significant contribution to overall latency. Some of our tactics are intended to deal with this situation.
  - *Dependency on other computation.* A computation may have to wait because it must synchronize with the results of another computation or because it is waiting for the results of a computation that it initiated. If a component calls another component and must wait for that component to respond, the time can be significant if the called component is at the other end of a network (as opposed to co-located on the same processor).

[1] A critical section is a section of code in a multi-threaded system in which at most one thread may be active at any time.

With this background, we turn to our tactic categories. We can either reduce demand for resources or make the resources we have handle the demand more effectively:

- *Control resource demand.* This tactic operates on the demand side to produce smaller demand on the resources that will have to service the events.
- *Manage resources.* This tactic operates on the response side to make the resources at hand work more effectively in handling the demands put to them.

Control Resource Demand

One way to increase performance is to carefully manage the demand for resources. This can be done by reducing the number of events processed by enforcing a sampling rate, or by limiting the rate at which the system responds to events. In addition, there are a number of techniques for ensuring that the resources that you do have are applied judiciously:

- *Manage sampling rate*. If it is possible to reduce the sampling frequency at which a stream of environmental data is captured, then demand can be reduced, typically with some attendant loss of fidelity. This is common in signal processing systems where, for example, different codecs can be chosen with different sampling rates and data formats. This design choice is made to maintain predictable levels of latency; you must decide whether having a lower fidelity but consistent stream of data is preferable to losing packets of data.
- *Limit event response*. When discrete events arrive at the system (or element) too rapidly to be processed, then the events must be queued until they can be processed. Because these events are discrete, it is typically not desirable to "downsample" them. In such a case, you may choose to process events only up to a set maximum rate, thereby ensuring more predictable processing when the events are actually processed. This tactic could be triggered by a queue size or processor utilization measure exceeding some warning level. If you adopt this tactic and it is unacceptable to lose any events, then you must ensure that your queues are large enough to handle the worst case. If, on the other hand, you choose to drop events, then you need to choose a policy for handling this situation: Do you log the dropped events, or simply ignore them? Do you notify other systems, users, or administrators?
- *Prioritize events*. If not all events are equally important, you can impose a priority scheme that ranks events according to how important it is to service them. If there are not enough resources available to service them when they arise, low-priority events might be ignored. Ignoring events consumes minimal resources (including time), and thus increases performance compared to a system that services all events all the time. For example, a building

management system may raise a variety of alarms. Life-threatening alarms such as a fire alarm should be given higher priority than informational alarms such as a room is too cold.

- *Reduce overhead*. The use of intermediaries (so important for modifiability, as we saw in Chapter 7) increases the resources consumed in processing an event stream, and so removing them improves latency. This is a classic modifiability/performance tradeoff. Separation of concerns, another linchpin of modifiability, can also increase the processing overhead necessary to service an event if it leads to an event being serviced by a chain of components rather than a single component. The context switching and intercomponent communication costs add up, especially when the components are on different nodes on a network. A strategy for reducing computational overhead is to co-locate resources. Co-location may mean hosting cooperating components on the same processor to avoid the time delay of network communication; it may mean putting the resources in the same runtime software component to avoid even the expense of a subroutine call. A special case of reducing computational overhead is to perform a periodic cleanup of resources that have become inefficient. For example, hash tables and virtual memory maps may require recalculation and reinitialization. Another common strategy is to execute single-threaded servers (for simplicity and avoiding contention) and split workload across them.
- *Bound execution times*. Place a limit on how much execution time is used to respond to an event. For iterative, data-dependent algorithms, limiting the number of iterations is a method for bounding execution times. The cost is usually a less accurate computation. If you adopt this tactic, you will need to assess its effect on accuracy and see if the result is "good enough." This resource management tactic is frequently paired with the manage sampling rate tactic.
- *Increase resource efficiency.* Improving the algorithms used in critical areas will decrease latency.

Manage Resources

Even if the demand for resources is not controllable, the management of these resources can be. Sometimes one resource can be traded for another. For example, intermediate data may be kept in a cache or it may be regenerated depending on time and space resource availability. This tactic is usually applied to the processor but is also effective when applied to other resources such as a disk. Here are some resource management tactics:

- *Increase resources.* Faster processors, additional processors, additional memory, and faster networks all have the potential for reducing latency.

Cost is usually a consideration in the choice of resources, but increasing the resources is definitely a tactic to reduce latency and in many cases is the cheapest way to get immediate improvement.

- *Introduce concurrency*. If requests can be processed in parallel, the blocked time can be reduced. Concurrency can be introduced by processing different streams of events on different threads or by creating additional threads to process different sets of activities. Once concurrency has been introduced, scheduling policies can be used to achieve the goals you find desirable. Different scheduling policies may maximize fairness (all requests get equal time), throughput (shortest time to finish first), or other goals. (See the sidebar.)
- *Maintain multiple copies of computations*. Multiple servers in a client-server pattern are replicas of computation. The purpose of replicas is to reduce the contention that would occur if all computations took place on a single server. A *load balancer* is a piece of software that assigns new work to one of the available duplicate servers; criteria for assignment vary but can be as simple as round-robin or assigning the next request to the least busy server.
- *Maintain multiple copies of data. Caching* is a tactic that involves keeping copies of data (possibly one a subset of the other) on storage with different access speeds. The different access speeds may be inherent (memory versus secondary storage) or may be due to the necessity for network communication. *Data replication* involves keeping separate copies of the data to reduce the contention from multiple simultaneous accesses. Because the data being cached or replicated is usually a copy of existing data, keeping the copies consistent and synchronized becomes a responsibility that the system must assume. Another responsibility is to choose the data to be cached. Some caches operate by merely keeping copies of whatever was recently requested, but it is also possible to predict users' future requests based on patterns of behavior, and begin the calculations or prefetches necessary to comply with those requests before the user has made them.
- *Bound queue sizes*. This controls the maximum number of queued arrivals and consequently the resources used to process the arrivals. If you adopt this tactic, you need to adopt a policy for what happens when the queues overflow and decide if not responding to lost events is acceptable. This tactic is frequently paired with the limit event response tactic.
- *Schedule resources*. Whenever there is contention for a resource, the resource must be scheduled. Processors are scheduled, buffers are scheduled, and networks are scheduled. Your goal is to understand the characteristics of each resource's use and choose the scheduling strategy that is compatible with it. (See the sidebar.)

The tactics for performance are summarized in Figure 8.3.

Scheduling Policies

A *scheduling policy* conceptually has two parts: a priority assignment and dispatching. All scheduling policies assign priorities. In some cases the assignment is as simple as first-in/first-out (or FIFO). In other cases, it can be tied to the deadline of the request or its semantic importance. Competing criteria for scheduling include optimal resource usage, request importance, minimizing the number of resources used, minimizing latency, maximizing throughput, preventing starvation to ensure fairness, and so forth. You need to be aware of these possibly conflicting criteria and the effect that the chosen tactic has on meeting them.

A high-priority event stream can be dispatched only if the resource to which it is being assigned is available. Sometimes this depends on preempting the current user of the resource. Possible preemption options are as follows: can occur anytime, can occur only at specific preemption points, and executing processes cannot be preempted. Some common scheduling policies are these:

- *First-in/first-out.* FIFO queues treat all requests for resources as equals and satisfy them in turn. One possibility with a FIFO queue is that one request will be stuck behind another one that takes a long time to generate a response. As long as all of the requests are truly equal, this is not a problem, but if some requests are of higher priority than others, it is problematic.
- *Fixed-priority scheduling.* Fixed-priority scheduling assigns each source of resource requests a particular priority and assigns the resources in that priority order. This strategy ensures better service for higher priority requests. But it admits the possibility of a lower priority, but important, request taking an arbitrarily long time to be serviced, because it is stuck behind a series of higher priority requests. Three common prioritization strategies are these:
    - *Semantic importance.* Each stream is assigned a priority statically according to some domain characteristic of the task that generates it.
    - *Deadline monotonic.* Deadline monotonic. Deadline monotonic is a static priority assignment that assigns higher priority to streams with shorter deadlines. This scheduling policy is used when streams of different priorities with real-time deadlines are to be scheduled.
    - *Rate monotonic.* Rate monotonic is a static priority assignment for periodic streams that assigns higher priority to streams with shorter periods. This scheduling policy is a special case of deadline monotonic but is better known and more likely to be supported by the operating system.
- *Dynamic priority scheduling.* Strategies include these:
    - *Round-robin.* Round-robin is a scheduling strategy that orders the requests and then, at every assignment possibility, assigns the resource to the next request in that order. A special form of

round-robin is a cyclic executive, where assignment possibilities are at fixed time intervals.

- *Earliest-deadline-first.* Earliest-deadline-first. Earliest-deadline-first assigns priorities based on the pending requests with the earliest deadline.
- *Least-slack-first.* This strategy assigns the highest priority to the job having the least "slack time," which is the difference between the execution time remaining and the time to the job's deadline.

For a single processor and processes that are preemptible (that is, it is possible to suspend processing of one task in order to service a task whose deadline is drawing near), both the earliest-deadline and least-slack scheduling strategies are optimal. That is, if the set of processes can be scheduled so that all deadlines are met, then these strategies will be able to schedule that set successfully.

- *Static scheduling.* A cyclic executive schedule is a scheduling strategy where the preemption points and the sequence of assignment to the resource are determined offline. The runtime overhead of a scheduler is thereby obviated.

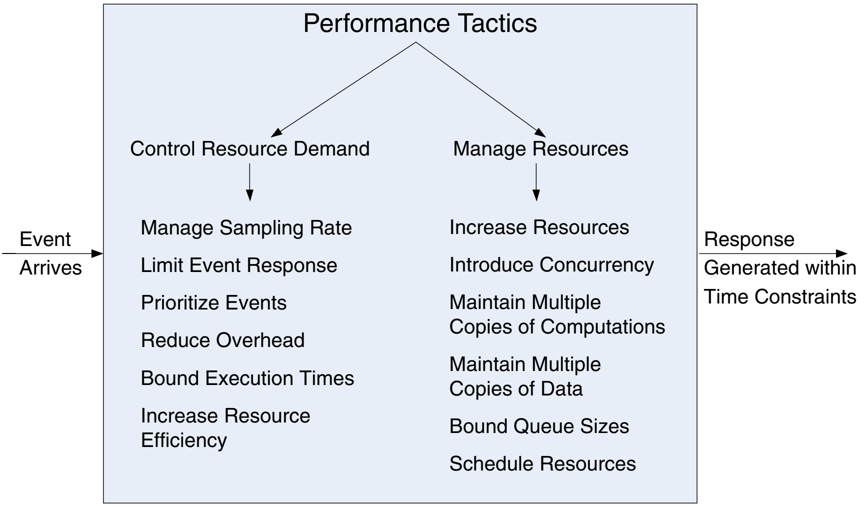


FIGURE 8.3 Performance tactics

Performance Tactics on the Road

Tactics are generic design principles. To exercise this point, think about the design of the systems of roads and highways where you live. Traffic engineers employ a bunch of design "tricks" to optimize the performance of these complex systems, where performance has a number of measures, such as throughput (how many cars per hour get from the suburbs to the football stadium), average-case latency (how long it takes, on average, to get from your house to downtown), and worst-case latency (how long does it take an emergency vehicle to get you to the hospital). What are these tricks? None other than our good old buddies, tactics.

Let's consider some examples:

- *Manage event rate.* Lights on highway entrance ramps let cars onto the highway only at set intervals, and cars must wait (queue) on the ramp for their turn.
- *Prioritize events.* Ambulances and police, with their lights and sirens going, have higher priority than ordinary citizens; some highways have high-occupancy vehicle (HOV) lanes, giving priority to vehicles with two or more occupants.
- *Maintain multiple copies.* Add traffic lanes to existing roads, or build parallel routes.

In addition, there are some tricks that users of the system can employ:

- *Increase resources.* Buy a Ferrari, for example. All other things being equal, the fastest car with a competent driver on an open road will get you to your destination more quickly.
- *Increase efficiency.* Find a new route that is quicker and/or shorter than your current route.
- *Reduce computational overhead.* You can drive closer to the car in front of you, or you can load more people into the same vehicle (that is, carpooling).

What is the point of this discussion? To paraphrase Gertrude Stein: performance is performance is performance. Engineers have been analyzing and optimizing systems for centuries, trying to improve their performance, and they have been employing the same design strategies to do so. So you should feel some comfort in knowing that when you try to improve the performance of your computer-based system, you are applying tactics that have been thoroughly "road tested."

—RK

8.3 A Design Checklist for Performance

Table 8.2 is a checklist to support the design and analysis process for performance.

TABLE 8.2 Checklist to Support the Design and Analysis Process for Performance

Category	**Checklist**
Allocation of Responsibilities	Determine the system's responsibilities that will involve heavy loading, have time-critical response requirements, are heavily used, or impact portions of the system where heavy loads or time-critical events occur. For those responsibilities, identify the processing requirements of each responsibility, and determine whether they may cause bottlenecks. Also, identify additional responsibilities to recognize and process requests appropriately, including ▪ Responsibilities that result from a thread of control crossing process or processor boundaries ▪ Responsibilities to manage the threads of control—allocation and deallocation of threads, maintaining thread pools, and so forth ▪ Responsibilities for scheduling shared resources or managing performance-related artifacts such as queues, buffers, and caches For the responsibilities and resources you identified, ensure that the required performance response can be met (perhaps by building a performance model to help in the evaluation).
Coordination Model	Determine the elements of the system that must coordinate with each other—directly or indirectly—and choose communication and coordination mechanisms that do the following: ▪ Support any introduced concurrency (for example, is it thread safe?), event prioritization, or scheduling strategy ▪ Ensure that the required performance response can be delivered ▪ Can capture periodic, stochastic, or sporadic event arrivals, as needed ▪ Have the appropriate properties of the communication mechanisms; for example, stateful, stateless, synchronous, asynchronous, guaranteed delivery, throughput, or latency
Data Model	Determine those portions of the data model that will be heavily loaded, have time-critical response requirements, are heavily used, or impact portions of the system where heavy loads or time-critical events occur. For those data abstractions, determine the following: ▪ Whether maintaining multiple copies of key data would benefit performance ▪ Whether partitioning data would benefit performance ▪ Whether reducing the processing requirements for the creation, initialization, persistence, manipulation, translation, or destruction of the enumerated data abstractions is possible ▪ Whether adding resources to reduce bottlenecks for the creation, initialization, persistence, manipulation, translation, or destruction of the enumerated data abstractions is feasible

continues

TABLE 8.2 Checklist to Support the Design and Analysis Process for Performance, *continued*

Category	Checklist
Mapping among Architectural Elements	Where heavy network loading will occur, determine whether co-locating some components will reduce loading and improve overall efficiency. Ensure that components with heavy computation requirements are assigned to processors with the most processing capacity. Determine where introducing concurrency (that is, allocating a piece of functionality to two or more copies of a component running simultaneously) is feasible and has a significant positive effect on performance. Determine whether the choice of threads of control and their associated responsibilities introduces bottlenecks.
Resource Management	Determine which resources in your system are critical for performance. For these resources, ensure that they will be monitored and managed under normal and overloaded system operation. For example: ▪ System elements that need to be aware of, and manage, time and other performance-critical resources ▪ Process/thread models ▪ Prioritization of resources and access to resources ▪ Scheduling and locking strategies ▪ Deploying additional resources on demand to meet increased loads
Binding Time	For each element that will be bound after compile time, determine the following: ▪ Time necessary to complete the binding ▪ Additional overhead introduced by using the late binding mechanism Ensure that these values do not pose unacceptable performance penalties on the system.
Choice of Technology	Will your choice of technology let you set and meet hard, real-time deadlines? Do you know its characteristics under load and its limits? Does your choice of technology give you the ability to set the following: ▪ Scheduling policy ▪ Priorities ▪ Policies for reducing demand ▪ Allocation of portions of the technology to processors ▪ Other performance-related parameters Does your choice of technology introduce excessive overhead for heavily used operations?

8.4 Summary

Performance is about the management of system resources in the face of particular types of demand to achieve acceptable timing behavior. Performance can be measured in terms of throughput and latency for both interactive and embedded real-time systems, although throughput is usually more important in interactive systems, and latency is more important in embedded systems.

Performance can be improved by reducing demand or by managing resources more appropriately. Reducing demand will have the side effect of reducing fidelity or refusing to service some requests. Managing resources more appropriately can be done through scheduling, replication, or just increasing the resources available.

8.5 For Further Reading

Performance has a rich body of literature. Here are some books we recommend:

- *Software Performance and Scalability: A Quantitative Approach* [Liu 09]. This books covers performance geared toward enterprise applications, with an emphasis on queuing theory and measurement.
- *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software* [Smith 01]. This book covers designing with performance in mind, with emphasis on building (and populating with real data) practical predictive performance models.
- *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems* [Douglass 99].
- *Real-Time Systems* [Liu 00].
- *Pattern-Oriented Software Architecture Volume 3: Patterns for Resource Management* [Kircher 03].

8.6 Discussion Questions

1. “Every system has real-time performance constraints.” Discuss. Or provide a counterexample.

2. Write a performance scenario that describes the average on-time flight arrival performance for an airline.

3. Write several performance scenarios for an automatic teller machine. Think about whether your major concern is worst-case latency, average-case latency, throughput, or some other response measure. How would you modify your automatic teller machine design to accommodate these scenarios?

4. Web-based systems often use *proxy servers*, which are the first element of the system to receive a request from a client (such as your browser). Proxy servers are able to serve up often-requested web pages, such as a company's home page, without bothering the real application servers that carry out transactions. There may be many proxy servers, and they are often located geographically close to large user communities, to decrease response time for routine requests. What performance tactics do you see at work here?

5. A fundamental difference between coordination mechanisms is whether interaction is synchronous or asynchronous. Discuss the advantages and disadvantages of each with respect to each of the performance responses: latency, deadline, throughput, jitter, miss rate, data loss, or any other required performance-related response you may be used to.

6. Find real-world (that is, nonsoftware) examples of applying each of the manage-resources tactics. For example, suppose you were managing a brick-and-mortar big-box retail store. How would you get people through the checkout lines faster using these tactics?

7. User interface frameworks typically are single-threaded. Why is this so and what are the performance implications of this single-threading?

9

Security

With Jungwoo Ryoo and Phil Laplante

> *Your personal identity isn't worth quite as much as it used to be—at least to thieves willing to swipe it. According to experts who monitor such markets, the value of stolen credit card data may range from $3 to as little as 40 cents. That's down tenfold from a decade ago—even though the cost to an individual who has a credit card stolen can soar into the hundreds of dollars.*
>
> —Forbes.com (Taylor Buley. "Hackonomics," Forbes.com, October 27, 2008, www.forbes.com/2008/10/25/credit-card-theft-tech-security-cz_tb1024theft.html)

Security is a measure of the system's ability to protect data and information from unauthorized access while still providing access to people and systems that are authorized. An action taken against a computer system with the intention of doing harm is called an attack and can take a number of forms. It may be an unauthorized attempt to access data or services or to modify data, or it may be intended to deny services to legitimate users.

The simplest approach to characterizing security has three characteristics: confidentiality, integrity, and availability (CIA):

1. *Confidentiality* is the property that data or services are protected from unauthorized access. For example, a hacker cannot access your income tax returns on a government computer.
2. *Integrity* is the property that data or services are not subject to unauthorized manipulation. For example, your grade has not been changed since your instructor assigned it.
3. *Availability* is the property that the system will be available for legitimate use. For example, a denial-of-service attack won't prevent you from ordering book from an online bookstore.

Other characteristics that are used to support CIA are these:

4. *Authentication* verifies the identities of the parties to a transaction and checks if they are truly who they claim to be. For example, when you get an email purporting to come from a bank, authentication guarantees that it actually comes from the bank.
5. *Nonrepudiation* guarantees that the sender of a message cannot later deny having sent the message, and that the recipient cannot deny having received the message. For example, you cannot deny ordering something from the Internet, or the merchant cannot disclaim getting your order.
6. *Authorization* grants a user the privileges to perform a task. For example, an online banking system authorizes a legitimate user to access his account.

We will use these characteristics in our general scenarios for security. Approaches to achieving security can be characterized as those that detect attacks, those that resist attacks, those that react to attacks, and those that recover from successful attacks. The objects that are being protected from attacks are data at rest, data in transit, and computational processes.

9.1 Security General Scenario

One technique that is used in the security domain is threat modeling. An “attack tree,” similar to a fault tree discussed in Chapter 5, is used by security engineers to determine possible threats. The root is a successful attack and the nodes are possible direct causes of that successful attack. Children nodes decompose the direct causes, and so forth. An attack is an attempt to break CIA, and the leaves of attack trees would be the stimulus in the scenario. The response to the attack is to preserve CIA or deter attackers through monitoring of their activities. From these considerations we can now describe the individual portions of a security general scenario. These are summarized in Table 9.1, and an example security scenario is given in Figure 9.1.

- *Source of stimulus*. The source of the attack may be either a human or another system. It may have been previously identified (either correctly or incorrectly) or may be currently unknown. A human attacker may be from outside the organization or from inside the organization.
- *Stimulus*. The stimulus is an attack. We characterize this as an unauthorized attempt to display data, change or delete data, access system services, change the system’s behavior, or reduce availability.
- *Artifact*. The target of the attack can be either the services of the system, the data within it, or the data produced or consumed by the system. Some attacks are made on particular components of the system known to be vulnerable.

- *Environment*. The attack can come when the system is either online or offline, either connected to or disconnected from a network, either behind a firewall or open to a network, fully operational, partially operational, or not operational.
- *Response.* The system should ensure that transactions are carried out in a fashion such that data or services are protected from unauthorized access; data or services are not being manipulated without authorization; parties to a transaction are identified with assurance; the parties to the transaction cannot repudiate their involvements; and the data, resources, and system services will be available for legitimate use.

  The system should also track activities within it by recording access or modification; attempts to access data, resources, or services; and notifying appropriate entities (people or systems) when an apparent attack is occurring.
- *Response measure*. Measures of a system's response include how much of a system is compromised when a particular component or data value is compromised, how much time passed before an attack was detected, how many attacks were resisted, how long it took to recover from a successful attack, and how much data was vulnerable to a particular attack.

Table 9.1 enumerates the elements of the general scenario, which characterize security, and Figure 9.1 shows a sample concrete scenario: A disgruntled employee from a remote location attempts to modify the pay rate table during normal operations. The system maintains an audit trail, and the correct data is restored within a day.

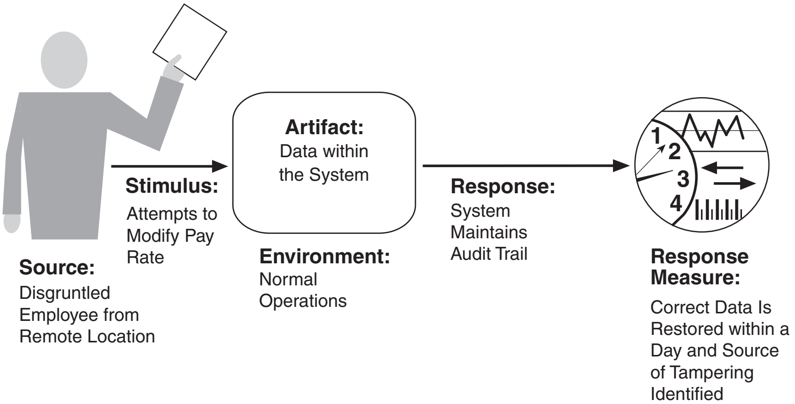

Stimulus:
Attempts to Modify Pay Rate
Artifact:
Data within the System
Environment:
Normal Operations
Response:
System Maintains Audit Trail
1
2
3
4
Response Measure:
Correct Data Is Restored within a Day and Source of Tampering Identified
Source:
Disgruntled Employee from Remote Location

FIGURE 9.1 Sample concrete security scenario

TABLE 9.1 Security General Scenario

Portion of Scenario	Possible Values
Source	Human or another system which may have been previously identified (either correctly or incorrectly) or may be currently unknown. A human attacker may be from outside the organization or from inside the organization.
Stimulus	Unauthorized attempt is made to display data, change or delete data, access system services, change the system's behavior, or reduce availability.
Artifact	System services, data within the system, a component or resources of the system, data produced or consumed by the system
Environment	The system is either online or offline; either connected to or disconnected from a network; either behind a firewall or open to a network; fully operational, partially operational, or not operational.
Response	Transactions are carried out in a fashion such that ▪ Data or services are protected from unauthorized access. ▪ Data or services are not being manipulated without authorization. ▪ Parties to a transaction are identified with assurance. ▪ The parties to the transaction cannot repudiate their involvements. ▪ The data, resources, and system services will be available for legitimate use. The system tracks activities within it by ▪ Recording access or modification ▪ Recording attempts to access data, resources, or services ▪ Notifying appropriate entities (people or systems) when an apparent attack is occurring
Response Measure	One or more of the following: ▪ How much of a system is compromised when a particular component or data value is compromised ▪ How much time passed before an attack was detected ▪ How many attacks were resisted ▪ How long does it take to recover from a successful attack ▪ How much data is vulnerable to a particular attack

9.2 Tactics for Security

One method for thinking about how to achieve security in a system is to think about physical security. Secure installations have limited access (e.g., by using security checkpoints), have means of detecting intruders (e.g., by requiring legitimate visitors to wear badges), have deterrence mechanisms such as armed guards, have reaction mechanisms such as automatic locking of doors, and have recovery mechanisms such as off-site backup. These lead to our four categories of tactics: detect, resist, react, and recover. Figure 9.2 shows these categories as the goal of security tactics.

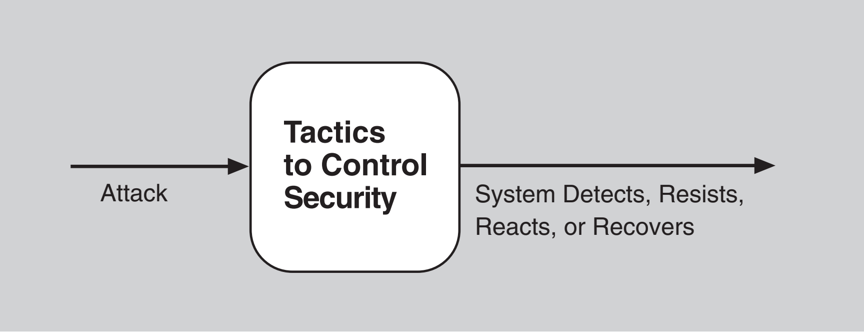

Attack
Tactics to Control Security
System Detects, Resists, Reacts, or Recovers

FIGURE 9.2 The goal of security tactics

Detect Attacks

The detect attacks category consists of four tactics: detect intrusion, detect service denial, verify message integrity, and detect message delay.

- *Detect intrusion* is the comparison of network traffic or service request patterns *within* a system to a set of signatures or known patterns of malicious behavior stored in a database. The signatures can be based on protocol, TCP flags, payload sizes, applications, source or destination address, or port number.
- *Detect service denial* is the comparison of the pattern or signature of network traffic *coming into* a system to historic profiles of known denial-of-service attacks.
- *Verify message integrity.* This tactic employs techniques such as checksums or hash values to verify the integrity of messages, resource files, deployment files, and configuration files. A checksum is a validation mechanism wherein the system maintains redundant information for configuration files and messages, and uses this redundant information to verify the configuration file or message when it is used. A hash value is a unique string generated by a hashing function whose input could be configuration files or messages. Even a slight change in the original files or messages results in a significant change in the hash value.
- *Detect message delay* is intended to detect potential man-in-the-middle attacks, where a malicious party is intercepting (and possibly modifying) messages. By checking the time that it takes to deliver a message, it is possible to detect suspicious timing behavior, where the time it takes to deliver a message is highly variable.

Resist Attacks

There are a number of well-known means of resisting an attack:

- *Identify actors*. Identifying "actors" is really about identifying the source of any external input to the system. Users are typically identified through user IDs. Other systems may be "identified" through access codes, IP addresses, protocols, ports, and so on.
- *Authenticate actors*. Authentication means ensuring that an actor (a user or a remote computer) is actually who or what it purports to be. Passwords, one-time passwords, digital certificates, and biometric identification provide a means for authentication.
- *Authorize actors*. Authorization means ensuring that an authenticated actor has the rights to access and modify either data or services. This mechanism is usually enabled by providing some access control mechanisms within a system. Access control can be by an actor or by an actor class. Classes of actors can be defined by actor groups, by actor roles, or by lists of individuals.
- *Limit access*. Limiting access involves controlling what and who may access which parts of a system. This may include limiting access to resources such as processors, memory, and network connections, which may be achieved by using process management, memory protection, blocking a host, closing a port, or rejecting a protocol. For example, a firewall is a single point of access to an organization's intranet. A demilitarized zone (DMZ) is a subnet between the Internet and an intranet, protected by two firewalls: one facing the Internet and the other the intranet. A DMZ is used when an organization wants to let external users access services that should be publicly available outside the intranet. This way the number of open ports in the internal firewall can be minimized. This tactic also limits access for actors (by identifying, authenticating, and authorizing them).
- *Limit exposure.* Limiting exposure refers to ultimately and indirectly reducing the probability of a successful attack, or restricting the amount of potential damage. This can be achieved by concealing facts about a system to be protected ("security by obscurity") or by dividing and distributing critical resources so that the exploitation of a single weakness cannot fully compromise any resource ("don't put all your eggs in one basket"). For example, a design decision to hide how many entry points a system has is a way of limiting exposure. A decision to distribute servers amongst several geographically dispersed data centers is also a way of limiting exposure.
- *Encrypt data*. Data should be protected from unauthorized access. Confidentiality is usually achieved by applying some form of encryption to data and to communication. Encryption provides extra protection to persistently maintained data beyond that available from authorization. Communication links, on the other hand, may not have authorization controls. In such cases, encryption is the only protection for passing data over publicly accessible communication links. The link can be implemented by a virtual private network (VPN) or by a Secure Sockets Layer (SSL) for

a web-based link. Encryption can be symmetric (both parties use the same key) or asymmetric (public and private keys).

- *Separate entities*. Separating different entities within the system can be done through physical separation on different servers that are attached to different networks; the use of virtual machines (see Chapter 26 for a discussion of virtual machines); or an "air gap," that is, by having no connection between different portions of a system. Finally, sensitive data is frequently separated from nonsensitive data to reduce the attack possibilities from those who have access to nonsensitive data.
- *Change default settings*. Many systems have default settings assigned when the system is delivered. Forcing the user to change those settings will prevent attackers from gaining access to the system through settings that are, generally, publicly available.

React to Attacks

Several tactics are intended to respond to a potential attack:

- *Revoke access*. If the system or a system administrator believes that an attack is underway, then access can be severely limited to sensitive resources, even for normally legitimate users and uses. For example, if your desktop has been compromised by a virus, your access to certain resources may be limited until the virus is removed from your system.
- *Lock computer*. Repeated failed login attempts may indicate a potential attack. Many systems limit access from a particular computer if there are repeated failed attempts to access an account from that computer. Legitimate users may make mistakes in attempting to log in. Therefore, the limited access may only be for a certain time period.
- *Inform actors*. Ongoing attacks may require action by operators, other personnel, or cooperating systems. Such personnel or systems—the set of relevant actors—must be notified when the system has detected an attack.

Recover from Attacks

Once a system has detected and attempted to resist an attack, it needs to recover. Part of recovery is restoration of services. For example, additional servers or network connections may be kept in reserve for such a purpose. Since a successful attack can be considered a kind of failure, the set of availability tactics (from Chapter 5) that deal with recovering from a failure can be brought to bear for this aspect of security as well.

In addition to the availability tactics that permit restoration of services, we need to maintain an audit trail. We audit—that is, keep a record of user and system actions and their effects—to help trace the actions of, and to identify, an attacker. We may analyze audit trails to attempt to prosecute attackers, or to create better defenses in the future.

The set of security tactics is shown in Figure 9.3.

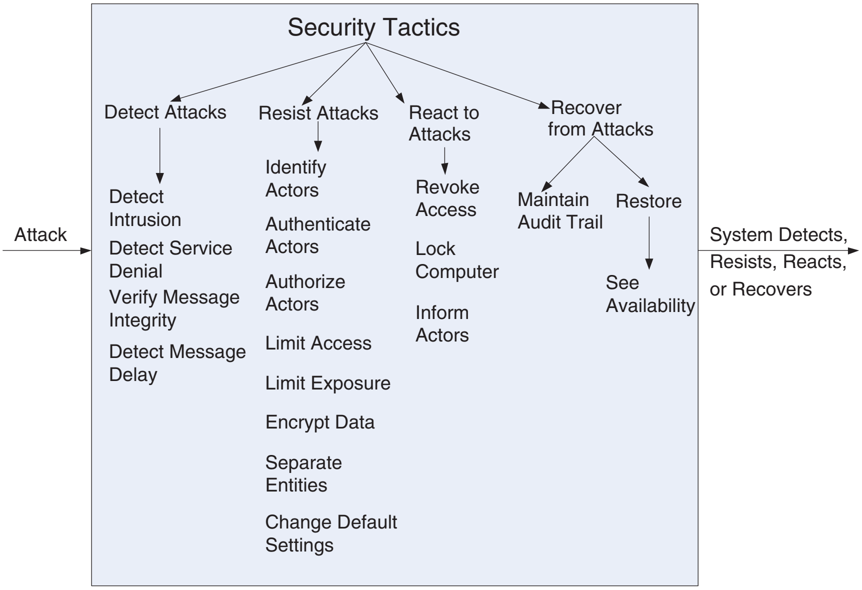


FIGURE 9.3 Security tactics

9.3 A Design Checklist for Security

Table 9.2 is a checklist to support the design and analysis process for security.

TABLE 9.2 Checklist to Support the Design and Analysis Process for Security

Category	Checklist
Allocation of Responsibilities	Determine which system responsibilities need to be secure. For each of these responsibilities, ensure that additional responsibilities have been allocated to do the following: ▪ Identify the actor ▪ Authenticate the actor ▪ Authorize actors ▪ Grant or deny access to data or services ▪ Record attempts to access or modify data or services ▪ Encrypt data ▪ Recognize reduced availability for resources or services and inform appropriate personnel and restrict access ▪ Recover from an attack ▪ Verify checksums and hash values

Category	Checklist
Coordination Model	Determine mechanisms required to communicate and coordinate with other systems or individuals. For these communications, ensure that mechanisms for authenticating and authorizing the actor or system, and encrypting data for transmission across the connection, are in place. Ensure also that mechanisms exist for monitoring and recognizing unexpectedly high demands for resources or services as well as mechanisms for restricting or terminating the connection.
Data Model	Determine the sensitivity of different data fields. For each data abstraction: ▪ Ensure that data of different sensitivity is separated. ▪ Ensure that data of different sensitivity has different access rights and that access rights are checked prior to access. ▪ Ensure that access to sensitive data is logged and that the log file is suitably protected. ▪ Ensure that data is suitably encrypted and that keys are separated from the encrypted data. ▪ Ensure that data can be restored if it is inappropriately modified.
Mapping among Architectural Elements	Determine how alternative mappings of architectural elements that are under consideration may change how an individual or system may read, write, or modify data; access system services or resources; or reduce availability to system services or resources. Determine how alternative mappings may affect the recording of access to data, services or resources and the recognition of unexpectedly high demands for resources. For each such mapping, ensure that there are responsibilities to do the following: ▪ Identify an actor ▪ Authenticate an actor ▪ Authorize actors ▪ Grant or deny access to data or services ▪ Record attempts to access or modify data or services ▪ Encrypt data ▪ Recognize reduced availability for resources or services, inform appropriate personnel, and restrict access ▪ Recover from an attack
Resource Management	Determine the system resources required to identify and monitor a system or an individual who is internal or external, authorized or not authorized, with access to specific resources or all resources. Determine the resources required to authenticate the actor, grant or deny access to data or resources, notify appropriate entities (people or systems), record attempts to access data or resources, encrypt data, recognize inexplicably high demand for resources, inform users or systems, and restrict access. For these resources consider whether an external entity can access a critical resource or exhaust a critical resource; how to monitor the resource; how to manage resource utilization; how to log resource utilization; and ensure that there are sufficient resources to perform the necessary security operations. Ensure that a contaminated element can be prevented from contaminating other elements. Ensure that shared resources are not used for passing sensitive data from an actor with access rights to that data to an actor without access rights to that data.

continues

TABLE 9.2 Checklist to Support the Design and Analysis Process for Security, *continued*

Category	Checklist
Binding Time	Determine cases where an instance of a late-bound component may be untrusted. For such cases ensure that late-bound components can be qualified; that is, if ownership certificates for late-bound components are required, there are appropriate mechanisms to manage and validate them; that access to late-bound data and services can be managed; that access by late-bound components to data and services can be blocked; that mechanisms to record the access, modification, and attempts to access data or services by late-bound components are in place; and that system data is encrypted where the keys are intentionally withheld for late-bound components
Choice of Technology	Determine what technologies are available to help user authentication, data access rights, resource protection, and data encryption. Ensure that your chosen technologies support the tactics relevant for your security needs.

9.4 Summary

Attacks against a system can be characterized as attacks against the confidentiality, integrity, or availability of a system or its data. Confidentiality means keeping data away from those who should not have access while granting access to those who should. Integrity means that there are no unauthorized modifications to or deletion of data, and availability means that the system is accessible to those who are entitled to use it.

The emphasis of distinguishing various classes of actors in the characterization leads to many of the tactics used to achieve security. Identifying, authenticating, and authorizing actors are tactics intended to determine which users or systems are entitled to what kind of access to a system.

An assumption is made that no security tactic is foolproof and that systems will be compromised. Hence, tactics exist to detect an attack, limit the spread of any attack, and to react and recover from an attack.

Recovering from an attack involves many of the same tactics as availability and, in general, involves returning the system to a consistent state prior to any attack.

9.5 For Further Reading

The architectural tactics that we have described in this chapter are only one aspect of making a system secure. Other aspects are these:

- *Coding. Secure Coding in C and C++* [Seacord 05] describes how to code securely. The Common Weakness Enumeration [CWE 12] is a list of the most common vulnerabilities discovered in systems.
- *Organizational processes.* Organizations must have processes that provide for responsibility for various aspects of security, including ensuring that systems are patched to put into place the latest protections. The National Institute of Standards and Technology (NIST) provides an enumeration of organizational processes [NIST 09]. [Cappelli 12] discusses insider threats.
- *Technical processes.* Microsoft has a life-cycle development process (The Secure Development Life Cycle) that includes modeling of threats. Four training classes are publicly available. www.microsoft.com/download/en/details.aspx?id=16420

NIST has several volumes that give definitions of security terms [NIST 04], categories of security controls [NIST 06], and an enumeration of security controls that an organization could employ [NIST 09]. A security control could be a tactic, but it could also be organizational, coding-related, or a technical process.

The attack surface of a system is the code that can be run by unauthorized users. A discussion of how to minimize the attack surface for a system can be found at [Howard 04].

Encryption and certificates of various types and strengths are commonly used to resist certain types of attacks. Encryption algorithms are particularly difficult to code correctly. A document produced by NIST [NIST 02] gives requirements for these algorithms.

Good books on engineering systems for security have been written by Ross Anderson [Anderson 08] and Bruce Schneier [Schneier 08].

Different domains have different specific sets of practices. The Payment Card Industry (PCI) has a set of standards intended for those involved in credit card processing (www.pcisecuritystandards.org). There is also a set of recommendations for securing various portions of the electric grid (www.smartgridipedia.org/index.php/ASAP-SG).

Data on the various sources of data breaches can be found in the Verizon 2012 Data Breach Investigations Report [Verizon 12].

John Viega has written several books about secure software development in various environments. See, for example, [Viega 01].

9.6 Discussion Questions

1. Write a set of concrete scenarios for security for an automatic teller machine. How would you modify your design for the automatic teller machine to satisfy these scenarios?

2. One of the most sophisticated attacks on record was carried out by a virus known as Stuxnet. Stuxnet first appeared in 2009 but became widely known in 2011 when it was revealed that it had apparently severely damaged or incapacitated the high-speed centrifuges involved in Iran's uranium enrichment program. Read about Stuxnet and see if you can devise a defense strategy against it based on the tactics in this chapter.

3. Some say that inserting security awareness into the software development life cycle is at least as important as designing software with security countermeasures. What are some examples of software development processes that can lead to more-secure systems?

4. Security and usability are often seen to be at odds with each other. Security often imposes procedures and processes that seem like needless overhead to the casual user. But some say that security and usability go (or should go) hand in hand and argue that making the system easy to use securely is the best way to promote security to the user. Discuss.

5. List some examples of critical resources for security that might become exhausted.

6. List an example of a mapping of architectural elements that has strong security implications. Hint: think of where data is stored.

7. Which of the tactics in our list will protect against an insider threat? Can you think of any that should be added?

8. In the United States, Facebook can account for more than 5 percent of all Internet traffic in a given week. How would you recognize a denial-of-service attack on Facebook.com?

9. The public disclosure of vulnerabilities in production systems is a matter of controversy. Discuss why this is so and the pros and cons of public disclosure of vulnerabilities.

10

Testability

Testing leads to failure, and failure
leads to understanding
—Burt Rutan

Industry estimates indicate that between 30 and 50 percent (or in some cases, even more) of the cost of developing well-engineered systems is taken up by testing. If the software architect can reduce this cost, the payoff is large.

Software testability refers to the ease with which software can be made to demonstrate its faults through (typically execution-based) testing. Specifically, testability refers to the probability, assuming that the software has at least one fault, that it will fail on its next test execution. Intuitively, a system is testable if it "gives up" its faults easily. If a fault is present in a system, then we want it to fail during testing as quickly as possible. Of course, calculating this probability is not easy and, as you will see when we discuss response measures for testability, other measures will be used.

Figure 10.1 shows a model of testing in which a program processes input and produces output. An oracle is an agent (human or mechanical) that decides whether the output is correct or not by comparing the output to the program's specification. Output is not just the functionally produced value, but it also can include derived measures of quality attributes such as how long it took to produce the output. Figure 10.1 also shows that the program's internal state can also be shown to the oracle, and an oracle can decide whether that is correct or not—that is, it can detect whether the program has entered an erroneous state and render a judgment as to the correctness of the program.

Setting and examining a program's internal state is an aspect of testing that will figure prominently in our tactics for testability.

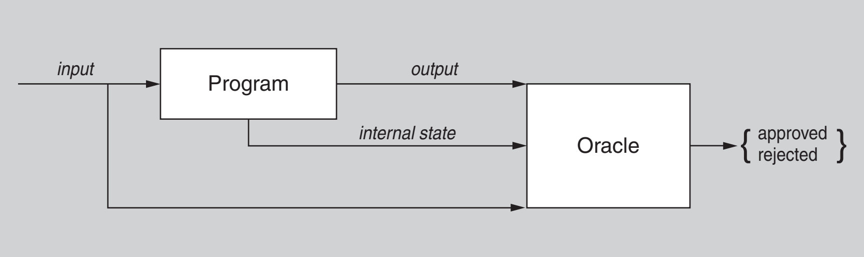


FIGURE 10.1 A model of testing

For a system to be properly testable, it must be possible to control each component's inputs (and possibly manipulate its internal state) and then to observe its outputs (and possibly its internal state, either after or on the way to computing the outputs). Frequently this control and observation is done through the use of a test harness, which is specialized software (or in some cases, hardware) designed to exercise the software under test. Test harnesses come in various forms, such as a record-and-playback capability for data sent across various interfaces, or a simulator for an external environment in which a piece of embedded software is tested, or even during production (see sidebar). The test harness can provide assistance in executing the test procedures and recording the output. A test harness can be a substantial piece of software in its own right, with its own architecture, stakeholders, and quality attribute requirements.

Testing is carried out by various developers, users, or quality assurance personnel. Portions of the system or the entire system may be tested. The response measures for testability deal with how effective the tests are in discovering faults and how long it takes to perform the tests to some desired level of coverage. Test cases can be written by the developers, the testing group, or the customer. The test cases can be a portion of acceptance testing or can drive the development as they do in certain types of Agile methodologies.

Netflix's Simian Army

Netflix distributes movies and television shows both via DVD and via streaming video. Their streaming video service has been extremely successful. In May 2011 Netflix streaming video accounted for 24 percent of the Internet traffic in North America. Naturally, high availability is important to Netflix.

Netflix hosts their computer services in the Amazon EC2 cloud, and they utilize what they call a "Simian Army" as a portion of their testing process. They began with a Chaos Monkey, which randomly kills processes in the running system. This allows the monitoring of the effect of failed processes and gives the ability to ensure that the system does not fail or suffer serious degradation as a result of a process failure.

Recently, the Chaos Monkey got some friends to assist in the testing. Currently, the Netflix Simian Army includes these:

- The Latency Monkey induces artificial delays in the client-server communication layer to simulate service degradation and measures if upstream services respond appropriately.
- The Conformity Monkey finds instances that don't adhere to best practices and shuts them down. For example, if an instance does not belong to an auto-scaling group, it will not appropriately scale when demand goes up.
- The Doctor Monkey taps into health checks that run on each instance as well as monitors other external signs of health (e.g., CPU load) to detect unhealthy instances.
- The Janitor Monkey ensures that the Netflix cloud environment is running free of clutter and waste. It searches for unused resources and disposes of them.
- The Security Monkey is an extension of Conformity Monkey. It finds security violations or vulnerabilities, such as improperly configured security groups, and terminates the offending instances. It also ensures that all the SSL and digital rights management (DRM) certificates are valid and are not coming up for renewal.
- The 10-18 Monkey (localization-internationalization) detects configuration and runtime problems in instances serving customers in multiple geographic regions, using different languages and character sets. The name 10-18 comes from *L10n-i18n*, a sort of shorthand for the words *localization* and *internationalization*.

Some of the members of the Simian Army use fault injection to place faults into the running system in a controlled and monitored fashion. Other members monitor various specialized aspects of the system and its environment. Both of these techniques have broader applicability than just Netflix.

Not all faults are equal in terms of severity. More emphasis should be placed on finding the most severe faults than on finding other faults. The Simian Army reflects a determination by Netflix that the faults they look for are the most serious in terms of their impact.

This strategy illustrates that some systems are too complex and adaptive to be tested fully, because some of their behaviors are emergent. An aspect of testing in that arena is logging of operational data produced by the system, so that when failures occur, the logged data can be analyzed in the lab to try to reproduce the faults. Architecturally this can require mechanisms to access and log certain system state. The Simian Army is one way to discover and log behavior in systems of this ilk.

—LB

Testing of code is a special case of validation, which is making sure that an engineered artifact meets the needs of its stakeholders or is suitable for use. In Chapter 21 we will discuss architectural design reviews. This is another kind of validation, where the artifact being tested is the architecture. In this chapter we are concerned only with the testability of a running system and of its source code.

10.1 Testability General Scenario

We can now describe the general scenario for testability.

- *Source of stimulus.* The testing is performed by unit testers, integration testers, or system testers (on the developing organization side), or acceptance testers and end users (on the customer side). The source could be human or an automated tester.
- *Stimulus*. A set of tests is executed due to the completion of a coding increment such as a class layer or service, the completed integration of a subsystem, the complete implementation of the whole system, or the delivery of the system to the customer.
- *Artifact*. A unit of code (corresponding to a module in the architecture), a subsystem, or the whole system is the artifact being tested.
- *Environment*. The test can happen at development time, at compile time, at deployment time, or while the system is running (perhaps in routine use). The environment can also include the test harness or test environments in use.
- *Response*. The system can be controlled to perform the desired tests and the results from the test can be observed.
- *Response measure*. Response measures are aimed at representing how easily a system under test "gives up" its faults. Measures might include the effort involved in finding a fault or a particular class of faults, the effort required to test a given percentage of statements, the length of the longest test chain (a measure of the difficulty of performing the tests), measures of effort to perform the tests, measures of effort to actually find faults, estimates of the probability of finding additional faults, and the length of time or amount of effort to prepare the test environment.

  Maybe one measure is the ease at which the system can be brought into a specific state. In addition, measures of the reduction in risk of the remaining errors in the system can be used. Not all faults are equal in terms of their possible impact. Measures of risk reduction attempt to rate the severity of faults found (or to be found).

Figure 10.2 shows a concrete scenario for testability. The unit tester completes a code unit during development and performs a test sequence whose results are captured and that gives 85 percent path coverage within three hours of testing.

Table 10.1 enumerates the elements of the general scenario that characterize testability.

TABLE 10.1 Testability General Scenario

Portion of Scenario	Possible Values
Source	Unit testers, integration testers, system testers, acceptance testers, end users, either running tests manually or using automated testing tools
Stimulus	A set of tests is executed due to the completion of a coding increment such as a class layer or service, the completed integration of a subsystem, the complete implementation of the whole system, or the delivery of the system to the customer.
Environment	Design time, development time, compile time, integration time, deployment time, run time
Artifacts	The portion of the system being tested
Response	One or more of the following: execute test suite and capture results, capture activity that resulted in the fault, control and monitor the state of the system
Response Measure	One or more of the following: effort to find a fault or class of faults, effort to achieve a given percentage of state space coverage, probability of fault being revealed by the next test, time to perform tests, effort to detect faults, length of longest dependency chain in test, length of time to prepare test environment, reduction in risk exposure (size(loss) × prob(loss))

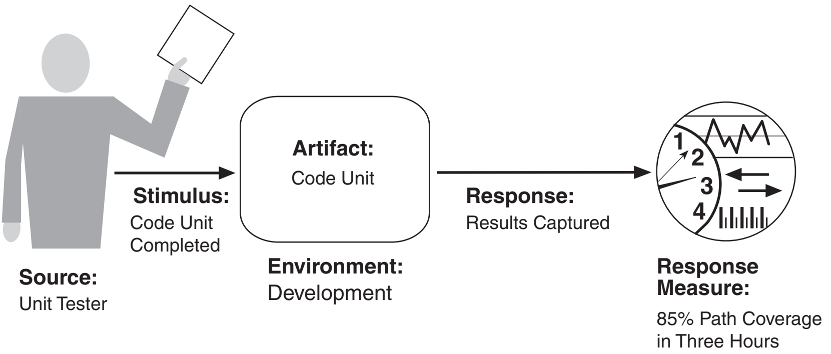


FIGURE 10. 2 Sample concrete testability scenario

10.2 Tactics for Testability

The goal of tactics for testability is to allow for easier testing when an increment of software development is completed. Figure 10.3 displays the use of tactics for testability. Architectural techniques for enhancing the software testability have not received as much attention as more mature quality attribute disciplines such as modifiability, performance, and availability, but as we stated before, anything the architect can do to reduce the high cost of testing will yield a significant benefit.

There are two categories of tactics for testability. The first category deals with adding controllability and observability to the system. The second deals with limiting complexity in the system's design.

Control and Observe System State

Control and observation are so central to testability that some authors even define testability in those terms. The two go hand-in-hand; it makes no sense to control something if you can't observe what happens when you do. The simplest form of control and observation is to provide a software component with a set of inputs, let it do its work, and then observe its outputs. However, the control and observe system state category of testability tactics provides insight into software that goes beyond its inputs and outputs. These tactics cause a component to maintain some sort of state information, allow testers to assign a value to that state information, and/or make that information accessible to testers on demand. The state information might be an operating state, the value of some key variable, performance load, intermediate process steps, or anything else useful to re-creating component behavior. Specific tactics include the following:

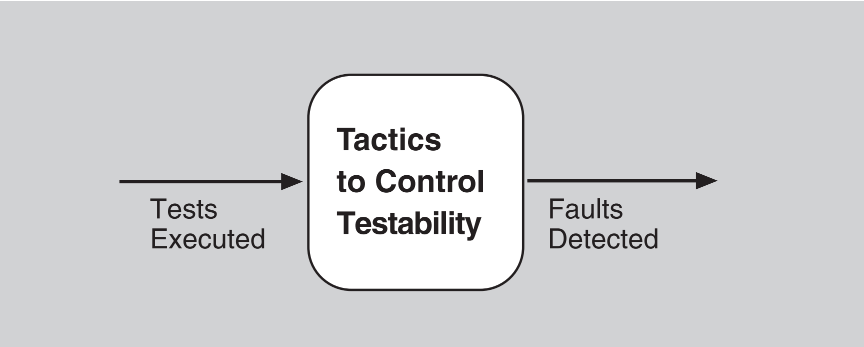


FIGURE 10.3 The goal of testability tactics

- *Specialized interfaces.* Having specialized testing interfaces allows you to control or capture variable values for a component either through a test harness or through normal execution. Examples of specialized test routines include these:

  - A *set* and *get* method for important variables, modes, or attributes (methods that might otherwise not be available except for testing purposes)
  - A *report* method that returns the full state of the object
  - A *reset* method to set the internal state (for example, all the attributes of a class) to a specified internal state
  - A method to turn on verbose output, various levels of event logging, performance instrumentation, or resource monitoring

  Specialized testing interfaces and methods should be clearly identified or kept separate from the access methods and interfaces for required functionality, so that they can be removed if needed. (However, in performance-critical and some safety-critical systems, it is problematic to field different code than that which was tested. If you remove the test code, how will you know the code you field has the same behavior, particularly the same timing behavior, as the code you tested? For other kinds of systems, however, this strategy is effective.)

- *Record/playback*. The state that caused a fault is often difficult to re-create. Recording the state when it crosses an interface allows that state to be used to "play the system back" and to re-create the fault. Record/playback refers to both capturing information crossing an interface and using it as input for further testing.
- *Localize state storage*. To start a system, subsystem, or module in an arbitrary state for a test, it is most convenient if that state is stored in a single place. By contrast, if the state is buried or distributed, this becomes difficult if not impossible. The state can be fine-grained, even bit-level, or coarse-grained to represent broad abstractions or overall operational modes. The choice of granularity depends on how the states will be used in testing. A convenient way to "externalize" state storage (that is, to make it able to be manipulated through interface features) is to use a state machine (or state machine object) as the mechanism to track and report current state.
- *Abstract data sources.* Similar to controlling a program's state, easily controlling its input data makes it easier to test. Abstracting the interfaces lets you substitute test data more easily. For example, if you have a database of customer transactions, you could design your architecture so that it is easy to point your test system at other test databases, or possibly even to files of test data instead, without having to change your functional code.
- *Sandbox*. "Sandboxing" refers to isolating an instance of the system from the real world to enable experimentation that is unconstrained by the worry

about having to undo the consequences of the experiment. Testing is helped by the ability to operate the system in such a way that it has no permanent consequences, or so that any consequences can be rolled back. This can be used for scenario analysis, training, and simulation. (The Spring framework, which is quite popular in the Java community, comes with a set of test utilities that support this. Tests are run as a "transaction," which is rolled back at the end.)

A common form of sandboxing is to virtualize resources. Testing a system often involves interacting with resources whose behavior is outside the control of the system. Using a sandbox, you can build a version of the resource whose behavior is under your control. For example, the system clock's behavior is typically not under our control—it increments one second each second—which means that if we want to make the system think it's midnight on the day when all of the data structures are supposed to overflow, we need a way to do that, because waiting around is a poor choice. By having the capability to abstract system time from clock time, we can allow the system (or components) to run at faster than wall-clock time, and to allow the system (or components) to be tested at critical time boundaries (such as the next shift on or off Daylight Savings Time). Similar virtualizations could be done for other resources, such as memory, battery, network, and so on. Stubs, mocks, and dependency injection are simple but effective forms of virtualization.

- *Executable assertions*. Using this tactic, assertions are (usually) hand-coded and placed at desired locations to indicate when and where a program is in a faulty state. The assertions are often designed to check that data values satisfy specified constraints. Assertions are defined in terms of specific data declarations, and they must be placed where the data values are referenced or modified. Assertions can be expressed as pre- and post-conditions for each method and also as class-level invariants. This results in increasing observability, when an assertion is flagged as having failed. Assertions systematically inserted where data values change can be seen as a manual way to produce an "extended" type. Essentially, the user is annotating a type with additional checking code. Any time an object of that type is modified, the checking code is automatically executed, and warnings are generated if any conditions are violated. To the extent that the assertions cover the test cases, they effectively embed the test oracle in the code—assuming the assertions are correct and correctly coded.

All of these tactics add capability or abstraction to the software that (were we not interested in testing) otherwise would not be there. They can be seen as replacing bare-bones, get-the-job-done software with more elaborate software that has bells and whistles for testing. There are a number of techniques for effecting this replacement. These are not testability tactics, per se, but techniques for replacing one component with a different version of itself. They include the following:

- Component replacement, which simply swaps the implementation of a component with a different implementation that (in the case of testability) has features that facilitate testing. Component replacement is often accomplished in a system's build scripts.
- Preprocessor macros that, when activated, expand to state-reporting code or activate probe statements that return or display information, or return control to a testing console.
- Aspects (in aspect-oriented programs) that handle the cross-cutting concern of how state is reported.

Limit Complexity

Complex software is harder to test. This is because, by the definition of complexity, its operating state space is very large and (all else being equal) it is more difficult to re-create an exact state in a large state space than to do so in a small state space. Because testing is not just about making the software fail but about finding the fault that caused the failure so that it can be removed, we are often concerned with making behavior repeatable. This category has three tactics:

- *Limit structural complexity.* This tactic includes avoiding or resolving cyclic dependencies between components, isolating and encapsulating dependencies on the external environment, and reducing dependencies between components in general (for example, reduce the number of external accesses to a module's public data). In object-oriented systems, you can simplify the inheritance hierarchy: Limit the number of classes from which a class is derived, or the number of classes derived from a class. Limit the depth of the inheritance tree, and the number of children of a class. Limit polymorphism and dynamic calls. One structural metric that has been shown empirically to correlate to testability is called the *response* of a class. The response of class C is a count of the number of methods of C plus the number of methods of other classes that are invoked by the methods of C. Keeping this metric low can increase testability.

  Having high cohesion, loose coupling, and separation of concerns—all modifiability tactics (see Chapter 7)—can also help with testability. They are a form of limiting the complexity of the architectural elements by giving each element a focused task with limited interaction with other elements. Separation of concerns can help achieve controllability and observability (as well as reducing the size of the overall program's state space). Controllability is critical to making testing tractable, as Robert Binder has noted: "A component that can act independently of others is more readily controllable. . . . With high coupling among classes it is typically more difficult to control the class under test, thus reducing testability. . . . If user interface capabilities are entwined with basic functions it will be more difficult to test each function" [Binder 94].

Also, systems that require complete data consistency at all times are often more complex than those that do not. If your requirements allow it, consider building your system under the "eventual consistency" model, where sooner or later (but maybe not right now) your data will reach a consistent state. This often makes system design simpler, and therefore easier to test.

Finally, some architectural styles lend themselves to testability. In a layered style, you can test lower layers first, then test higher layers with confidence in the lower layers.

- *Limit nondeterminism.* The counterpart to limiting structural complexity is limiting behavioral complexity, and when it comes to testing, nondeterminism is a very pernicious form of complex behavior. Nondeterministic systems are harder to test than deterministic systems. This tactic involves finding all the sources of nondeterminism, such as unconstrained parallelism, and weeding them out as much as possible. Some sources of nondeterminism are unavoidable—for instance, in multi-threaded systems that respond to unpredictable events—but for such systems, other tactics (such as record/playback) are available.

Figure 10.4 provides a summary of the tactics used for testability.

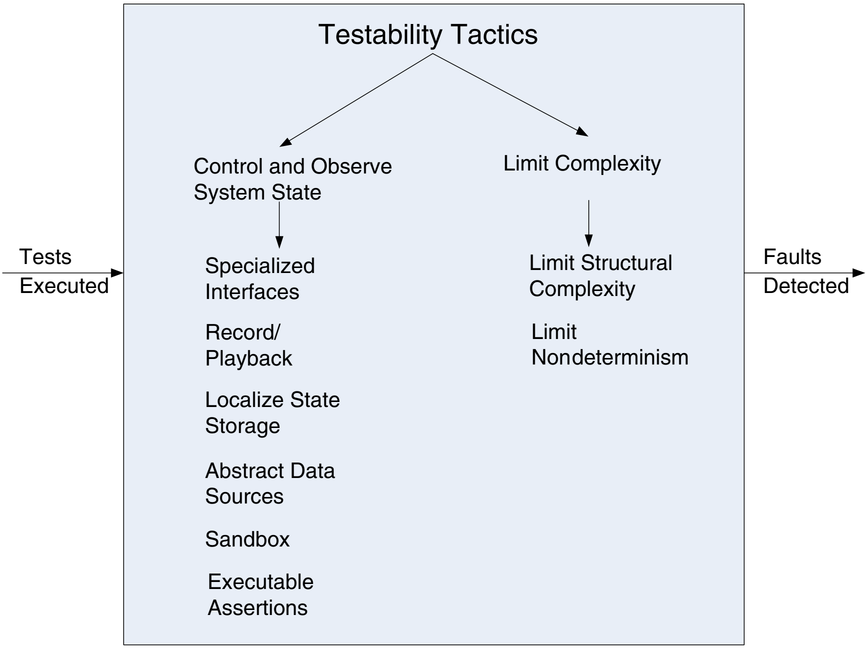


FIGURE 10.4 Testability tactics

10.3 A Design Checklist for Testability

Table 10.2 is a checklist to support the design and analysis process for testability.

TABLE 10.2 Checklist to Support the Design and Analysis Process for Testability

Category	Checklist
Allocation of Responsibilities	Determine which system responsibilities are most critical and hence need to be most thoroughly tested. Ensure that additional system responsibilities have been allocated to do the following: ▪ Execute test suite and capture results (external test or self-test) ▪ Capture (log) the activity that resulted in a fault *or* that resulted in unexpected (perhaps emergent) behavior that was not necessarily a fault ▪ Control and observe relevant system state for testing Make sure the allocation of functionality provides high cohesion, low coupling, strong separation of concerns, and low structural complexity.
Coordination Model	Ensure the system's coordination and communication mechanisms: ▪ Support the execution of a test suite and capture the results within a system or between systems ▪ Support capturing activity that resulted in a fault within a system or between systems ▪ Support injection and monitoring of state into the communication channels for use in testing, within a system or between systems ▪ Do not introduce needless nondeterminism
Data Model	Determine the major data abstractions that must be tested to ensure the correct operation of the system. ▪ Ensure that it is possible to capture the values of instances of these data abstractions ▪ Ensure that the values of instances of these data abstractions can be set when state is injected into the system, so that system state leading to a fault may be re-created ▪ Ensure that the creation, initialization, persistence, manipulation, translation, and destruction of instances of these data abstractions can be exercised and captured
Mapping among Architectural Elements	Determine how to test the possible mappings of architectural elements (especially mappings of processes to processors, threads to processes, and modules to components) so that the desired test response is achieved and potential race conditions identified. In addition, determine whether it is possible to test for illegal mappings of architectural elements.

continues

TABLE 10.2 Checklist to Support the Design and Analysis Process for Testability, *continued*

Category	Checklist
Resource Management	Ensure there are sufficient resources available to execute a test suite and capture the results. Ensure that your test environment is representative of (or better yet, identical to) the environment in which the system will run. Ensure that the system provides the means to do the following: ▪ Test resource limits ▪ Capture detailed resource usage for analysis in the event of a failure ▪ Inject new resource limits into the system for the purposes of testing ▪ Provide virtualized resources for testing
Binding Time	Ensure that components that are bound later than compile time can be tested in the late-bound context. Ensure that late bindings can be captured in the event of a failure, so that you can re-create the system's state leading to the failure. Ensure that the full range of binding possibilities can be tested.
Choice of Technology	Determine what technologies are available to help achieve the testability scenarios that apply to your architecture. Are technologies available to help with regression testing, fault injection, recording and playback, and so on? Determine how testable the technologies are that you have chosen (or are considering choosing in the future) and ensure that your chosen technologies support the level of testing appropriate for your system. For example, if your chosen technologies do not make it possible to inject state, it may be difficult to re-create fault scenarios.

Now That Your Architecture Is Set to Help You Test . . .

By Nick Rozanski, coauthor (with Eoin Woods) of Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives

In addition to architecting your system to make it amenable to testing, you will need to overcome two more specific and daunting challenges when testing very large or complex systems, namely test data and test automation.

Test Data

Your first challenge is how to create large, consistent and useful *test data sets*. This is a significant problem in my experience, particularly for integration testing (that is, testing a number of components to confirm that they work together correctly) and performance testing (confirming that the system meets it requirements for throughput, latency, and response time). For unit tests, and usually for user acceptance tests, the test data is typically created by hand.

For example, you might need 50 products, 100 customers, and 500 orders in your test database, so that you can test the functional steps involved in creating, amending, or deleting orders. This data has to be sufficiently varied to make testing worthwhile, it has to conform to all the referential integrity rules and other constraints of your data model, and you need to be able to calculate and specify the expected results of the tests.

I've seen—and been involved in—two ways of doing this: you either write a system to generate your test data, or you capture a representative data set from the production environment and anonymize it as necessary. (Anonymizing test data involves removing any sensitive information, such as personal data about people or organizations, financial details, and so on.)

Creating your own test data is the ideal, because you know what data you are using and can ensure that it covers all of your edge cases, but it is a lot of effort. Capturing data from the live environment is easier, assuming that there is a system there already, but you don't know what data and hence what coverage you're going to get, and you may have to take extra care to conform to privacy and data protection legislation.

This can have an impact on the system's architecture in a number of ways, and should be given due consideration early on by the architect. For example, the system may need to be able to capture live transactions, or take "snapshots" of live data, which can be used to generate test data. In addition, the test-data-generation system may need an architecture of its own.

Test Automation

Your second challenge is around *test automation*. In practice it is not possible to test large systems by hand because of the number of tests, their complexity, and the amount of checking of results that's required. In the ideal world, you create a test automation framework to do this automatically, which you feed with test data, and set running every night, or even run every time you check in something (the continuous integration model).

This is an area that is given too little attention on many large software development projects. It is often not budgeted for in the project plan, with an unwritten assumption that the effort needed to build it can be somehow "absorbed" into the development costs. A test automation framework can be a significantly complex thing in its own right (which raises the question of how you test it!). It should be scoped and planned like any other project deliverable.

Due consideration should be given to how the framework will invoke functions on the system under test, particularly for testing user interfaces, which is almost without exception a nightmare. (The execution of a UI test is highly dependent on the layout of the windows, the ordering of fields, and so on, which usually changes a lot in heavily user-focused systems. It is sometimes possible to execute window controls programmatically, but in the worst case you may have to record and replay keystrokes or mouse movements.)

There are lots of tools to help with this nowadays, such as Quick Test Pro, TestComplete, or Selenium for testing, and CruiseControl, Hudson, and TeamCity for continuous integration. A comprehensive list on the web can be found here: en.wikipedia.org/wiki/Test_automation.

10.4 Summary

Ensuring that a system is easily testable has payoffs both in terms of the cost of testing and the reliability of the system. A vehicle often used to execute the tests is the test harness. Test harnesses are software systems that encapsulate test resources such as test cases and test infrastructure so that it is easy to reapply tests across iterations and it is easy to apply the test infrastructure to new increments of the system. Another vehicle is the creation of test cases prior to the development of a component, so that developers know which tests their component must pass.

Controlling and observing the system state is a major class of testability tactics. Providing the ability to do fault injection, to record system state at key portions of the system, to isolate the system from its environment, and to abstract various resources are all different tactics to support the control and observation of a system and its components.

Complex systems are difficult to test because of the large state space in which their computations take place, and because of the larger number of interconnections among the elements of the system. Consequently, keeping the system simple is another class of tactics that supports testability.

10.5 For Further Reading

An excellent general introduction to software testing is [Beizer 90]. For a more modern take on testing, and from the software developer's perspective rather than the tester's, Freeman and Pryce cover test-driven development in the object-oriented realm [Freeman 09].

Bertolino and Strigini [Bertolino 96] are the developers of the model of testing shown in Figure 10.1.

Yin and Bieman [Yin 94] have written about executable assertions. Hartman [Hartman 10] describes a technique for using executable assertions as a means for detecting race conditions.

Bruntink and van Deursen [Bruntink 06] write about the impact of structure on testing.

Jeff Voas's foundational work on testability and the relationship between testability and reliability is worthwhile. There are several papers to choose from, but [Voas 95] is a good start that will point you to others.

10.6 Discussion Questions

1. A testable system is one that gives up its faults easily. That is, if a system contains a fault, then it doesn't take long or much effort to make that fault show up. On the other hand, fault tolerance is all about designing systems that jealously hide their faults; there, the whole idea is to make it very difficult for a system to reveal its faults. Is it possible to design a system that is both highly testable *and* highly fault tolerant, or are these two design goals inherently incompatible? Discuss.
2. "Once my system is in routine use by end users, it should *not* be highly testable, because if it still contains faults—and all systems probably do—then I don't want them to be easily revealed." Discuss.
3. Many of the tactics for testability are also useful for achieving modifiability. Why do you think that is?
4. Write some concrete testability scenarios for an automatic teller machine. How would you modify your design for the automatic teller machine to accommodate these scenarios?
5. What other quality attributes do you think testability is most in conflict with? What other quality attributes do you think testability is most compatible with?
6. One of our tactics is to limit nondeterminism. One method is to use locking to enforce synchronization. What impact does the use of locks have on other quality attributes?
7. Suppose you're building the next great social networking system. You anticipate that within a month of your debut, you will have half a million users. You can't pay half a million people to test your system, and yet it has to be robust and easy to use when all half a million are banging away at it. What should you do? What tactics will help you? Write a testability scenario for this social networking system.
8. Suppose you use executable assertions to improve testability. Make a case for, and then a case against, allowing the assertions to run in the production system as opposed to removing them after testing.

This page intentionally left blank

11

Usability

Any darn fool can make something complex; it takes a genius to make something simple.
—Albert Einstein

Usability is concerned with how easy it is for the user to accomplish a desired task and the kind of user support the system provides. Over the years, a focus on usability has shown itself to be one of the cheapest and easiest ways to improve a system's quality (or more precisely, the user's perception of quality).

Usability comprises the following areas:

- *Learning system features*. If the user is unfamiliar with a particular system or a particular aspect of it, what can the system do to make the task of learning easier? This might include providing help features.
- *Using a system efficiently*. What can the system do to make the user more efficient in its operation? This might include the ability for the user to redirect the system after issuing a command. For example, the user may wish to suspend one task, perform several operations, and then resume that task.
- *Minimizing the impact of errors*. What can the system do so that a user error has minimal impact? For example, the user may wish to cancel a command issued incorrectly.
- *Adapting the system to user needs*. How can the user (or the system itself) adapt to make the user's task easier? For example, the system may automatically fill in URLs based on a user's past entries.
- *Increasing confidence and satisfaction*. What does the system do to give the user confidence that the correct action is being taken? For example, providing feedback that indicates that the system is performing a long-running task and the extent to which the task is completed will increase the user's confidence in the system.

11.1 Usability General Scenario

The portions of the usability general scenarios are these:

- *Source of stimulus.* The end user (who may be in a specialized role, such as a system or network administrator) is always the source of the stimulus for usability.
- *Stimulus*. The stimulus is that the end user wishes to use a system efficiently, learn to use the system, minimize the impact of errors, adapt the system, or configure the system.
- *Environment.* The user actions with which usability is concerned always occur at runtime or at system configuration time.
- *Artifact*. The artifact is the system or the specific portion of the system with which the user is interacting.
- *Response*. The system should either provide the user with the features needed or anticipate the user's needs.
- *Response measure.* The response is measured by task time, number of errors, number of tasks accomplished, user satisfaction, gain of user knowledge, ratio of successful operations to total operations, or amount of time or data lost when an error occurs.

Table 11.1 enumerates the elements of the general scenario that characterize usability.

Figure 11.1 gives an example of a concrete usability scenario that you could generate using Table 11.1: The user downloads a new application and is using it productively after two minutes of experimentation.

TABLE 11.1 Usability General Scenario

Portion of Scenario	Possible Values
Source	End user, possibly in a specialized role
Stimulus	End user tries to use a system efficiently, learn to use the system, minimize the impact of errors, adapt the system, or configure the system.
Environment	Runtime or configuration time
Artifacts	System or the specific portion of the system with which the user is interacting
Response	The system should either provide the user with the features needed or anticipate the user's needs.
Response Measure	One or more of the following: task time, number of errors, number of tasks accomplished, user satisfaction, gain of user knowledge, ratio of successful operations to total operations, or amount of time or data lost when an error occurs

11.2 Tactics for Usability

Recall that usability is concerned with how easy it is for the user to accomplish a desired task, as well as the kind of support the system provides to the user. Researchers in human-computer interaction have used the terms *user initiative*, *system initiative*, and *mixed initiative* to describe which of the human-computer pair takes the initiative in performing certain actions and how the interaction proceeds. Usability scenarios can combine initiatives from both perspectives. For example, when canceling a command, the user issues a cancel—user initiative—and the system responds. During the cancel, however, the system may put up a progress indicator—system initiative. Thus, cancel may demonstrate mixed initiative. We use this distinction between user and system initiative to discuss the tactics that the architect uses to achieve the various scenarios.

Figure 11.2 shows the goal of the set of runtime usability tactics.

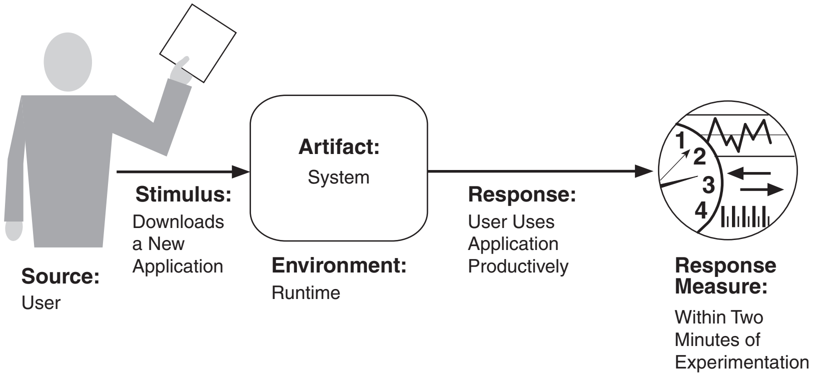

Artifact:
System
Stimulus:
Downloads a New Application
Response:
User Uses Application Productively
Source:
User
Environment:
Runtime
Response Measure:
Within Two Minutes of Experimentation
1
2
3
4

FIGURE 11.1 Sample concrete usability scenario

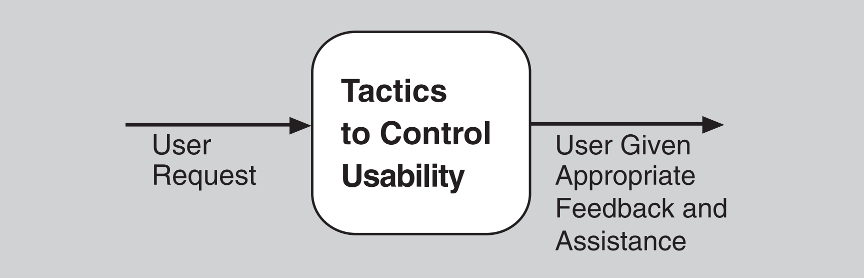

User Request
Tactics to Control Usability
User Given Appropriate Feedback and Assistance

FIGURE 11.2 The goal of runtime usability tactics

Separate the User Interface!

One of the most helpful things an architect can do to make a system usable is to facilitate experimentation with the user interface via the construction of rapid prototypes. Building a prototype, or several prototypes, to let real users experience the interface and give their feedback pays enormous dividends. The best way to do this is to design the software so that the user interface can be quickly changed.

Tactics for modifiability that we saw in Chapter 7 support this goal perfectly well, especially these:

- Increase semantic coherence, encapsulate, and co-locate related responsibilities, which localize user interface responsibilities to a single place
- Restrict dependencies, which minimizes the ripple effect to other software when the user interface changes
- Defer binding, which lets you make critical user interface choices without having to recode

Defer binding is especially helpful here, because you can expect that your product's user interface will face pressure to change during testing and even after it goes to market.

User interface generation tools are consistent with these tactics; most produce a single module with an abstract interface to the rest of the software. Many provide the capability to change the user interface after compile time. You can do your part by restricting dependencies on the generated module, should you later decide to adopt a different tool.

Much work in different user interface separation patterns occurred in the 1980s and 90s. With the advent of the web and the modernization of the model-view-controller (MVC) pattern to reflect web interfaces, MVC has become the dominant separation pattern. Now the MVC pattern is built into a wide variety of different frameworks. (See Chapter 14 for a discussion of MVC.) MVC makes it easy to provide multiple views of the data, supporting user initiative, as we discuss next.

Many times quality attributes are in conflict with each other. Usability and modifiability, on the other hand, often complement each other, because one of the best ways to make a system more usable is to make it modifiable. However, this is not always the case. In many systems business rules drive the UI—for example, specifying how to validate input. To realize this validation, the UI may need to call a server (which can negatively affect performance). To get around this performance penalty, the architect may choose to duplicate these rules in the client and the server, which then makes evolution difficult. Alas, the architect's life is never easy!

There is a connection between the achievement of usability and modifiability. The user interface design process consists of generating and then testing a user interface design. Deficiencies in the design are corrected and the process repeats. If the user interface has already been constructed as a portion of the system, then the system must be modified to reflect the latest design. Hence the connection with modifiability. This connection has resulted in standard patterns to support user interface design (see sidebar).

Support User Initiative

Once a system is executing, usability is enhanced by giving the user feedback as to what the system is doing and by allowing the user to make appropriate responses. For example, the tactics described next—*cancel*, *undo*, *pause/resume*, and *aggregate*—support the user in either correcting errors or being more efficient.

The architect designs a response for user initiative by enumerating and allocating the responsibilities of the system to respond to the user command. Here are some common examples of user initiative:

- *Cancel*. When the user issues a cancel command, the system must be listening for it (thus, there is the responsibility to have a constant listener that is not blocked by the actions of whatever is being canceled); the command being canceled must be terminated; any resources being used by the canceled command must be freed; and components that are collaborating with the canceled command must be informed so that they can also take appropriate action.
- *Undo*. To support the ability to undo, the system must maintain a sufficient amount of information about system state so that an earlier state may be restored, at the user's request. Such a record may be in the form of state "snapshots"—for example, checkpoints—or as a set of reversible operations. Not all operations can be easily reversed: for example, changing all occurrences of the letter "a" to the letter "b" in a document cannot be reversed by changing all instances of "b" to "a", because some of those instances of "b" may have existed prior to the original change. In such a case the system must maintain a more elaborate record of the change. Of course, some operations, such as ringing a bell, cannot be undone.
- *Pause/resume*. When a user has initiated a long-running operation—say, downloading a large file or set of files from a server—it is often useful to provide the ability to pause and resume the operation. Effectively pausing a long-running operation requires the ability to temporarily free resources so that they may be reallocated to other tasks.

- *Aggregate*. When a user is performing repetitive operations, or operations that affect a large number of objects in the same way, it is useful to provide the ability to aggregate the lower-level objects into a single group, so that the operation may be applied to the group, thus freeing the user from the drudgery (and potential for mistakes) of doing the same operation repeatedly. For example, aggregate all of the objects in a slide and change the text to 14-point font.

Support System Initiative

When the system takes the initiative, it must rely on a model of the user, the task being undertaken by the user, or the system state itself. Each model requires various types of input to accomplish its initiative. The *support system initiative* tactics are those that identify the models the system uses to predict either its own behavior or the user's intention. Encapsulating this information will make it easier for it to be tailored or modified. Tailoring and modification can be either dynamically based on past user behavior or offline during development. These tactics are the following:

- *Maintain task model*. The task model is used to determine context so the system can have some idea of what the user is attempting and provide assistance. For example, knowing that sentences start with capital letters would allow an application to correct a lowercase letter in that position.
- *Maintain user model*. This model explicitly represents the user's knowledge of the system, the user's behavior in terms of expected response time, and other aspects specific to a user or a class of users. For example, maintaining a user model allows the system to pace mouse selection so that not all of the document is selected when scrolling is required. Or a model can control the amount of assistance and suggestions automatically provided to a user. A special case of this tactic is commonly found in user interface *customization*, wherein a user can explicitly modify the system's user model.
- *Maintain system model*. Here the system maintains an explicit model of itself. This is used to determine expected system behavior so that appropriate feedback can be given to the user. A common manifestation of a system model is a progress bar that predicts the time needed to complete the current activity.

Figure 11.3 shows a summary of the tactics to achieve usability.

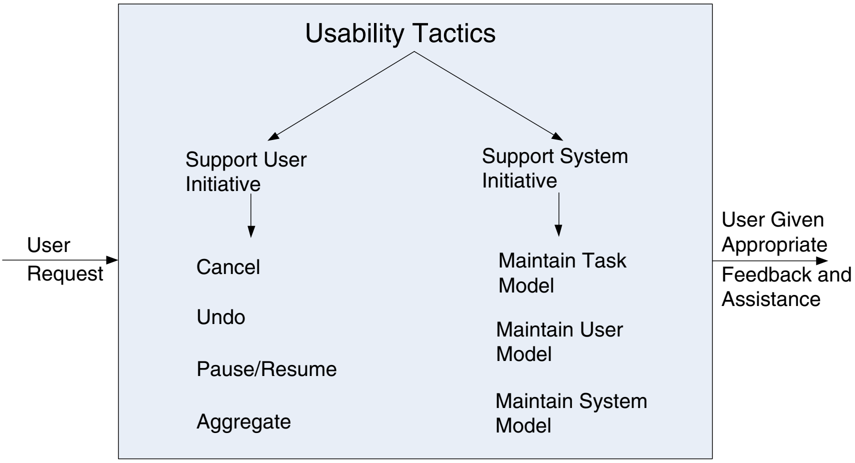


FIGURE 11.3 Usability tactics

11.3 A Design Checklist for Usability

Table 11.2 is a checklist to support the design and analysis process for usability.

TABLE 11.2 Checklist to Support the Design and Analysis Process for Usability

Category	Checklist
Allocation of Responsibilities	Ensure that additional system responsibilities have been allocated, as needed, to assist the user in the following: ▪ Learning how to use the system ▪ Efficiently achieving the task at hand ▪ Adapting and configuring the system ▪ Recovering from user and system errors
Coordination Model	Determine whether the properties of system elements' coordination—timeliness, currency, completeness, correctness, consistency—affect how a user learns to use the system, achieves goals or completes tasks, adapts and configures the system, recovers from user and system errors, and gains increased confidence and satisfaction. For example, can the system respond to mouse events and give semantic feedback in real time? Can long-running events be canceled in a reasonable amount of time?

continues

TABLE 11.2 Checklist to Support the Design and Analysis Process for Usability, *continued*

Category	Checklist
Data Model	Determine the major data abstractions that are involved with user-perceivable behavior. Ensure these major data abstractions, their operations, and their properties have been designed to assist the user in achieving the task at hand, adapting and configuring the system, recovering from user and system errors, learning how to use the system, and increasing satisfaction and user confidence. For example, the data abstractions should be designed to support *undo* and *cancel* operations: the transaction granularity should not be so great that canceling or undoing an operation takes an excessively long time.
Mapping among Architectural Elements	Determine what mapping among architectural elements is visible to the end user (for example, the extent to which the end user is aware of which services are local and which are remote). For those that are visible, determine how this affects the ways in which, or the ease with which, the user will learn how to use the system, achieve the task at hand, adapt and configure the system, recover from user and system errors, and increase confidence and satisfaction.
Resource Management	Determine how the user can adapt and configure the system's use of resources. Ensure that resource limitations under all user-controlled configurations will not make users less likely to achieve their tasks. For example, attempt to avoid configurations that would result in excessively long response times. Ensure that the level of resources will not affect the users' ability to learn how to use the system, or decrease their level of confidence and satisfaction with the system.
Binding Time	Determine which binding time decisions should be under user control and ensure that users can make decisions that aid in usability. For example, if the user can choose, at runtime, the system's configuration, or its communication protocols, or its functionality via plug-ins, you need to ensure that such choices do not adversely affect the user's ability to learn system features, use the system efficiently, minimize the impact of errors, further adapt and configure the system, or increase confidence and satisfaction.
Choice of Technology	Ensure the chosen technologies help to achieve the usability scenarios that apply to your system. For example, do these technologies aid in the creation of online help, the production of training materials, and the collection of user feedback? How usable are any of your chosen technologies? Ensure the chosen technologies do not adversely affect the usability of the system (in terms of learning system features, using the system efficiently, minimizing the impact of errors, adapting/configuring the system, and increasing confidence and satisfaction).

11.4 Summary

Architectural support for usability involves both allowing the user to take the initiative—in circumstances such as canceling a long-running command or undoing a completed command—and aggregating data and commands.

To be able to predict user or system responses, the system must keep an explicit model of the user, the system, and the task.

There is a strong relationship between supporting the user interface design process and supporting modifiability; this relation is promoted by patterns that enforce separation of the user interface from the rest of the system, such as the MVC pattern.

11.5 For Further Reading

Claire Marie Karat has investigated the relation between usability and business advantage [Karat 94].

Jakob Nielsen has also written extensively on this topic, including a calculation on the ROI of usability [Nielsen 08].

Bonnie John and Len Bass have investigated the relation between usability and software architecture. They have enumerated around two dozen usability scenarios that have architectural impact and given associated patterns for these scenarios [Bass 03].

Greg Hartman has defined attentiveness as the ability of the system to support user initiative and allow cancel or pause/resume [Hartman 10].

Some of the patterns for separating the user interface are Arch/Slinky, Seeheim, and PAC. These are discussed in Chapter 8 of *Human-Computer Interaction* [Dix 04].

11.6 Discussion Questions

1. Write a concrete usability scenario for your automobile that specifies how long it takes you to set your favorite radio stations? Now consider another part of the driver experience and create scenarios that test other aspects of the response measures from the general scenario table.

2. Write a concrete usability scenario for an automatic teller machine. How would your design be modified to satisfy these scenarios?

3. How might usability trade off against security? How might it trade off against performance?

4. Pick a few of your favorite web sites that do similar things, such as social networking or online shopping. Now pick one or two appropriate responses from the usability general scenario (such as “achieve the task at hand”) and a correspondingly appropriate response measure. Using the response and response measure you chose, compare the web sites’ usability.

5. Specify the data model for a four-function calculator that allows undo.

6. Why is it that in so many systems, the cancel button in a dialog box appears to be unresponsive? What architectural principles do you think were ignored in these systems?

7. Why do you think that progress bars frequently behave erratically, moving from 10 to 90 percent in one step and then getting stuck on 90 percent?

8. Research the crash of Air France Flight 296 into the forest at Habsheim, France, on June 26, 1988. The pilots said they were unable to read the digital display of the radio altimeter or hear its audible readout. If they could have, do you believe the crash would have been averted? In this context, discuss the relationship between usability and safety.

12

Other Quality Attributes

Quality is not an act, it is a habit.
—Aristotle

Chapters 5–11 each dealt with a particular quality attribute important to software systems. Each of those chapters discussed how its particular quality attribute is defined, gave a general scenario for that quality attribute, and showed how to write specific scenarios to express precise shades of meaning concerning that quality attribute. And each gave a collection of techniques to achieve that quality attribute in an architecture. In short, each chapter presented a kind of portfolio for specifying and designing to achieve a particular quality attribute.

Those seven chapters covered seven of the most important quality attributes, in terms of their occurrence in modern software-reliant systems. However, as is no doubt clear, seven only begins to scratch the surface of the quality attributes that you might find needed in a software system you're working on.

Is cost a quality attribute? It is not a technical quality attribute, but it certainly affects fitness for use. We consider economic factors in Chapter 23.

This chapter will give a brief introduction to a few other quality attributes—a sort of "B list" of quality attributes—but, more important, show how to build the same kind of specification or design portfolio for a quality attribute not covered in our list.

12.1 Other Important Quality Attributes

Besides the quality attributes we've covered in depth in Chapters 5–11, some others that arise frequently are variability, portability, development distributability, scalability and elasticity, deployability, mobility, and monitorability. We discuss "green" computing in Section 12.3.

Variability

Variability is a special form of modifiability. It refers to the ability of a system and its supporting artifacts such as requirements, test plans, and configuration specifications to support the production of a set of variants that differ from each other in a preplanned fashion. Variability is an especially important quality attribute in a software product line (this will be explored in depth in Chapter 25), where it means the ability of a core asset to adapt to usages in the different product contexts that are within the product line scope. The goal of variability in a software product line is to make it easy to build and maintain products in the product line over a period of time. Scenarios for variability will deal with the binding time of the variation and the people time to achieve it.

Portability

Portability is also a special form of modifiability. Portability refers to the ease with which software that was built to run on one platform can be changed to run on a different platform. Portability is achieved by minimizing platform dependencies in the software, isolating dependencies to well-identified locations, and writing the software to run on a "virtual machine" (such as a Java Virtual Machine) that encapsulates all the platform dependencies within. Scenarios describing portability deal with moving software to a new platform by expending no more than a certain level of effort or by counting the number of places in the software that would have to change.

Development Distributability

Development distributability is the quality of designing the software to support distributed software development. Many systems these days are developed using globally distributed teams. One problem that must be overcome when developing with distributed teams is coordinating their activities. The system should be designed so that coordination among teams is minimized. This minimal coordination needs to be achieved both for the code and for the data model. Teams working on modules that communicate with each other may need to negotiate the interfaces of those modules. When a module is used by many other modules, each developed by a different team, communication and negotiation become more complex and burdensome. Similar considerations apply for the data model. Scenarios for development distributability will deal with the compatibility of the communication structures and data model of the system being developed and the coordination mechanisms of the organizations doing the development.

Scalability

Two kinds of scalability are horizontal scalability and vertical scalability. Horizontal scalability (scaling out) refers to adding more resources to logical units, such as adding another server to a cluster of servers. Vertical scalability (scaling up) refers to adding more resources to a physical unit, such as adding more memory to a single computer. The problem that arises with either type of scaling is how to effectively utilize the additional resources. Being *effective* means that the additional resources result in a measurable improvement of some system quality, did not require undue effort to add, and did not disrupt operations. In cloud environments, horizontal scalability is called *elasticity*. Elasticity is a property that enables a customer to add or remove virtual machines from the resource pool (see Chapter 26 for further discussion of such environments). These virtual machines are hosted on a large collection of upwards of 10,000 physical machines that are managed by the cloud provider. Scalability scenarios will deal with the impact of adding or removing resources, and the measures will reflect associated availability and the load assigned to existing and new resources.

Deployability

Deployability is concerned with how an executable arrives at a host platform and how it is subsequently invoked. Some of the issues involved in deploying software are: How does it arrive at its host (push, where updates are sent to users unbidden, or pull, where users must explicitly request updates)? How is it integrated into an existing system? Can this be done while the existing system is executing? Mobile systems have their own problems in terms of how they are updated, because of concerns about bandwidth. Deployment scenarios will deal with the type of update (push or pull), the form of the update (medium, such as DVD or Internet download, and packaging, such as executable, app, or plug-in), the resulting integration into an existing system, the efficiency of executing the process, and the associated risk.

Mobility

Mobility deals with the problems of movement and affordances of a platform (e.g., size, type of display, type of input devices, availability and volume of bandwidth, and battery life). Issues in mobility include battery management, reconnecting after a period of disconnection, and the number of different user interfaces necessary to support multiple platforms. Scenarios will deal with specifying the desired effects of mobility or the various affordances. Scenarios may also deal with variability, where the same software is deployed on multiple (perhaps radically different) platforms.

Monitorability

Monitorability deals with the ability of the operations staff to monitor the system while it is executing. Items such as queue lengths, average transaction processing time, and the health of various components should be visible to the operations staff so that they can take corrective action in case of potential problems. Scenarios will deal with a potential problem and its visibility to the operator, and potential corrective action.

Safety

In 2009 an employee of the Shushenskaya hydroelectric power station in Siberia sent commands over a network to remotely, and accidentally, activate an unused turbine. The offline turbine created a "water hammer" that flooded and then destroyed the plant and killed dozens of workers.

The thought that software could kill people used to belong in the realm of kitschy computers-run-amok science fiction. Sadly, it didn't stay there. As software has come to control more and more of the devices in our lives, software safety has become a critical concern.

Safety is not purely a software concern, but a concern for any system that can affect its environment. As such it receives mention in Section 12.3, where we discuss system quality attributes. But there are means to address safety that are wholly in the software realm, which is why we discuss it here as well.

Software safety is about the software's ability to avoid entering states that cause or lead to damage, injury, or loss of life to actors in the software's environment, and to recover and limit the damage when it does enter into bad states. Another way to put this is that safety is concerned with the prevention of and recovery from hazardous failures. Because of this, the architectural concerns with safety are almost identical to those for availability, which is also about avoiding and recovering from failures. Tactics for safety, then, overlap with those for availability to a large degree. Both comprise tactics to prevent failures and to detect and recover from failures that do occur.

Safety is not the same as reliability. A system can be reliable (consistent with its specification) but still unsafe (for example, when the specification ignores conditions leading to unsafe action). In fact, paying careful attention to the specification for safety-critical software is perhaps the most powerful thing you can do to produce safe software. Failures and hazards cannot be detected, prevented, or ameliorated if the software has not been designed with them in mind. Safety is frequently engineered by performing failure mode and effects analysis, hazard analysis, and fault tree analysis. (These techniques are discussed in Chapter 5.) These techniques are intended to discover possible hazards that could result from the system's operation and provide plans to cope with these hazards.

12.2 Other Categories of Quality Attributes

We have primarily focused on product qualities in our discussions of quality attributes, but there are other types of quality attributes that measure "goodness" of something other than the final product. Here are three:

Conceptual Integrity of the Architecture

Conceptual integrity refers to consistency in the design of the architecture, and it contributes to the understandability of the architecture and leads to fewer errors of confusion. Conceptual integrity demands that the same thing is done in the same way through the architecture. In an architecture with conceptual integrity, less is more. For example, there are countless ways that components can send information to each other: messages, data structures, signaling of events, and so forth. An architecture with conceptual integrity would feature one way only, and only provide alternatives if there was a compelling reason to do so. Similarly, components should all report and handle errors in the same way, log events or transactions in the same way, interact with the user in the same way, and so forth.

Quality in Use

ISO/IEC 25010, which we discuss in Section 12.4, has a category of qualities that pertain to the use of the system by various stakeholders. For example, time-to-market is an important characteristic of a system, but it is not discernible from an examination of the product itself. Some of the qualities in this category are these:

- *Effectiveness*. This refers to the distinction between building the system correctly (the system performs according to its requirements) and building the correct system (the system performs in the manner the user wishes). Effectiveness is a measure of whether the system is correct.
- *Efficiency*. The effort and time required to develop a system. Put another way, what is the architecture's impact on the project's cost and schedule? Would a different set of architectural choices have resulted in a system that would be faster or cheaper to bring to fruition? Efficiency can include training time for developers; an architecture that uses technology unfamiliar to the staff on hand is less buildable. Is the architecture appropriate for the organization in terms of its experience and its available supporting infrastructure (such as test facilities or development environments)?
- *Freedom from risk*. The degree to which a product or system affects economic status, human life, health, or the environment.

A special case of efficiency is how easy it is to build (that is, compile and assemble) the system after a change. This becomes critical during testing. A recompile process that takes hours or overnight is a schedule-killer. Architects have control over this by managing dependencies among modules. If the architect doesn't do this, then what often happens is that some bright-eyed developer writes a makefile early on, it works, and people add to it and add to it. Eventually the project ends up with a seven-hour compile step and very unhappy integrators and testers who are already behind schedule (because they always are).

Marketability

An architecture's marketability is another quality attribute of concern. Some systems are well known by their architectures, and these architectures sometimes carry a meaning all their own, independent of what other quality attributes they bring to the system. The current craze in building cloud-based systems has taught us that the perception of an architecture can be more important than the qualities the architecture brings. Many organizations have felt they had to build cloud-based systems (or some other *technology du jour*) whether or not that was the correct technical choice.

12.3 Software Quality Attributes and System Quality Attributes

Physical systems, such as aircraft or automobiles or kitchen appliances, that rely on software embedded within are designed to meet a whole other litany of quality attributes: weight, size, electric consumption, power output, pollution output, weather resistance, battery life, and on and on. For many of these systems, safety tops the list (see the sidebar).

Sometimes the software architecture can have a surprising effect on the system's quality attributes. For example, software that makes inefficient use of computing resources might require additional memory, a faster processor, a bigger battery, or even an additional processor. Additional processors can add to a system's power consumption, weight, required cabinet space, and of course expense.

Green computing is an issue of growing concern. Recently there was a controversy about how much greenhouse gas was pumped into the atmosphere by Google's massive processor farms. Given the daily output and the number of daily requests, it is possible to estimate how much greenhouse gas *you* cause to be emitted each time *you* ask Google to perform a search. (Current estimates range from 0.2 grams to 7 grams of CO_2.) Green computing is all the rage. Eve Troeh, on the American Public Media show "Marketplace" (July 5, 2011), reports:

> Two percent of all U.S. electricity now goes to data centers, according to the Environmental Protection Agency. Electricity has become the biggest cost for processing data—more than the equipment to do it, more than the buildings to house that equipment. . . . Google's making data servers that can float offshore, cooled by ocean breezes. HP has plans to put data servers near farms, and power them with methane gas from cow pies.

The lesson here is that if you are the architect for software that resides in a larger system, you will need to understand the quality attributes that are important for the containing system to achieve, and work with the system architects and engineers to see how your software architecture can contribute to achieving them.

The Vanishing Line between Software and System Qualities

This is a book about software architecture, and so we treat quality attributes from a software architect's perspective. But you may have already noticed that the quality attributes that the software architect can bring to the party are limited by the architecture of the system in which the software runs.

For example:

- The performance of a piece of software is fundamentally constrained by the performance of the computer that runs it. No matter how well you design the software, you just can't run the latest whole-earth weather forecasting models on Grampa's Commodore 64 and hope to know if it's going to rain tomorrow.
- Physical security is probably more important and more effective than software security at preventing fraud and theft. If you don't believe this, write your laptop's password on a slip of paper, tape it to your laptop, and leave it in an unlocked car with the windows down. (Actually, don't really do that. Consider this a thought experiment.)
- If we're being perfectly honest here, how usable is a device for web browsing that has a screen smaller than a credit card and keys the size of a raisin?

For me, nowhere is the barrier between software and system more nebulous than in the area of safety. The thought that software—strings of 0's and 1's—can kill or maim or destroy is still an unnatural notion. Of course, it's not the 0's and 1's that wreak havoc. At least, not directly. It's what they're connected to. Software, and the system in which it runs, has to be connected to the outside world in some way before it can do damage. That's the good news. The bad news is that the good news isn't all that good. Software *is* connected to the outside world, always. If your program has no effect whatsoever that is observable outside of itself, it probably serves no purpose.

There are notorious examples of software-related failures. The Siberian hydroelectric plant catastrophe mentioned in the text, the Therac-25 fatal radiation overdose, the Ariane 5 explosion, and a hundred lesser known accidents all caused harm because the *software* was part of a *system* that included a turbine, an X-ray emitter, or a rocket's steering controls, in the examples just cited. In these cases, flawed software commanded some hardware in the system to take a disastrous action, and the hardware simply obeyed. Actuators are devices that connect hardware to software; they are the bridge between the world of 0's and 1's and the world of motion and control. Send a digital value to an actuator (or write a bit string in the hardware register corresponding to the actuator) and that value is translated to some mechanical action, for better or worse.

But connection to an actuator is not required for software-related disasters. Sometimes all the computer has to do is send erroneous information to its human operators. In September 1983, a Soviet satellite sent data to its ground system computer, which interpreted that data as a missile launched from the United States aimed at Moscow. Seconds later, the computer reported a second missile in flight. Soon, a third, then a fourth, and then a fifth appeared. Soviet Strategic Rocket Forces lieutenant colonel Stanislav Yevgrafovich Petrov made the astonishing decision to ignore the warning system, believing it to be in error. He thought it extremely unlikely that the U.S. would have fired just a few missiles, thereby inviting total retaliatory destruction. He decided to wait it out, to see if the missiles were real—that is, to see if his country's capital city was going to be incinerated. As we know, it wasn't. The Soviet system had mistaken a rare sunlight condition for missiles in flight. Similar mistakes have occurred on the U.S. side.

Of course, the humans don't always get it right. On the dark and stormy night of June 1, 2009, Air France flight 447 from Rio de Janeiro to Paris plummeted into the Atlantic Ocean, killing all on board. The Airbus A-330's flight recorders were not recovered until May 2011, and as this book goes to publication it appears that the pilots never knew that the aircraft had entered a high-altitude stall. The sensors that measure airspeed had become clogged with ice and therefore unreliable. The software was required to disengage the autopilot in this situation, which it did. The human pilots thought the aircraft was going too fast (and in danger of structural failure) when in fact it was going too slow (and falling). During the entire three-minute-plus plunge from 38,000 feet, the pilots kept trying to pull the nose up and throttles back to lower the speed. It's a good bet that adding to the confusion was the way the A-330's stall warning system worked. When the system detects a stall, it emits a loud audible alarm. The computers deactivate the stall warning when they "think" that the angle of attack measurements are invalid. This can occur when the airspeed readings are very low. That is exactly what happened with Air France 447: Its forward speed dropped below 60 knots, and the angle of attack was extremely high. As a consequence of a rule in the flight control *software*, the stall warning stopped and started several times. Worse, it came on whenever the pilot let the nose fall a bit (increasing the airspeed and taking the readings into the "valid" range, but still in stall) and then stopped when he pulled back. That is, doing the right thing resulted in the wrong feedback and vice versa.

Was this an unsafe system, or a safe system unsafely operated? Ultimately the courts will decide.

Software that can physically harm us is a fact of our modern life. Sometimes the link between software and physical harm is direct, as in the Ariane example, and sometimes it's much more tenuous, as in the Air France 447 example. But as software professionals, we cannot take refuge in the fact that our software can't actually inflict harm any more than the person who shouts "Fire!" in a crowded theater can claim it was the stampede, not the shout, that caused injury.

—PCC

12.4 Using Standard Lists of Quality Attributes—or Not

Architects have no shortage of lists of quality attributes for software systems at their disposal. The standard with the pause-and-take-a-breath title of "ISO/IEC FCD 25010: Systems and software engineering—Systems and software product Quality Requirements and Evaluation (SQuaRE)—System and software quality models," is a good example. The standard divides quality attributes into those supporting a "quality in use" model and those supporting a "product quality" model. That division is a bit of a stretch in some places, but nevertheless begins a divide-and-conquer march through a breathtaking array of qualities. See Figure 12.1 for this array.

The standard lists the following quality attributes that deal with product quality:

- *Functional suitability*. The degree to which a product or system provides functions that meet stated and implied needs when used under specified conditions
- *Performance efficiency*. Performance relative to the amount of resources used under stated conditions
- *Compatibility*. The degree to which a product, system, or component can exchange information with other products, systems, or components, and/or perform its required functions, while sharing the same hardware or software environment
- *Usability*. The degree to which a product or system can be used by specified users to achieve specified goals with effectiveness, efficiency, and satisfaction in a specified context of use

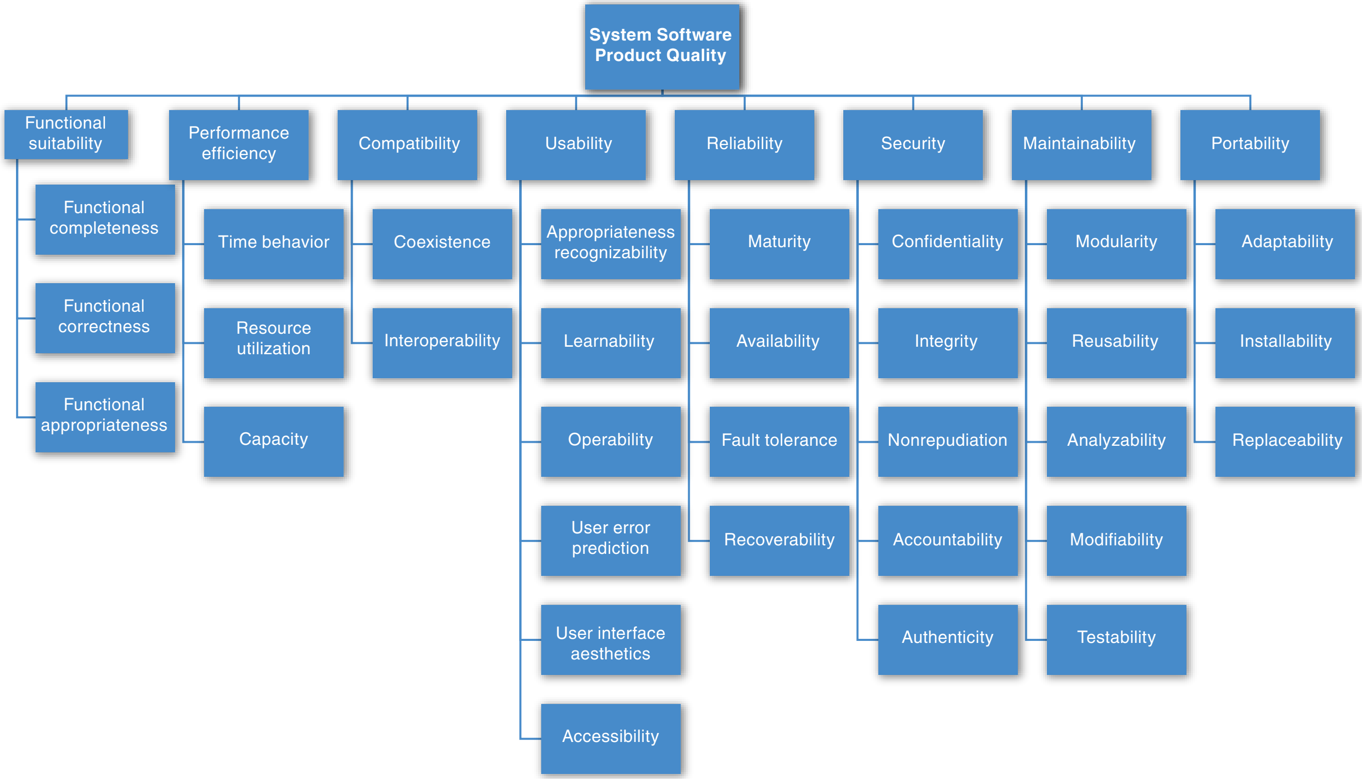


FIGURE 12.1 The ISO/IEC FCD 25010 product quality standard

- *Reliability*. The degree to which a system, product, or component performs specified functions under specified conditions for a specified period of time
- *Security*. The degree to which a product or system protects information and data so that persons or other products or systems have the degree of data access appropriate to their types and levels of authorization
- *Maintainability*. The degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers
- *Portability*. The degree of effectiveness and efficiency with which a system, product, or component can be transferred from one hardware, software, or other operational or usage environment to another

In ISO 25010, these "quality characteristics" are each composed of "quality subcharacteristics" (for example, nonrepudiation is a subcharacteristic of security). The standard slogs through almost five dozen separate descriptions of quality subcharacteristics in this way. It defines for us the qualities of "pleasure" and "comfort." It distinguishes among "functional correctness" and "functional completeness," and then adds "functional appropriateness" for good measure. To exhibit "compatibility," systems must either have "interoperability" or just plain "coexistence." "Usability" is a product quality, not a quality-in-use quality, although it includes "satisfaction," which *is* a quality-in-use quality. "Modifiability" and "testability" are both part of "maintainability." So is "modularity," which is a strategy for achieving a quality rather than a goal in its own right. "Availability" is part of "reliability." "Interoperability" is part of "compatibility." And "scalability" isn't mentioned at all.

Got all that?

Lists like these—and there are many—do serve a purpose. They can be helpful checklists to assist requirements gatherers in making sure that no important needs were overlooked. Even more useful than standalone lists, they can serve as the basis for creating your own checklist that contains the quality attributes of concern in your domain, your industry, your organization, and your products. Quality attribute lists can also serve as the basis for establishing measures. If "pleasure" turns out to be an important concern in your system, how do you measure it to know if your system is providing enough of it?

However, general lists like these also have drawbacks. First, no list will ever be complete. As an architect, you will be called upon to design a system to meet a stakeholder concern not foreseen by any list-maker. For example, some writers speak of "manageability," which expresses how easy it is for system administrators to manage the application. This can be achieved by inserting useful instrumentation for monitoring operation and for debugging and performance tuning. We know of an architecture that was designed with the conscious goal of retaining key staff and attracting talented new hires to a quiet region of the American Midwest. That system's architects spoke of imbuing the system with "Iowability." They achieved it by bringing in state-of-the-art technology and giving their development teams wide creative latitude. Good luck finding "Iowability" in any standard list of quality attributes, but that QA was as important to that organization as any other.

Second, lists often generate more controversy than understanding. You might argue persuasively that "functional correctness" should be part of "reliability," or that "portability" is just a kind of "modifiability," or that "maintainability" is a kind of "modifiability" (not the other way around). The writers of ISO 25010 apparently spent time and effort deciding to make security its own characteristic, instead of a subcharacteristic of functionality, which it was in a previous version. We believe that effort in making these arguments could be better spent elsewhere.

Third, these lists often purport to be *taxonomies*, which are lists with the special property that every member can be assigned to exactly one place. Quality attributes are notoriously squishy in this regard. We discussed denial of service as being part of security, availability, performance, and usability in Chapter 4.

Finally, these lists force architects to pay attention to every quality attribute on the list, even if only to finally decide that the particular quality attribute is irrelevant to their system. Knowing how to quickly decide that a quality attribute is irrelevant to a specific system is a skill gained over time.

These observations reinforce the lesson introduced in Chapter 4 that quality attribute names, by themselves, are largely useless and are at best invitations to begin a conversation; that spending time worrying about what qualities are subqualities of what other qualities is also almost useless; and that scenarios provide the best way for us to specify precisely what we mean when we speak of a quality attribute.

Use standard lists of quality attributes to the extent that they are helpful as checklists, but don't feel the need to slavishly adhere to their terminology.

12.5 Dealing with "X-ability": Bringing a New Quality Attribute into the Fold

Suppose, as an architect, you must deal with a quality attribute for which there is no compact body of knowledge, no "portfolio" like Chapters 5–11 provided for those seven QAs? Suppose you find yourself having to deal with a quality attribute like "green computing" or "manageability" or even "Iowability"? What do you do?

Capture Scenarios for the New Quality Attribute

The first thing to do is interview the stakeholders whose concerns have led to the need for this quality attribute. You can work with them, either individually or as a group, to build a set of attribute characterizations that refine what is meant by the QA. For example, security is often decomposed into concerns such as confidentiality, integrity, availability, and others. After that refinement, you can work with the stakeholders to craft a set of specific scenarios that characterize what is meant by that QA.

Once you have a set of specific scenarios, then you can work to generalize the collection. Look at the set of stimuli you've collected, the set of responses, the set of response measures, and so on. Use those to construct a general scenario by making each part of the general scenario a generalization of the specific instances you collected.

In our experience, the steps described so far tend to consume about half a day.

Assemble Design Approaches for the New Quality Attribute

After you have a set of guiding scenarios for the QA, you can assemble a set of design approaches for dealing with it. You can do this by

1. Revisiting a body of patterns you're familiar with and asking yourself how each one affects the QA of interest.
2. Searching for designs that have had to deal with this QA. You can search on the name you've given the QA itself, but you can also search for the terms you chose when you refined the QA into subsidiary attribute characterizations (such as "confidentiality" for the QA of security).
3. Finding experts in this area and interviewing them or simply writing and asking them for advice.
4. Using the general scenario to try to catalog a list of design approaches to produce the responses in the response category.
5. Using the general scenario to catalog a list of ways in which a problematic architecture would fail to produce the desired responses, and thinking of design approaches to head off those cases.

Model the New Quality Attribute

If you can build a conceptual model of the quality attribute, this can be helpful in creating a set of design approaches for it. By "model," we don't mean anything more than understanding the set of parameters to which the quality attribute is sensitive. For example, a model of modifiability might tell us that modifiability is a function of how many places in a system have to be changed in response to a modification, and the interconnectedness of those places. A model for performance might tell us that throughput is a function of transactional workload, the dependencies among the transactions, and the number of transactions that can be processed in parallel.

Once you have a model for your QA, then you can work to catalog the architectural approaches (tactics and patterns) open to you for manipulating each of the relevant parameters in your favor.

Assemble a Set of Tactics for the New Quality Attribute

There are two sources that can be used to derive tactics for any quality attribute: models and experts.

Figure 12.2 shows a queuing model for performance. Such models are widely used to analyze the latency and throughput of various types of queuing systems, including manufacturing and service environments, as well as computer systems.

Within this model, there are seven parameters that can affect the latency that the model predicts:

- Arrival rate
- Queuing discipline
- Scheduling algorithm
- Service time
- Topology
- Network bandwidth
- Routing algorithm

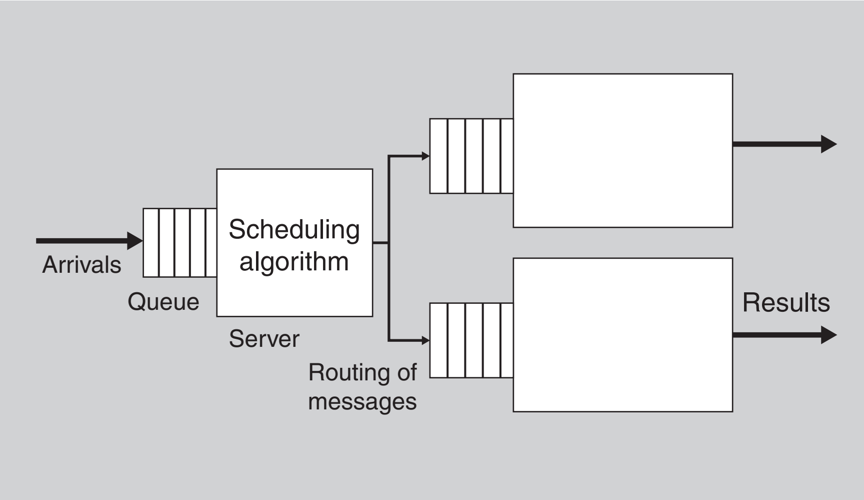


FIGURE 12.2 A generic queuing model

These are the *only* parameters that can affect latency within this model. This is what gives the model its power. Furthermore, each of these parameters can be affected by various architectural decisions. This is what makes the model useful for an architect. For example, the routing algorithm can be fixed or it could be a load-balancing algorithm. A scheduling algorithm must be chosen. The topology can be affected by dynamically adding or removing new servers. And so forth.

The process of generating tactics based on a model is this:

- Enumerate the parameters of the model
- For each parameter, enumerate the architectural decisions that can affect this parameter

What results is a list of tactics to, in the example case, control performance and, in the more general case, to control the quality attribute that the model is concerned with. This makes the design problem seem much more tractable. This list of tactics is finite and reasonably small, because the number of parameters of the model is bounded, and for each parameter, the number of architectural decisions to affect the parameter is limited.

Deriving tactics from models is fine as long as the quality attribute in question has a model. Unfortunately, the number of such models is limited and is a subject of active research. There are no good *architectural* models for usability or security, for example. In the cases where we had no model to work from, we did four things to catalog the tactics:

1. We interviewed experts in the field, asking them what they do as architects to improve the quality attribute response.
2. We examined systems that were touted as having high usability (or testability, or whatever tactic we were focusing on).
3. We scoured the relevant design literature looking for common themes in design.
4. We examined documented architectural patterns to look for ways they achieved the quality attribute responses touted for them.

Construct Design Checklists for the New Quality Attribute

Finally, examine the seven categories of design decisions in Chapter 4 and ask yourself (or your experts) how to specialize your new quality of interest to these categories. In particular, think about reviewing a software architecture and trying to figure out how well it satisfies your new qualities in these seven categories. What questions would you ask the architect of that system to understand how the design attempts to achieve the new quality? These are the basis for the design checklist.

12.6 For Further Reading

For most of the quality attributes we discussed in this chapter, the Internet is your friend. You can find reasonable discussions of scalability, portability, and deployment strategies using your favorite search engine. Mobility is harder to find because it has so many meanings, but look under "mobile computing" as a start.

Distributed development is a topic covered in the International Conference on Global Software Engineering, and looking at the proceedings of this conference will give you access to the latest research in this area (www.icgse.org).

Release It! [Nygard 07] has a good discussion of monitorability (which he calls transparency) as well as potential problems that are manifested after extended operation of a system. The book also includes various patterns for dealing with some of the problems.

To gain an appreciation for the importance of software safety, we suggest reading some of the disaster stories that arise when software fails. A venerable source is the ACM Risks Forum newsgroup, known as comp.risks in the USENET community, available at www.risks.org. This list has been moderated by Peter Neumann since 1985 and is still going strong.

Nancy Leveson is an undisputed thought leader in the area of software and safety. If you're working in safety-critical systems, you should become familiar with her work. You can start small with a paper like [Leveson 04], which discusses a number of software-related factors that have contributed to spacecraft accidents. Or you can start at the top with [Leveson 11], a book that treats safety in the context of today's complex, sociotechnical, software-intensive systems.

The Federal Aviation Administration is the U.S. government agency charged with oversight of the U.S. airspace system, and the agency is extremely concerned about safety. Their 2000 System Safety Handbook is a good practical overview of the topic [FAA 00].

IEEE STD-1228-1994 ("Software Safety Plans") defines best practices for conducting software safety hazard analyses, to help ensure that requirements and attributes are specified for safety-critical software [IEEE 94]. The aeronautical standard DO-178B (due to be replaced by DO-178C as this book goes to publication) covers software safety requirements for aerospace applications.

A discussion of safety tactics can be found in the work of Wu and Kelly [Wu 06].

In particular, interlocks are an important tactic for safety. They enforce some safe sequence of events, or ensure that a safe condition exists before an action is taken. Your microwave oven shuts off when you open the door because of a hardware interlock. Interlocks can be implemented in software also. For an interesting case study of this, see [Wozniak 07].

12.7 Discussion Questions

1. The Kingdom of Bhutan measures the happiness of its population, and government policy is formulated to increase Bhutan's GNH (gross national happiness). Go read about how the GNH is measured (try www.grossnationalhappiness.com) and then sketch a general scenario for the quality attribute of *happiness* that will let you express concrete happiness requirements for a software system.

2. Choose a quality attribute not described in Chapters 5–11. For that quality attribute, assemble a set of specific scenarios that describe what you mean by it. Use that set of scenarios to construct a general scenario for it.

3. For the QA you chose for discussion question 2, assemble a set of design approaches (patterns and tactics) that help you achieve it.

4. For the QA you chose for discussion question 2, develop a design checklist for that quality attribute using the seven categories of guiding quality design decisions outlined in Chapter 4.

5. What might cause you to add a tactic or pattern to the sets of quality attributes already described in Chapters 5–11 (or any other quality attribute, for that matter)?

6. According to slate.com and other sources, a teenage girl in Germany "went into hiding after she forgot to set her Facebook birthday invitation to private and accidentally invited the entire Internet. After 15,000 people confirmed they were coming, the girl's parents canceled the party, notified police, and hired private security to guard their home." Fifteen hundred people showed up anyway; several minor injuries ensued. Is Facebook "unsafe"? Discuss.

7. Author James Gleick ("A Bug and a Crash," www.around.com/ariane.html) writes that "It took the European Space Agency 10 years and $7 billion to produce Ariane 5, a giant rocket capable of hurling a pair of three-ton satellites into orbit with each launch. . . . All it took to explode that rocket less than a minute into its maiden voyage . . . was a small computer program trying to stuff a 64-bit number into a 16-bit space. One bug, one crash. Of all the careless lines of code recorded in the annals of computer science, this one may stand as the most devastatingly efficient." Write a safety scenario that addresses the Ariane 5 disaster and discuss tactics that might have prevented it.

8. Discuss how you think development distributability tends to "trade off" against the quality attributes of performance, availability, modifiability, and interoperability.

Extra Credit: Close your eyes and, without peeking, spell "distributability." Bonus points for successfully saying "development distributability" three times as fast as you can.

9. What is the relationship between mobility and security?

10. Relate monitorability to observability and controllability, the two parts of testability. Are they the same? If you want to make your system more of one, can you just optimize for the other?

13

Architectural Tactics and Patterns

I have not failed. I've just found
10,000 ways that won't work.
—Thomas Edison

There are many ways to do design badly, and just a few ways to do it well. Because success in architectural design is complex and challenging, designers have been looking for ways to capture and reuse hard-won architectural knowledge. Architectural patterns and tactics are ways of capturing proven good design structures, so that they can be reused.

Architectural patterns have seen increased interest and attention, from both software practitioners and theorists, over the past 15 years or more. An architectural pattern

- is a package of design decisions that is found repeatedly in practice,
- has known properties that permit reuse, and
- describes a *class* of architectures.

Because patterns are (by definition) found in practice, one does not invent them; one discovers them. Cataloging patterns is akin to the job of a Linnaean botanist or zoologist: "discovering" patterns and describing their shared characteristics. And like the botanist, zoologist, or ecologist, the pattern cataloger strives to understand how the characteristics lead to different behaviors and different responses to environmental conditions. For this reason there will never be a complete list of patterns: patterns spontaneously emerge in reaction to environmental conditions, and as long as those conditions change, new patterns will emerge.

Architectural design seldom starts from first principles. Experienced architects typically think of creating an architecture as a process of selecting, tailoring, and combining patterns. The software architect must decide how to instantiate a pattern—how to make it fit with the specific context and the constraints of the problem.

In Chapters 5–11 we have seen a variety of architectural tactics. These are simpler than patterns. Tactics typically use just a single structure or computational mechanism, and they are meant to address a single architectural force. For this reason they give more precise control to an architect when making design decisions than patterns, which typically combine multiple design decisions into a package. Tactics are the "building blocks" of design, from which architectural patterns are created. Tactics are atoms and patterns are molecules. Most patterns consist of (are constructed from) several different tactics. For this reason we say that patterns package tactics.

In this chapter we will take a very brief tour through the patterns universe, touching on some of the most important and most commonly used patterns for architecture, and we will then look at the relationships between patterns and tactics: showing how a pattern is constructed from tactics, and showing how tactics can be used to tailor patterns when the pattern that you find in a book or on a website doesn't quite address your design needs.

13.1 Architectural Patterns

An architectural pattern establishes a relationship between:

- *A context*. A recurring, common situation in the world that gives rise to a problem.
- *A problem.* The problem, appropriately generalized, that arises in the given context. The pattern description outlines the problem and its variants, and describes any complementary or opposing forces. The description of the problem often includes quality attributes that must be met.
- *A solution.* A successful architectural resolution to the problem, appropriately abstracted. The solution describes the architectural structures that solve the problem, including how to balance the many forces at work. The solution will describe the responsibilities of and static relationships among elements (using a module structure), or it will describe the runtime behavior of and interaction between elements (laying out a component-and-connector or allocation structure). The solution for a pattern is determined and described by:

  - A set of element types (for example, data repositories, processes, and objects)
  - A set of interaction mechanisms or connectors (for example, method calls, events, or message bus)
  - A topological layout of the components
  - A set of semantic constraints covering topology, element behavior, and interaction mechanisms

The solution description should also make clear what quality attributes are provided by the static and runtime configurations of elements.

This {*context*, *problem*, *solution*} form constitutes a template for documenting a pattern.

Complex systems exhibit multiple patterns at once. A web-based system might employ a three-tier client-server architectural pattern, but within this pattern it might also use replication (mirroring), proxies, caches, firewalls, MVC, and so forth, each of which may employ more patterns and tactics. And all of these parts of the client-server pattern likely employ layering to internally structure their software modules.

13.2 Overview of the Patterns Catalog

In this section we list an assortment of useful and widely used patterns. This catalog is not meant to be exhaustive—in fact no such catalog is possible. Rather it is meant to be representative. We show patterns of runtime elements (such as broker or client-server) and of design-time elements (such as layers). For each pattern we list the *context*, *problem*, and *solution*. As part of the solution, we briefly describe the *elements*, *relations*, and *constraints* of each pattern.

Applying a pattern is not an all-or-nothing proposition. Pattern definitions given in catalogs are strict, but in practice architects may choose to violate them in small ways when there is a good design tradeoff to be had (sacrificing a little of whatever the violation cost, but gaining something that the deviation gained). For example, the layered pattern expressly forbids software in lower layers from using software in upper layers, but there may be cases (such as to gain some performance) when an architecture might allow a few specific exceptions.

Patterns can be categorized by the dominant type of elements that they show: module patterns show modules, component-and-connector (C&C) patterns show components and connectors, and allocation patterns show a combination of software elements (modules, components, connectors) and nonsoftware elements. Most published patterns are C&C patterns, but there are module patterns and allocation patterns as well. We'll begin with the granddaddy of module patterns, the layered pattern.

Module Patterns

Layered Pattern

Context: All complex systems experience the need to develop and evolve portions of the system independently. For this reason the developers of the system need a clear and well-documented separation of concerns, so that modules of the system may be independently developed and maintained.

Problem: The software needs to be segmented in such a way that the modules can be developed and evolved separately with little interaction among the parts, supporting portability, modifiability, and reuse.

Solution: To achieve this separation of concerns, the layered pattern divides the software into units called layers. Each layer is a grouping of modules that offers a cohesive set of services. There are constraints on the *allowed-to-use* relationship among the layers: the relations must be unidirectional. Layers completely partition a set of software, and each partition is exposed through a public interface. The layers are created to interact according to a strict ordering relation. If (A,B) is in this relation, we say that the implementation of layer A is allowed to use any of the public facilities provided by layer B. In some cases, modules in one layer might be required to directly use modules in a nonadjacent lower layer; normally only next-lower-layer uses are allowed. This case of software in a higher layer using modules in a nonadjacent lower layer is called *layer bridging*. If many instances of layer bridging occur, the system may not meet its portability and modifiability goals that strict layering helps to achieve. Upward usages are not allowed in this pattern.

Of course, none of this comes for free. Someone must design and build the layers, which can often add up-front cost and complexity to a system. Also, if the layering is not designed correctly, it may actually get in the way, by not providing the lower-level abstractions that programmers at the higher levels need. And layering always adds a performance penalty to a system. If a call is made to a function in the top-most layer, this may have to traverse many lower layers before being executed by the hardware. Each of these layers adds some overhead of their own, at minimum in the form of context switching.

Table 13.1 summarizes the solution of the layered pattern.

Layers are almost always drawn as a stack of boxes. The *allowed-to-use* relation is denoted by geometric adjacency and is read from the top down, as in Figure 13.1.

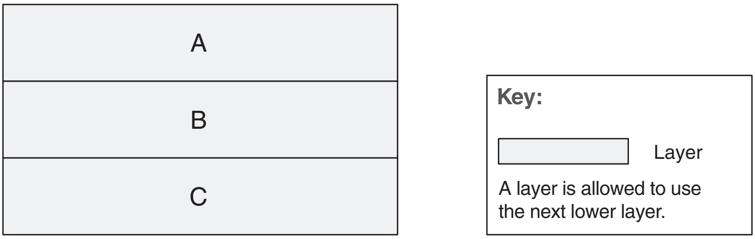


FIGURE 13.1 Stack-of-boxes notation for layered designs

TABLE 13.1 Layered Pattern Solution

Overview	The layered pattern defines layers (groupings of modules that offer a cohesive set of services) and a unidirectional *allowed-to-use* relation among the layers. The pattern is usually shown graphically by stacking boxes representing layers on top of each other.
Elements	*Layer*, a kind of module. The description of a layer should define what modules the layer contains and a characterization of the cohesive set of services that the layer provides.
Relations	*Allowed to use*, which is a specialization of a more generic *depends-on* relation. The design should define what the layer usage rules are (e.g., "a layer is allowed to use any lower layer" or "a layer is allowed to use only the layer immediately below it") and any allowable exceptions.
Constraints	▪ Every piece of software is allocated to exactly one layer. ▪ There are at least two layers (but usually there are three or more). ▪ The *allowed-to-use* relations should not be circular (i.e., a lower layer cannot use a layer above).
Weaknesses	▪ The addition of layers adds up-front cost and complexity to a system. ▪ Layers contribute a performance penalty.

Some Finer Points of Layers

A layered architecture is one of the few places where connections among components can be shown by adjacency, and where "above" and "below" matter. If you turn Figure 13.1 upside-down so that C is on top, this would represent a completely different design. Diagrams that use arrows among the boxes to denote relations retain their semantic meaning no matter the orientation.

The layered pattern is one of the most commonly used patterns in all of software engineering, but I'm often surprised by how many people still get it wrong.

First, it is impossible to look at a stack of boxes and tell whether layer bridging is allowed or not. That is, can a layer use any lower layer, or just the next lower one? It is the easiest thing in the world to resolve this; all the architect has to do is include the answer in the key to the diagram's notation (something we recommend for all diagrams). For example, consider the layered pattern presented in Figure 13.2 on the next page.

But I'm still surprised at how few architects actually bother to do this. And if they don't, their layer diagrams are ambiguous.

Second, any old set of boxes stacked on top of each other does not constitute a layered architecture. For instance, look at the design shown in Figure 13.3, which uses arrows instead of adjacency to indicate the relationships among the boxes. Here, everything is allowed to use everything. This is decidedly *not* a layered architecture. The reason is that if Layer A is replaced by a different version, Layer C (which uses it in this figure) might well have to change. We don't want our virtual machine layer to change every time our application layer changes. But I'm still surprised at how many people call a stack of boxes lined up with each other "layers" (or think that layers are the same as tiers in a multi-tier architecture).

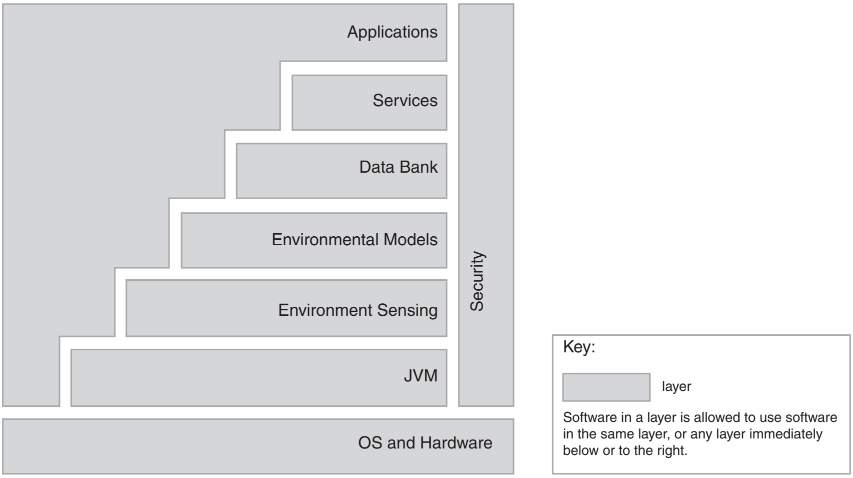

Applications
Services
Data Bank
Environmental Models
Environment Sensing
JVM
Security
OS and Hardware
Key:
layer
Software in a layer is allowed to use software in the same layer, or any layer immediately below or to the right.

FIGURE 13.2 A simple layer diagram, with a simple key answering the uses question

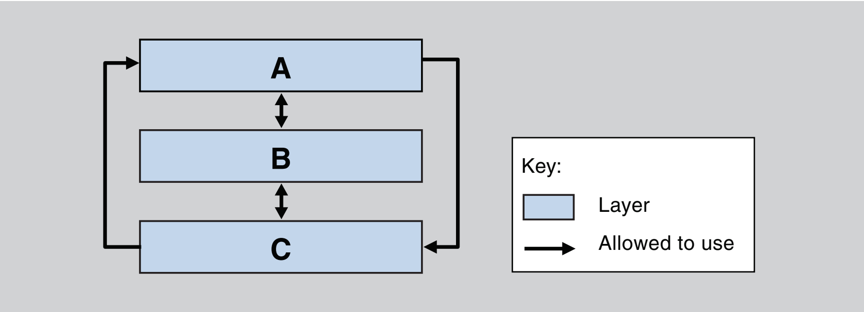


FIGURE 13.3 A wolf in layer's clothing

Third, many architectures that purport to be layered look something like Figure 13.4. This diagram *probably* means that modules in A, B, or C can use modules in D, but without a key to tell us for sure, it could mean anything. “Sidecars” like this often contain common utilities (sometimes imported), such as error handlers, communication protocols, or database access mechanisms. This kind of diagram makes sense only in the case where no layer bridging is allowed in the main stack. Otherwise, D could simply be made the bottom-most layer in the main stack, and the “sidecar” geometry would be unnecessary. But I’m still surprised at how often I see this layout go unexplained.

Sometimes layers are divided into segments denoting a finer-grained decomposition of the modules. Sometimes this occurs when a preexisting set of units, such as imported modules, share the same *allowed-to-use* relation. When this happens, you have to specify what usage rules are in effect among the segments. Many usage rules are possible, but they must be made explicit. In Figure 13.5, the top and the bottom layers are

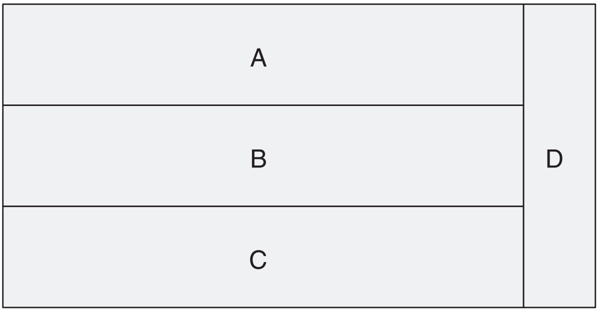


FIGURE 13.4 Layers with a “sidecar”

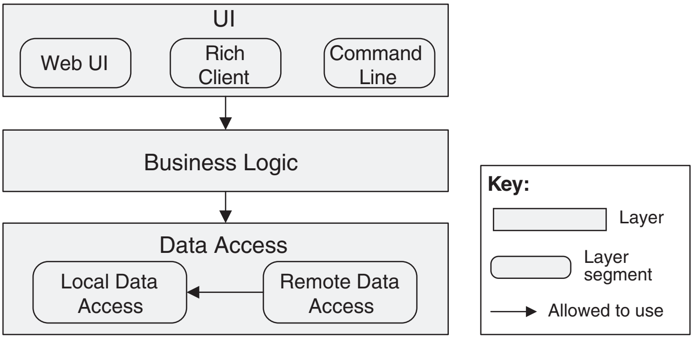


FIGURE 13.5 Layered design with segmented layers

segmented. Segments of the top layer are not allowed to use each other, but segments of the bottom layer are. If you draw the same diagram without the arrows, it will be harder to differentiate the different usage rules within segmented layers. Layered diagrams are often a source of hidden ambiguity because the diagram does not make explicit the *allowed-to-use* relations.

Finally, the most important point about layering is that a layer isn't allowed to *use* any layer above it. A module "uses" another module when it depends on the answer it gets back. But a layer is allowed to make upward calls, as long as it isn't expecting an answer from them. This is how the common error-handling scheme of callbacks works. A program in layer A calls a program in a lower layer B, and the parameters include a pointer to an error-handling program in A that the lower layer should call in case of error. The software in B makes the call to the program in A, but cares not in the least what it does. By not depending in any way on the contents of A, B is insulated from changes in A.

—PCC

Other Module Patterns

Designers in a particular domain often publish "standard" module decompositions for systems in that domain. These standard decompositions, if put in the "context, problem, solution" form, constitute module decomposition patterns.

Similarly in the object-oriented realm, "standard" or published class/object design solutions for a class of system constitute object-oriented patterns.

Component-and-Connector Patterns

Broker Pattern

Context: Many systems are constructed from a collection of services distributed across multiple servers. Implementing these systems is complex because you need to worry about how the systems will interoperate—how they will connect to each other and how they will exchange information—as well as the availability of the component services.

Problem: How do we structure distributed software so that service users do not need to know the nature and location of service providers, making it easy to dynamically change the bindings between users and providers?

Solution: The broker pattern separates users of services (clients) from providers of services (servers) by inserting an intermediary, called a broker. When a client needs a service, it queries a broker via a service interface. The broker then forwards the client's service request to a server, which processes the request. The service result is communicated from the server back to the broker, which then returns the result (and any exceptions) back to the requesting client. In this way the client remains completely ignorant of the identity, location, and characteristics of the server. Because of this separation, if a server becomes unavailable, a replacement can be dynamically chosen by the broker. If a server is replaced with a different (compatible) service, again, the broker is the only component that needs to know of this change, and so the client is unaffected. Proxies are commonly introduced as intermediaries in addition to the broker to help with details of the interaction with the broker, such as marshaling and unmarshaling messages.

The down sides of brokers are that they add complexity (brokers and possibly proxies must be designed and implemented, along with messaging protocols) and add a level of indirection between a client and a server, which will add latency to their communication. Debugging brokers can be difficult because they are involved in highly dynamic environments where the conditions leading to a failure may be difficult to replicate. The broker would be an obvious point of attack, from a security perspective, and so it needs to be hardened appropriately. Also a broker, if it is not designed carefully, can be a single point of failure for a large and complex system. And brokers can potentially be bottlenecks for communication.

Table 13.2 summarizes the solution of the broker pattern.

TABLE 13.2 Broker Pattern Solution

Overview	The broker pattern defines a runtime component, called a broker, that mediates the communication between a number of clients and servers.
Elements	*Client,* a requester of services *Server,* a provider of services *Broker,* an intermediary that locates an appropriate server to fulfill a client's request, forwards the request to the server, and returns the results to the client *Client-side proxy,* an intermediary that manages the actual communication with the broker, including marshaling, sending, and unmarshaling of messages *Server-side proxy,* an intermediary that manages the actual communication with the broker, including marshaling, sending, and unmarshaling of messages
Relations	The *attachment* relation associates clients (and, optionally, client-side proxies) and servers (and, optionally, server-side proxies) with brokers.
Constraints	The client can only attach to a broker (potentially via a client-side proxy). The server can only attach to a broker (potentially via a server-side proxy).
Weaknesses	Brokers add a layer of indirection, and hence latency, between clients and servers, and that layer may be a communication bottleneck. The broker can be a single point of failure. A broker adds up-front complexity. A broker may be a target for security attacks. A broker may be difficult to test.

The broker is, of course, the critical component in this pattern. The pattern provides all of the modifiability benefits of the use-an-intermediary tactic (described in Chapter 7), an availability benefit (because the broker pattern makes it easy to replace a failed server with another), and a performance benefit (because the broker pattern makes it easy to assign work to the least-busy server). However, the pattern also carries with it some liabilities. For example, the use of a broker precludes performance optimizations that you might make if you knew the precise location and characteristics of the server. Also the use of this pattern adds the overhead of the intermediary and thus latency.

The original version of the broker pattern, as documented by Gamma, Helm, Johnson, and Vlissides [Gamma 94], is given in Figure 13.6.

The first widely used implementation of the broker pattern was in the Common Object Request Broker Architecture (CORBA). Other common uses of this pattern are found in Enterprise Java Beans (EJB) and Microsoft's .NET platform—essentially any modern platform for distributed service providers and consumers implements some form of a broker. The service-oriented architecture (SOA) approach depends crucially on brokers, most commonly in the form of an enterprise service bus.

Model-View-Controller Pattern

Context: User interface software is typically the most frequently modified portion of an interactive application. For this reason it is important to keep modifications

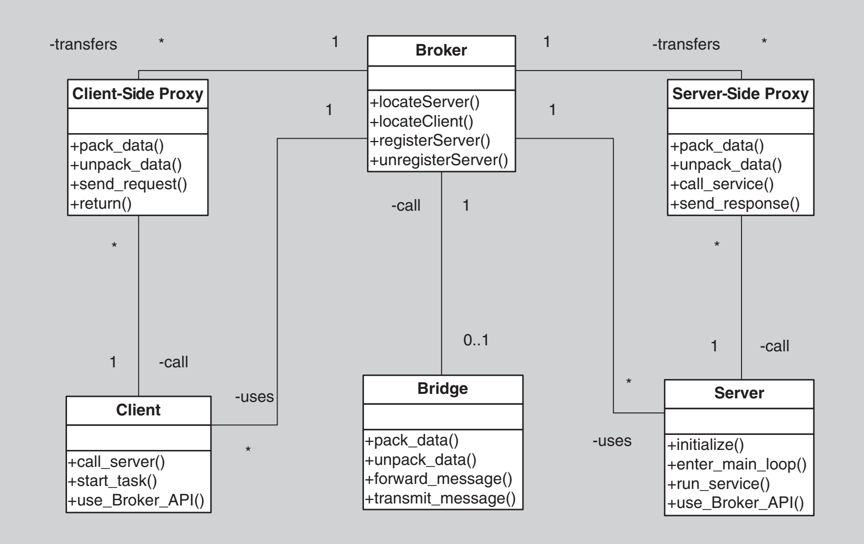


FIGURE 13.6 The broker pattern

to the user interface software separate from the rest of the system. Users often wish to look at data from different perspectives, such as a bar graph or a pie chart. These representations should both reflect the current state of the data.

Problem: How can user interface functionality be kept separate from application functionality and yet still be responsive to user input, or to changes in the underlying application's data? And how can multiple views of the user interface be created, maintained, and coordinated when the underlying application data changes?

Solution: The model-view-controller (MVC) pattern separates application functionality into three kinds of components:

- A *model*, which contains the application's data
- A *view*, which displays some portion of the underlying data and interacts with the user
- A *controller*, which mediates between the *model* and the *view* and manages the notifications of state changes

MVC is not appropriate for every situation. The design and implementation of three distinct kinds of components, along with their various forms of interaction, may be costly, and this cost may not make sense for relatively simple user interfaces. Also, the match between the abstractions of MVC and commercial user interface toolkits is not perfect. The view and the controller split apart input and output, but these functions are often combined into individual widgets. This may result in a conceptual mismatch between the architecture and the user interface toolkit.

Table 13.3 summarizes the solution of the MVC pattern.

TABLE 13.3 Model-View-Controller Pattern Solution

Overview	The MVC pattern breaks system functionality into three components: a model, a view, and a controller that mediates between the model and the view.
Elements	The *model* is a representation of the application data or state, and it contains (or provides an interface to) application logic. The *view* is a user interface component that either produces a representation of the model for the user or allows for some form of user input, or both. The *controller* manages the interaction between the model and the view, translating user actions into changes to the model or changes to the view.
Relations	The *notifies* relation connects instances of model, view, and controller, notifying elements of relevant state changes.
Constraints	There must be at least one instance each of model, view, and controller. The model component should not interact directly with the controller.
Weaknesses	The complexity may not be worth it for simple user interfaces. The model, view, and controller abstractions may not be good fits for some user interface toolkits.

There may, in fact, be many views and many controllers associated with a model. For example, a set of business data may be represented as columns of numbers in a spreadsheet, as a scatter plot, or as a pie chart. Each of these is a separate view, and this view can be dynamically updated as the model changes (for example, showing live transactions in a transaction processing system). A model may be updated by different controllers; for example, a map could be zoomed and panned via mouse movements, trackball movements, keyboard clicks, or voice commands; each of these different forms of input needs to be managed by a controller.

The MVC components are connected to each other via some flavor of notification, such as events or callbacks. These notifications contain state updates. A change in the model needs to be communicated to the views so that they may be updated. An external event, such as a user input, needs to be communicated to the controller, which may in turn update the view and/or the model. Notifications may be either push or pull.

Because these components are loosely coupled, it is easy to develop and test them in parallel, and changes to one have minimal impact on the others. The relationships between the components of MVC are shown in Figure 13.7.

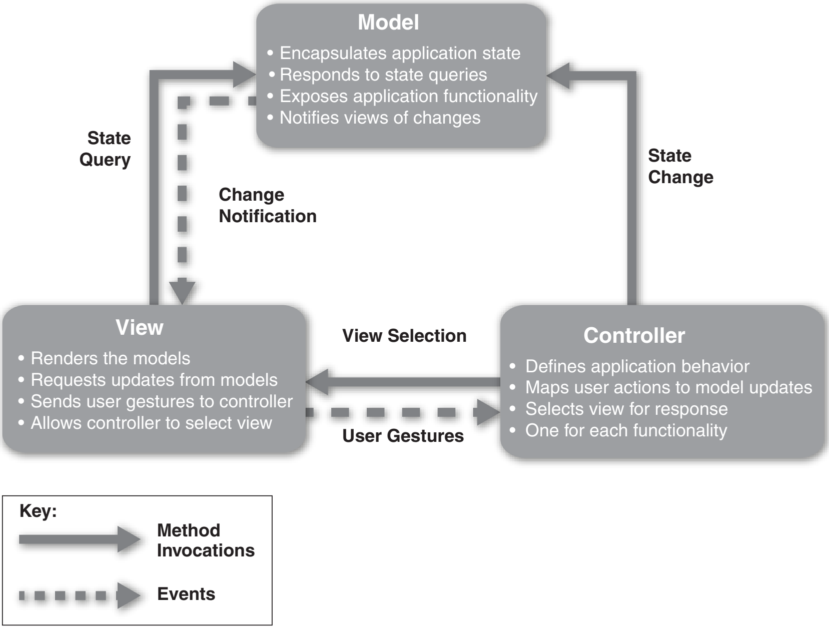

Model
• Encapsulates application state
• Responds to state queries
• Exposes application functionality
• Notifies views of changes
State Query
Change Notification
State Change
View
• Renders the models
• Requests updates from models
• Sends user gestures to controller
• Allows controller to select view
View Selection
User Gestures
Controller
• Defines application behavior
• Maps user actions to model updates
• Selects view for response
• One for each functionality
Key:
Method Invocations
Events

FIGURE 13.7 The model-view-controller pattern

The MVC pattern is widely used in user interface libraries such as Java's Swing classes, Microsoft's ASP.NET framework, Adobe's Flex software development kit, Nokia's Qt framework, and many others. As such, it is common for a single application to contain many instances of MVC (often one per user interface object).

Pipe-and-Filter Pattern

Context: Many systems are required to transform streams of discrete data items, from input to output. Many types of transformations occur repeatedly in practice, and so it is desirable to create these as independent, reusable parts.

Problem: Such systems need to be divided into reusable, loosely coupled components with simple, generic interaction mechanisms. In this way they can be flexibly combined with each other. The components, being generic and loosely coupled, are easily reused. The components, being independent, can execute in parallel.

Solution: The pattern of interaction in the pipe-and-filter pattern is characterized by successive transformations of streams of data. Data arrives at a filter's input port(s), is transformed, and then is passed via its output port(s) through a pipe to the next filter. A single filter can consume data from, or produce data to, one or more ports.

There are several weaknesses associated with the pipe-and-filter pattern. For instance, this pattern is typically not a good choice for an interactive system, as it disallows cycles (which are important for user feedback). Also, having large numbers of independent filters can add substantial amounts of computational overhead, because each filter runs as its own thread or process. Also, pipe-and-filter systems may not be appropriate for long-running computations, without the addition of some form of checkpoint/restore functionality, as the failure of any filter (or pipe) can cause the entire pipeline to fail.

The solution of the pipe-and-filter pattern is summarized in Table 13.4.

Pipes buffer data during communication. Because of this property, filters can execute asynchronously and concurrently. Moreover, a filter typically does not know the identity of its upstream or downstream filters. For this reason, pipeline pipe-and-filter systems have the property that the overall computation can be treated as the functional composition of the computations of the filters, making it easier for the architect to reason about end-to-end behavior.

Data transformation systems are typically structured as pipes and filters, with each filter responsible for one part of the overall transformation of the input data. The independent processing at each step supports reuse, parallelization, and simplified reasoning about overall behavior. Often such systems constitute the front end of signal-processing applications. These systems receive sensor data at a set of initial filters; each of these filters compresses the data and performs initial processing (such as smoothing). Downstream filters reduce the data further and do synthesis across data derived from different sensors. The final filter typically passes its data to an application, for example providing input to modeling or visualization tools.

Other systems that use pipe-and-filter include those built using UNIX pipes, the request processing architecture of the Apache web server, the map-reduce pattern (presented later in this chapter), Yahoo! Pipes for processing RSS feeds, many workflow engines, and many scientific computation systems that have to process and analyze large streams of captured data. Figure 13.8 shows a UML diagram of a pipe-and-filter system.

TABLE 13.4 Pipe-and-Filter Pattern Solution

Overview	Data is transformed from a system's external inputs to its external outputs through a series of transformations performed by its filters connected by pipes.
Elements	*Filter,* which is a component that transforms data read on its input port(s) to data written on its output port(s). Filters can execute concurrently with each other. Filters can incrementally transform data; that is, they can start producing output as soon as they start processing input. Important characteristics include processing rates, input/output data formats, and the transformation executed by the filter. *Pipe,* which is a connector that conveys data from a filter's output port(s) to another filter's input port(s). A pipe has a single source for its input and a single target for its output. A pipe preserves the sequence of data items, and it does not alter the data passing through. Important characteristics include buffer size, protocol of interaction, transmission speed, and format of the data that passes through a pipe.
Relations	The *attachment* relation associates the output of filters with the input of pipes and vice versa.
Constraints	Pipes connect filter output ports to filter input ports. Connected filters must agree on the type of data being passed along the connecting pipe. Specializations of the pattern may restrict the association of components to an acyclic graph or a linear sequence, sometimes called a pipeline. Other specializations may prescribe that components have certain named ports, such as the *stdin*, *stdout*, and *stderr* ports of UNIX filters.
Weaknesses	The pipe-and-filter pattern is typically not a good choice for an interactive system. Having large numbers of independent filters can add substantial amounts of computational overhead. Pipe-and-filter systems may not be appropriate for long-running computations.

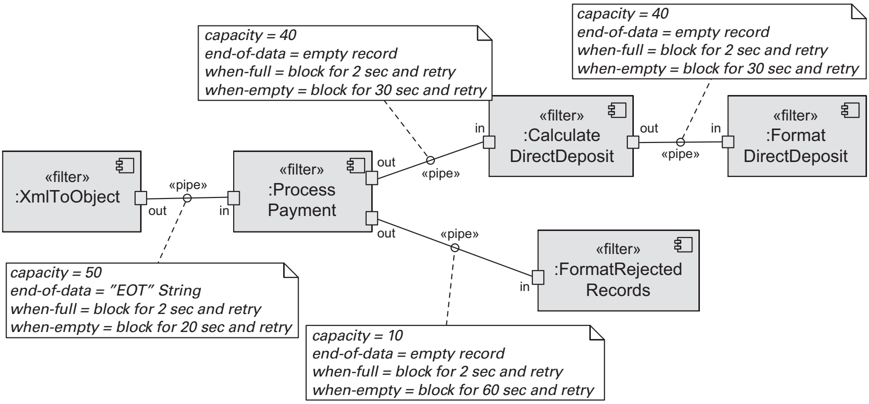

capacity = 40
end-of-data = empty record
when-full = block for 2 sec and retry
when-empty = block for 30 sec and retry
capacity = 40
end-of-data = empty record
when-full = block for 2 sec and retry
when-empty = block for 30 sec and retry
«filter»
:Calculate
DirectDeposit
«filter»
:Format
DirectDeposit
in
out
in
«pipe»
out
«pipe»
«filter»
:XmlToObject
«pipe»
out
in
«filter»
:Process
Payment
out
«pipe»
«filter»
:FormatRejected
Records
in
capacity = 50
end-of-data = "EOT" String
when-full = block for 2 sec and retry
when-empty = block for 20 sec and retry
capacity = 10
end-of-data = empty record
when-full = block for 2 sec and retry
when-empty = block for 60 sec and retry

FIGURE 13.8 A UML diagram of a pipe-and-filter-based system

Client-Server Pattern

Context: There are shared resources and services that large numbers of distributed clients wish to access, and for which we wish to control access or quality of service.

Problem: By managing a set of shared resources and services, we can promote modifiability and reuse, by factoring out common services and having to modify these in a single location, or a small number of locations. We want to improve scalability and availability by centralizing the control of these resources and services, while distributing the resources themselves across multiple physical servers.

Solution: Clients interact by requesting services of servers, which provide a set of services. Some components may act as both clients and servers. There may be one central server or multiple distributed ones.

The client-server pattern solution is summarized in Table 13.5; the component types are *clients* and *servers*; the principal connector type for the client-server pattern is a data connector driven by a request/reply protocol used for invoking services.

Some of the disadvantages of the client-server pattern are that the server can be a performance bottleneck and it can be a single point of failure. Also, decisions about where to locate functionality (in the client or in the server) are often complex and costly to change after a system has been built.

TABLE 13.5 Client-Server Pattern Solution

Overview	Clients initiate interactions with servers, invoking services as needed from those servers and waiting for the results of those requests.
Elements	*Client,* a component that invokes services of a server component. Clients have ports that describe the services they require. *Server,* a component that provides services to clients. Servers have ports that describe the services they provide. Important characteristics include information about the nature of the server ports (such as how many clients can connect) and performance characteristics (e.g., maximum rates of service invocation). *Request/reply connector,* a data connector employing a request/reply protocol, used by a client to invoke services on a server. Important characteristics include whether the calls are local or remote, and whether data is encrypted.
Relations	The *attachment* relation associates clients with servers.
Constraints	Clients are connected to servers through request/reply connectors. Server components can be clients to other servers. Specializations may impose restrictions: ▪ Numbers of attachments to a given port ▪ Allowed relations among servers Components may be arranged in tiers, which are logical groupings of related functionality or functionality that will share a host computing environment (covered more later in this chapter).
Weaknesses	Server can be a performance bottleneck. Server can be a single point of failure. Decisions about where to locate functionality (in the client or in the server) are often complex and costly to change after a system has been built.

Some common examples of systems that use the client-server pattern are these:

- Information systems running on local networks where the clients are GUI-launched applications and the server is a database management system
- Web-based applications where the clients are web browsers and the servers are components running on an e-commerce site

The computational flow of pure client-server systems is asymmetric: clients initiate interactions by invoking services of servers. Thus, the client must know the identity of a service to invoke it, and clients initiate all interactions. In contrast, servers do not know the identity of clients in advance of a service request and must respond to the initiated client requests.

In early forms of client-server, service invocation is synchronous: the requester of a service waits, or is blocked, until a requested service completes its actions, possibly providing a return result. However, variants of the client-server pattern may employ more-sophisticated connector protocols. For example:

- Web browsers don't block until the data request is served up.
- In some client-server patterns, servers are permitted to initiate certain actions on their clients. This might be done by allowing a client to register notification procedures, or callbacks, that the server calls at specific times.
- In other systems service calls over a request/reply connector are bracketed by a "session" that delineates the start and end of a set of a client-server interaction.

The client-server pattern separates client applications from the services they use. This pattern simplifies systems by factoring out common services, which are reusable. Because servers can be accessed by any number of clients, it is easy to add new clients to a system. Similarly, servers may be replicated to support scalability or availability.

The World Wide Web is the best-known example of a system that is based on the client-server pattern, allowing clients (web browsers) to access information from servers across the Internet using HyperText Transfer Protocol (HTTP). HTTP is a request/reply protocol. HTTP is stateless; the connection between the client and the server is terminated after each response from the server.

Figure 13.9 uses an informal notation to describe the client-server view of an automatic teller machine (ATM) banking system.

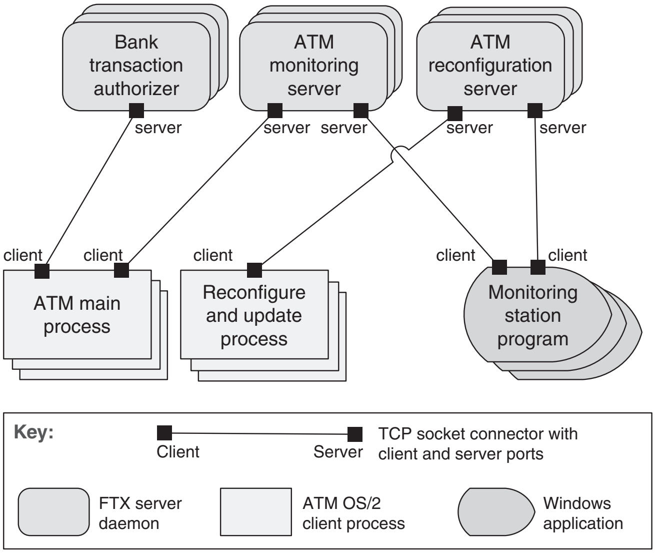


FIGURE 13.9 The client-server architecture of an ATM banking system

Peer-to-Peer Pattern

Context: Distributed computational entities—each of which is considered equally important in terms of initiating an interaction and each of which provides its own resources—need to cooperate and collaborate to provide a service to a distributed community of users.

Problem: How can a set of "equal" distributed computational entities be connected to each other via a common protocol so that they can organize and share their services with high availability and scalability?

Solution: In the peer-to-peer (P2P) pattern, components directly interact as peers. All peers are "equal" and no peer or group of peers can be critical for the health of the system. Peer-to-peer communication is typically a request/reply interaction without the asymmetry found in the client-server pattern. That is, any component can, in principle, interact with any other component by requesting its services. The interaction may be initiated by either party—that is, in client-server terms, each peer component is both a client and a server. Sometimes the interaction is just to forward data without the need for a reply. Each peer provides and consumes similar services and uses the same protocol. Connectors in peer-to-peer systems involve bidirectional interactions, reflecting the two-way communication that may exist between two or more peer-to-peer components.

Peers first connect to the peer-to-peer network on which they discover other peers they can interact with, and then initiate actions to achieve their computation by cooperating with other peers by requesting services. Often a peer's search for another peer is propagated from one peer to its connected peers for a limited number of hops. A peer-to-peer architecture may have specialized peer nodes (called supernodes) that have indexing or routing capabilities and allow a regular peer's search to reach a larger number of peers.

Peers can be added and removed from the peer-to-peer network with no significant impact, resulting in great scalability for the whole system. This provides flexibility for deploying the system across a highly distributed platform.

Typically multiple peers have overlapping capabilities, such as providing access to the same data or providing equivalent services. Thus, a peer acting as client can collaborate with multiple peers acting as servers to complete a certain task. If one of these multiple peers becomes unavailable, the others can still provide the services to complete the task. The result is improved overall availability. There are also performance advantages: The load on any given peer component acting as a server is reduced, and the responsibilities that might have required more server capacity and infrastructure to support it are distributed. This can decrease the need for other communication for updating data and for central server storage, but at the expense of storing the data locally.

The drawbacks of the peer-to-peer pattern are strongly related to its strengths. Because peer-to-peer systems are decentralized, managing security, data consistency, data and service availability, backup, and recovery are all more complex. In many cases it is difficult to provide guarantees with peer-to-peer systems because the peers come and go; instead, the architect can, at best, offer probabilities that quality goals will be met, and these probabilities typically increase with the size of the population of peers.

Table 13.6 on the next page summarizes the peer-to-peer pattern solution.

Peer-to-peer computing is often used in distributed computing applications such as file sharing, instant messaging, desktop grid computing, routing, and wireless ad hoc networking. Examples of peer-to-peer systems include file-sharing networks such as BitTorrent and eDonkey, and instant messaging and VoIP applications such as Skype. Figure 13.10 shows an example of an instantiation of the peer-to-peer pattern.

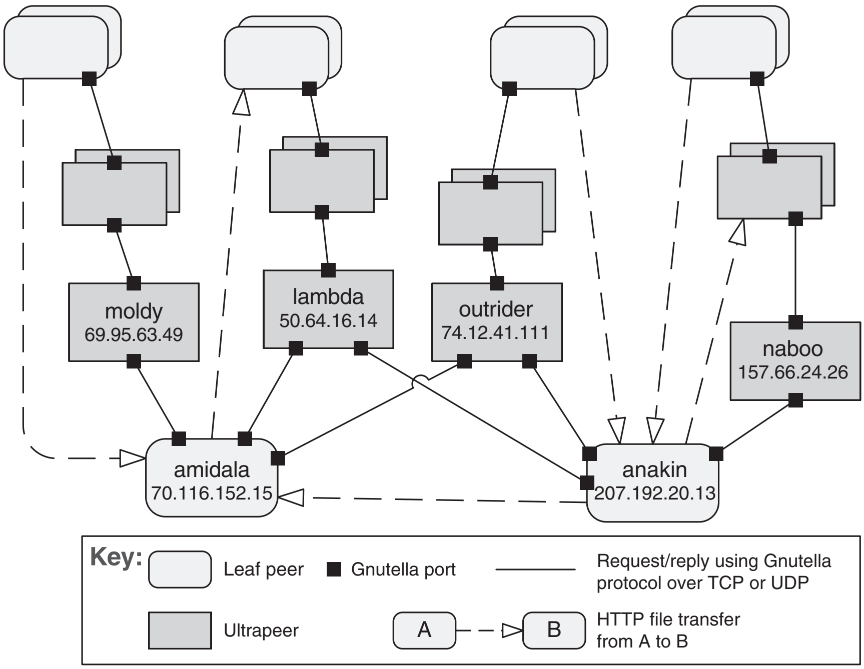


FIGURE 13.10 A peer-to-peer view of a Gnutella network using an informal C&C notation. For brevity, only a few peers are identified. Each of the identified leaf peers uploads and downloads files directly from other peers.

TABLE 13.6 Peer-to-Peer Pattern Solution

Overview	Computation is achieved by cooperating peers that request service from and provide services to one another across a network.
Elements	*Peer,* which is an independent component running on a network node. Special peer components can provide routing, indexing, and peer search capability. *Request/reply connector,* which is used to connect to the peer network, search for other peers, and invoke services from other peers. In some cases, the need for a reply is done away with.
Relations	The relation associates peers with their connectors. Attachments may change at runtime.
Constraints	Restrictions may be placed on the following: ▪ The number of allowable attachments to any given peer ▪ The number of hops used for searching for a peer ▪ Which peers know about which other peers Some P2P networks are organized with star topologies, in which peers only connect to supernodes.
Weaknesses	Managing security, data consistency, data/service availability, backup, and recovery are all more complex. Small peer-to-peer systems may not be able to consistently achieve quality goals such as performance and availability.

Service-Oriented Architecture Pattern

Context: A number of services are offered (and described) by service providers and consumed by service consumers. Service consumers need to be able to understand and use these services without any detailed knowledge of their implementation.

Problem: How can we support interoperability of distributed components running on different platforms and written in different implementation languages, provided by different organizations, and distributed across the Internet? How can we locate services and combine (and dynamically recombine) them into meaningful coalitions while achieving reasonable performance, security, and availability?

Solution: The service-oriented architecture (SOA) pattern describes a collection of distributed components that provide and/or consume services. In an SOA, *service provider* components and *service consumer* components can use different implementation languages and platforms. Services are largely standalone: service providers and service consumers are usually deployed independently, and often belong to different systems or even different organizations. Components have interfaces that describe the services they request from other components and the services they provide. A service's quality attributes can be specified and guaranteed with a service-level agreement (SLA). In some cases, these are legally binding. Components achieve their computation by requesting services from one another.

The elements in this pattern include service providers and service consumers, which in practice can take different forms, from JavaScript running on a web browser to CICS transactions running on a mainframe. In addition to the service provider and service consumer components, an SOA application may use specialized components that act as intermediaries and provide infrastructure services:

- Service invocation can be mediated by an *enterprise service bus* (ESB). An ESB routes messages between service consumers and service providers. In addition, an ESB can convert messages from one protocol or technology to another, perform various data transformations (e.g., format, content, splitting, merging), perform security checks, and manage transactions. Using an ESB promotes interoperability, security, and modifiability. Of course, communicating through an ESB adds overhead thereby lowering performance, and introduces an additional point of failure. When an ESB is not in place, service providers and consumers communicate with each other in a point-to-point fashion.
- To improve the independence of service providers, a *service registry* can be used in SOA architectures. The registry is a component that allows services to be registered at runtime. This enables runtime discovery of services, which increases system modifiability by hiding the location and identity of the service provider. A registry can even permit multiple live versions of the same service.
- An *orchestration server* (or orchestration engine) orchestrates the interaction among various service consumers and providers in an SOA system. It executes scripts upon the occurrence of a specific event (e.g., a purchase order request arrived). Applications with well-defined business processes or workflows that involve interactions with distributed components or systems gain in modifiability, interoperability, and reliability by using an orchestration server. Many commercially available orchestration servers support various workflow or business process language standards.

The basic types of connectors used in SOA are these:

- *SOAP*. The standard protocol for communication in the web services technology. Service consumers and providers interact by exchanging request/reply XML messages typically on top of HTTP.
- *Representational State Transfer (REST)*. A service consumer sends non-blocking HTTP requests. These requests rely on the four basic HTTP commands (POST, GET, PUT, DELETE) to tell the service provider to create, retrieve, update, or delete a resource.
- *Asynchronous messaging*, a "fire-and-forget" information exchange. Participants do not have to wait for an acknowledgment of receipt, because the infrastructure is assumed to have delivered the message successfully. The messaging connector can be point-to-point or publish-subscribe.

In practice, SOA environments may involve a mix of the three connectors just listed, along with legacy protocols and other communication alternatives (e.g., SMTP). Commercial products such as IBM's WebSphere MQ, Microsoft's MSMQ, or Apache's ActiveMQ are infrastructure components that provide asynchronous messaging. SOAP and REST are described in more detail in Chapter 6.

As you can see, the SOA pattern can be quite complex to design and implement (due to dynamic binding and the concomitant use of metadata). Other potential problems with this pattern include the performance overhead of the middleware that is interposed between services and clients and the lack of performance guarantees (because services are shared and, in general, not under control of the requester). These weaknesses are all shared with the broker pattern, which is not surprising because the SOA pattern shares many of the design concepts and goals of broker. In addition, because you do not, in general, control the evolution of the services that you use, you may have to endure high and unplanned-for maintenance costs.

Table 13.7 summarizes the SOA pattern.

The main benefit and the major driver of SOA is interoperability. Because service providers and service consumers may run on different platforms, service-oriented architectures often integrate a variety of systems, including legacy systems. SOA also offers the necessary elements to interact with external services available over the Internet. Special SOA components such as the registry or the ESB also allow dynamic reconfiguration, which is useful when there's a need to replace or add versions of components with no system interruption.

Figure 13.11 shows the SOA view of a system called Adventure Builder. Adventure Builder allows a customer on the web to assemble a vacation by choosing an activity and lodging at and transportation to a destination. The Adventure Builder system interacts with external service providers to construct the vacation, and with bank services to process payment. The central OPC (Order Processing Center) component coordinates the interaction with internal and external service consumers and providers. Note that the external providers can be legacy mainframe systems, Java systems, .NET systems, and so on. The nature of these external components is transparent because SOAP provides the necessary interoperability.

TABLE 13.7 Service-Oriented Architecture Pattern Solution

Overview	Computation is achieved by a set of cooperating components that provide and/or consume services over a network. The computation is often described using a workflow language.
Elements	Components: ▪ *Service providers,* which provide one or more services through published interfaces. Concerns are often tied to the chosen implementation technology, and include performance, authorization constraints, availability, and cost. In some cases these properties are specified in a service-level agreement. ▪ *Service consumers,* which invoke services directly or through an intermediary. ▪ *Service providers* may also be service consumers. ▪ *ESB,* which is an intermediary element that can route and transform messages between service providers and consumers. ▪ *Registry of services,* which may be used by providers to register their services and by consumers to discover services at runtime. ▪ *Orchestration server,* which coordinates the interactions between service consumers and providers based on languages for business processes and workflows. Connectors: ▪ *SOAP connector,* which uses the SOAP protocol for synchronous communication between web services, typically over HTTP. ▪ *REST connector,* which relies on the basic request/reply operations of the HTTP protocol. ▪ *Asynchronous messaging connector,* which uses a messaging system to offer point-to-point or publish-subscribe asynchronous message exchanges.
Relations	Attachment of the different kinds of components available to the respective connectors
Constraints	Service consumers are connected to service providers, but intermediary components (e.g., ESB, registry, orchestration server) may be used.
Weaknesses	SOA-based systems are typically complex to build. You don't control the evolution of independent services. There is a performance overhead associated with the middleware, and services may be performance bottlenecks, and typically do not provide performance guarantees.

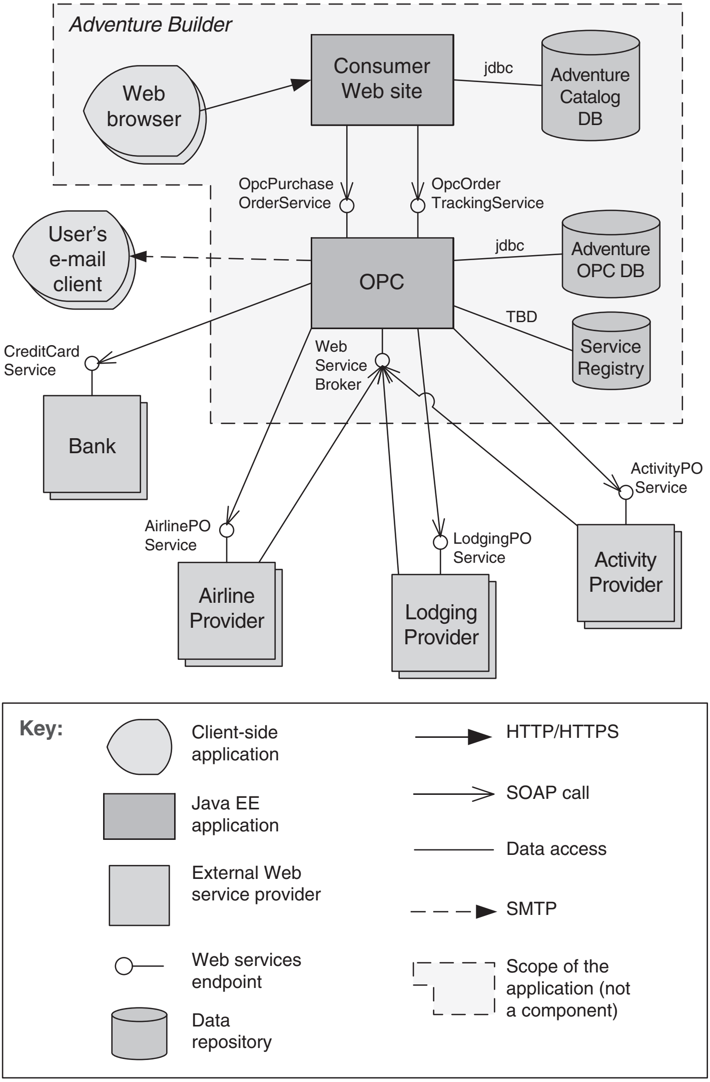


FIGURE 13.11 Diagram of the SOA view for the Adventure Builder system. OPC stands for "Order Processing Center."

Publish-Subscribe Pattern

Context: There are a number of independent producers and consumers of data that must interact. The precise number and nature of the data producers and consumers are not predetermined or fixed, nor is the data that they share.

Problem: How can we create integration mechanisms that support the ability to transmit messages among the producers and consumers in such a way that they are unaware of each other's identity, or potentially even their existence?

Solution: In the publish-subscribe pattern, summarized in Table 13.8, components interact via announced messages, or events. Components may subscribe to a set of events. It is the job of the publish-subscribe runtime infrastructure to make sure that each published event is delivered to all subscribers of that event. Thus, the main form of connector in these patterns is an *event bus*. Publisher components place events on the bus by announcing them; the connector then delivers those events to the subscriber components that have registered an interest in those events. Any component may be both a publisher and a subscriber.

Publish-subscribe adds a layer of indirection between senders and receivers. This has a negative effect on latency and potentially scalability, depending on how it is implemented. One would typically not want to use publish-subscribe in a system that had hard real-time deadlines to meet, as it introduces uncertainty in message delivery times.

Also, the publish-subscribe pattern suffers in that it provides less control over ordering of messages, and delivery of messages is not guaranteed (because the sender cannot know if a receiver is listening). This can make the publish-subscribe pattern inappropriate for complex interactions where shared state is critical.

TABLE 13.8 Publish-Subscribe Pattern Solution

Overview	Components publish and subscribe to events. When an event is announced by a component, the connector infrastructure dispatches the event to all registered subscribers.
Elements	*Any C&C component* with at least one publish or subscribe port. Concerns include which events are published and subscribed to, and the granularity of events. *The publish-subscribe connector*, which will have *announce* and *listen* roles for components that wish to publish and subscribe to events.
Relations	The *attachment* relation associates components with the publish-subscribe connector by prescribing which components announce events and which components are registered to receive events.
Constraints	All components are connected to an event distributor that may be viewed as either a bus—connector—or a component. Publish ports are attached to announce roles and subscribe ports are attached to listen roles. Constraints may restrict which components can listen to which events, whether a component can listen to its own events, and how many publish-subscribe connectors can exist within a system. A component may be both a publisher and a subscriber, by having ports of both types.
Weaknesses	Typically increases latency and has a negative effect on scalability and predictability of message delivery time. Less control over ordering of messages, and delivery of messages is not guaranteed.

There are some specific refinements of this pattern that are in common use. We will describe several of these later in this section.

The computational model for the publish-subscribe pattern is best thought of as a system of independent processes or objects, which react to events generated by their environment, and which in turn cause reactions in other components as a side effect of their event announcements. An example of the publish-subscribe pattern, implemented on top of the Eclipse platform, is shown in Figure 13.12.

Typical examples of systems that employ the publish-subscribe pattern are the following:

- Graphical user interfaces, in which a user's low-level input actions are treated as events that are routed to appropriate input handlers
- MVC-based applications, in which view components are notified when the state of a model object changes
- Enterprise resource planning (ERP) systems, which integrate many components, each of which is only interested in a subset of system events
- Extensible programming environments, in which tools are coordinated through events
- Mailing lists, where a set of subscribers can register interest in specific topics

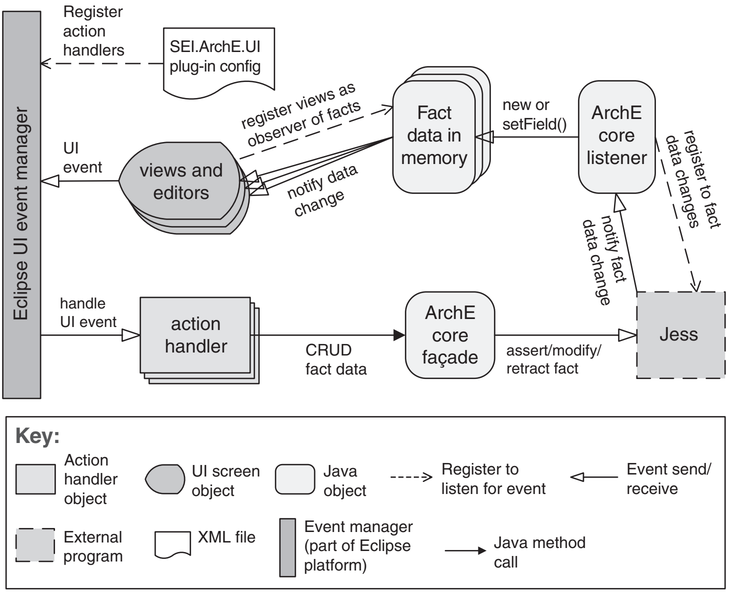


FIGURE 13.12 A typical publish-subscribe pattern realization

- Social networks, where "friends" are notified when changes occur to a person's website

The publish-subscribe pattern is used to send events and messages to an unknown set of recipients. Because the set of event recipients is unknown to the event producer, the correctness of the producer cannot, in general, depend on those recipients. Thus, new recipients can be added without modification to the producers.

Having components be ignorant of each other's identity results in easy modification of the system (adding or removing producers and consumers of data) but at the cost of runtime performance, because the publish-subscribe infrastructure is a kind of indirection, which adds latency. In addition, if the publish-subscribe connector fails completely, this is a single point of failure for the entire system.

The publish-subscribe pattern can take several forms:

- *List-based publish-subscribe* is a realization of the pattern where every publisher maintains a subscription list—a list of subscribers that have registered an interest in receiving the event. This version of the pattern is less decoupled than others, as we shall see below, and hence it does not provide as much modifiability, but it can be quite efficient in terms of runtime overhead. Also, if the components are distributed, there is no single point of failure.
- *Broadcast-based publish-subscribe* differs from list-based publish-subscribe in that publishers have less (or no) knowledge of the subscribers. Publishers simply publish events, which are then broadcast. Subscribers (or in a distributed system, services that act on behalf of the subscribers) examine each event as it arrives and determine whether the published event is of interest. This version has the potential to be very inefficient if there are lots of messages and most messages are not of interest to a particular subscriber.
- *Content-based publish-subscribe* is distinguished from the previous two variants, which are broadly categorized as "topic-based." Topics are predefined events, or messages, and a component subscribes to all events within the topic. Content, on the other hand, is much more general. Each event is associated with a set of attributes and is delivered to a subscriber only if those attributes match subscriber-defined patterns.

In practice the publish-subscribe pattern is typically realized by some form of message-oriented middleware, where the middleware is realized as a broker, managing the connections and channels of information between producers and consumers. This middleware is often responsible for the transformation of messages (or message protocols), in addition to routing and sometimes storing the messages. Thus the publish-subscribe pattern inherits the strengths and weaknesses of the broker pattern.

Shared-Data Pattern

Context: Various computational components need to share and manipulate large amounts of data. This data does not belong solely to any one of those components.

Problem: How can systems store and manipulate persistent data that is accessed by multiple independent components?

Solution: In the shared-data pattern, interaction is dominated by the exchange of persistent data between multiple *data accessors* and at least one *shared-data store*. Exchange may be initiated by the accessors or the data store. The connector type is *data reading and writing*. The general computational model associated with shared-data systems is that data accessors perform operations that require data from the data store and write results to one or more data stores. That data can be viewed and acted on by other data accessors. In a pure shared-data system, data accessors interact only through one or more shared-data stores. However, in practice shared-data systems also allow direct interactions between data accessors. The data-store components of a shared-data system provide shared access to data, support data persistence, manage concurrent access to data through transaction management, provide fault tolerance, support access control, and handle the distribution and caching of data values.

Specializations of the shared-data pattern differ with respect to the nature of the stored data—existing approaches include relational, object structures, layered, and hierarchical structures.

Although the sharing of data is a critical task for most large, complex systems, there are a number of potential problems associated with this pattern. For one, the shared-data store may be a performance bottleneck. For this reason, performance optimization has been a common theme in database research. The shared-data store is also potentially a single point of failure. Also, the producers and consumers of the shared data may be tightly coupled, through their knowledge of the structure of the shared data.

The shared-data pattern solution is summarized in Table 13.9.

The shared-data pattern is useful whenever various data items are persistent and have multiple accessors. Use of this pattern has the effect of decoupling the producer of the data from the consumers of the data; hence, this pattern supports modifiability, as the producers do not have direct knowledge of the consumers. Consolidating the data in one or more locations and accessing it in a common fashion facilitates performance tuning. Analyses associated with this pattern usually center on qualities such as data consistency, performance, security, privacy, availability, scalability, and compatibility with, for example, existing repositories and their data.

When a system has more than one data store, a key architecture concern is the mapping of data and computation to the data. Use of multiple stores may occur because the data is naturally, or historically, partitioned into separable stores. In other cases data may be replicated over several stores to improve performance or availability through redundancy. Such choices can strongly affect the qualities noted above.

Figure 13.13 shows the diagram of a shared-data view of an enterprise access management system. There are three types of accessor components: Windows applications, web applications, and headless programs (programs or scripts that run in background and don't provide any user interface).

TABLE 13.9 Shared-Data Pattern Solution

Overview	Communication between data accessors is mediated by a shared-data store. Control may be initiated by the data accessors or the data store. Data is made persistent by the data store.
Elements	*Shared-data store.* Concerns include types of data stored, data performance-oriented properties, data distribution, and number of accessors permitted. *Data accessor component*. *Data reading and writing connector*. An important choice here is whether the connector is transactional or not, as well as the read/write language, protocols, and semantics.
Relations	*Attachment* relation determines which data accessors are connected to which data stores.
Constraints	Data accessors interact with the data store(s).
Weaknesses	The shared-data store may be a performance bottleneck. The shared-data store may be a single point of failure. Producers and consumers of data may be tightly coupled.

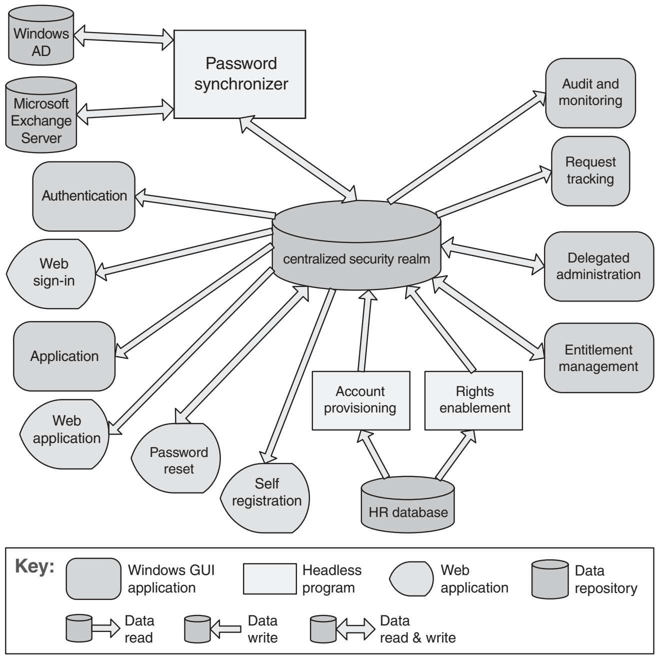


FIGURE 13.13 The shared-data diagram of an enterprise access management system

Allocation Patterns

Map-Reduce Pattern

Context: Businesses have a pressing need to quickly analyze enormous volumes of data they generate or access, at petabyte scale. Examples include logs of interactions in a social network site, massive document or data repositories, and pairs of <source, target> web links for a search engine. Programs for the analysis of this data should be easy to write, run efficiently, and be resilient with respect to hardware failure.

Problem: For many applications with ultra-large data sets, sorting the data and then analyzing the grouped data is sufficient. The problem the map-reduce pattern solves is to efficiently perform a distributed and parallel sort of a large data set and provide a simple means for the programmer to specify the analysis to be done.

Solution: The map-reduce pattern requires three parts: First, a specialized infrastructure takes care of allocating software to the hardware nodes in a massively parallel computing environment and handles sorting the data as needed. A node may be a standalone processor or a core in a multi-core chip. Second and third are two programmer-coded functions called, predictably enough, *map* and *reduce*.

The map function takes as input a key (key1) and a data set. The purpose of the map function is to filter and sort the data set. All of the heavy analysis takes place in the reduce function. The input key in the map function is used to filter the data. Whether a data record is to be involved in further processing is determined by the map function. A second key (key2) is also important in the map function. This is the key that is used for sorting. The output of the map function consists of a <key2, value> pair, where the key2 is the sorting value and the value is derived from the input record.

Sorting is performed by a combination of the map and the infrastructure. Each record output by map is hashed by key2 into a disk partition. The infrastructure maintains an index file for key2 on the disk partition. This allows for the values on the disk partition to be retrieved in key2 order.

The performance of the map phase of map-reduce is enhanced by having multiple map instances, each processing a different portion of the disk file being processed. Figure 13.14 shows how the map portion of map-reduce processes data. An input file is divided into portions, and a number of map instances are created to process each portion. The map function processes its portion into a number of partitions, based on programmer-specified logic.

The reduce function is provided with all the sets of <key2, value> pairs emitted by all the map instances in sorted order. Reduce does some programmer-specified analysis and then emits the results of that analysis. The output set is almost always much smaller than the input sets, hence the name "reduce." The term "load" is sometimes used to describe the final set of data emitted. Figure 13.14 also shows one instance (of many possible instances) of the reduce processing, called Reduce Instance 2. Reduce Instance 2 is receiving data from all of the Partition 2s produced by the various map instances. It is possible that there are several iterations of reduce for large files, but this is not shown in Figure 13.14.

A classic teaching problem for map-reduce is counting word occurrences in a document. This example can be carried out with a single map function. The document is the data set. The map function will find every word in the document and output a <word, 1> pair for each. For example, if the document begins with the words "Having a whole book . . . ," then the first results of map will be

```
<Having, 1>
<a, 1>
<whole, 1>
<book, 1>
```

In practice, the "a" would be one of the words filtered by map.

Pseudocode for map might look like this:

```
map(String key, String value):
// key: document name
// value: document contents
for each word w in value:
Emit (w, "1");
```

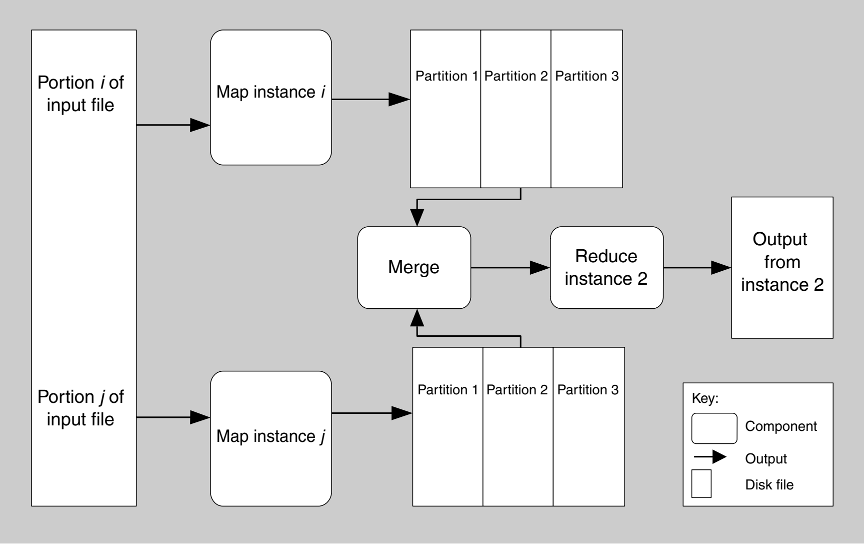


FIGURE 13.14 A component-and-connector view of map-reduce showing how the data processed by map is partitioned and subsequently processed by reduce

The reduce function will take that list in sorted order, add up the 1s for each word to get a count, and output the result.

The corresponding reduce function would look like this:

```
reduce(List <key, value>):
// key: a word
// value: an integer
int result = 0;
sort input
for each input value:
for each input pair with same word
result ++ ;
Emit (word, result)
result = 0
```

Larger data sets lead to a much more interesting solution. Suppose we want to continuously analyze Twitter posts over the last hour to see what topics are currently “trending.” This is analogous to counting word occurrences in millions of documents. In that case, each document (tweet) can be assigned to its own instance of the map function. (If you don’t have millions of processors handy, you can break the tweet collection into groups that match the number of processors in your processor farm, and process the collection in waves, one group after the other.) Or we can use a dictionary to give us a list of words, and each map function can be assigned its own word to look for across all tweets.

There can also be multiple instances of reduce. These are usually arranged so that the reduction happens in stages, with each stage processing a smaller list (with a smaller number of reduce instances) than the previous stage. The final stage is handled by a single reduce function that produces the final output.

Of course, the map-reduce pattern is not appropriate in all instances. Some considerations that would argue against adopting this pattern are these:

- If you do not have large data sets, then the overhead of map-reduce is not justified.
- If you cannot divide your data set into similar sized subsets, the advantages of parallelism are lost.
- If you have operations that require multiple reduces, this will be complex to orchestrate.

Commercial implementations of map-reduce provide infrastructure that takes care of assignment of function instances to hardware, recovery and reassignment in case of hardware failure (a common occurrence in massively parallel computing environments), and utilities like sorting of the massive lists that are produced along the way.

Table 13.10 summarizes the solution of the map-reduce pattern.

Map-reduce is a cornerstone of the software of some of the most familiar names on the web, including Google, Facebook, eBay, and Yahoo!

TABLE 13.10 Map-Reduce Pattern Solution

Overview	The map-reduce pattern provides a framework for analyzing a large distributed set of data that will execute in parallel, on a set of processors. This parallelization allows for low latency and high availability. The map performs the *extract* and *transform* portions of the analysis and the reduce performs the *loading* of the results. (*Extract-transform-load* is sometimes used to describe the functions of the map and reduce.)
Elements	*Map* is a function with multiple instances deployed across multiple processors that performs the extract and transformation portions of the analysis. *Reduce* is a function that may be deployed as a single instance or as multiple instances across processors to perform the load portion of extract-transform-load. The *infrastructure* is the framework responsible for deploying map and reduce instances, shepherding the data between them, and detecting and recovering from failure.
Relations	*Deploy on* is the relation between an instance of a map or reduce function and the processor onto which it is installed. *Instantiate, monitor, and control* is the relation between the infrastructure and the instances of map and reduce.
Constraints	The data to be analyzed must exist as a set of files. The map functions are stateless and do not communicate with each other. The only communication between the map instances and the reduce instances is the data emitted from the map instances as <key, value> pairs.
Weaknesses	If you do not have large data sets, the overhead of map-reduce is not justified. If you cannot divide your data set into similar sized subsets, the advantages of parallelism are lost. Operations that require multiple reduces are complex to orchestrate.

Multi-tier Pattern

The multi-tier pattern is a C&C pattern or an allocation pattern, depending on the criteria used to define the tiers. Tiers can be created to group components of similar functionality, in which case it is a C&C pattern. However, in many, if not most, cases tiers are defined with an eye toward the computing environment on which the software will run: A client tier in an enterprise system will not be running on the computer that hosts the database. That makes it an allocation pattern, mapping software elements—perhaps produced by applying C&C patterns—to computing elements. Because of that reason, we have chosen to list it as an allocation pattern.

Context: In a distributed deployment, there is often a need to distribute a system's infrastructure into distinct subsets. This may be for operational or business reasons (for example, different parts of the infrastructure may belong to different organizations).

Problem: How can we split the system into a number of computationally independent execution structures—groups of software and hardware—connected by some communications media? This is done to provide specific server environments optimized for operational requirements and resource usage.

Solution: The execution structures of many systems are organized as a set of logical groupings of components. Each grouping is termed a *tier*. The grouping of components into tiers may be based on a variety of criteria, such as the type of component, sharing the same execution environment, or having the same runtime purpose.

The use of tiers may be applied to any collection (or pattern) of runtime components, although in practice it is most often used in the context of client-server patterns. Tiers induce topological constraints that restrict which components may communicate with other components. Specifically, connectors may exist only between components in the same tier or residing in adjacent tiers. The multi-tier pattern found in many Java EE and Microsoft .NET applications is an example of organization in tiers derived from the client-server pattern.

Additionally, tiers may constrain the *kinds* of communication that can take place across adjacent tiers. For example, some tiered patterns require call-return communication in one direction but event-based notification in the other.

The main weakness with the multi-tier architecture is its cost and complexity. For simple systems, the benefits of the multi-tier architecture may not justify its up-front and ongoing costs, in terms of hardware, software, and design and implementation complexity.

Tiers are not components, but rather logical groupings of components. Also, don't confuse tiers with layers! Layering is a pattern of modules (a unit of implementation), while tiers applies only to runtime entities.

Table 13.11 summarizes the solution part of the multi-tier pattern.

Tiers make it easier to ensure security, and to optimize performance and availability in specialized ways. They also enhance the modifiability of the system, as the computationally independent subgroups need to agree on protocols for interaction, thus reducing their coupling.

Figure 13.15 uses an informal notation to describe the multi-tier architecture of the Consumer Website Java EE application. This application is part of the Adventure Builder system. Many component-and-connector types are specific to the supporting platform, which is Java EE in this case.

TABLE 13.11 Multi-tier Pattern Solution

Overview	The execution structures of many systems are organized as a set of logical groupings of components. Each grouping is termed a *tier*. The grouping of components into tiers may be based on a variety of criteria, such as the type of component, sharing the same execution environment, or having the same runtime purpose.
Elements	*Tier,* which is a logical grouping of software components. Tiers may be formed on the basis of common computing platforms, in which case those platforms are also elements of the pattern.
Relations	*Is part of,* to group components into tiers. *Communicates with,* to show how tiers and the components they contain interact with each other. *Allocated to,* in the case that tiers map to computing platforms.
Constraints	A software component belongs to exactly one tier.
Weaknesses	Substantial up-front cost and complexity.

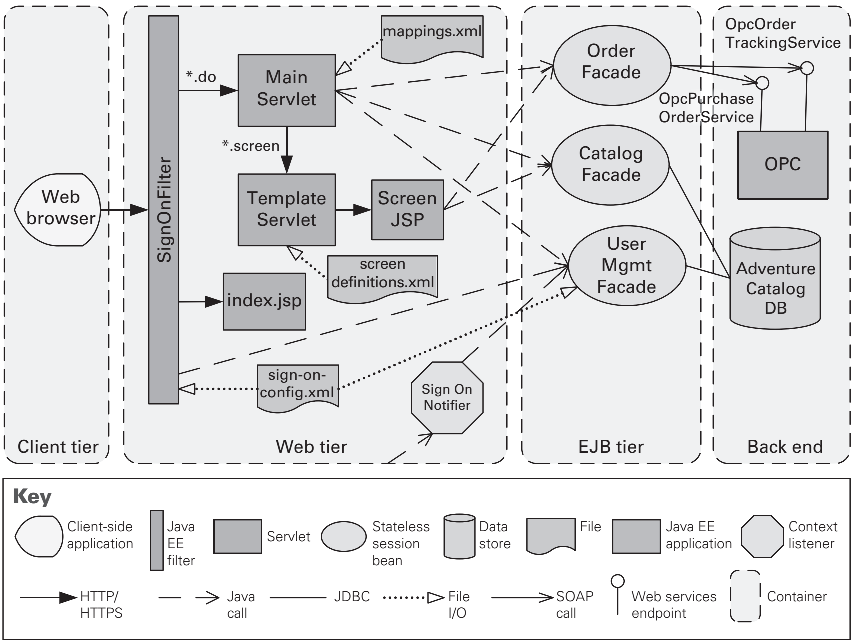


FIGURE 13.15 A multi-tier view of the Consumer Website Java EE application, which is part of the Adventure Builder system

Other Allocation Patterns. There are several published deployment styles. Microsoft publishes a "Tiered Distribution" pattern, which prescribes a particular allocation of components in a multi-tier architecture to the hardware they will run on. Similarly, IBM's WebSphere handbooks describe a number of what they call "topologies" along with the quality attribute criteria for choosing among them. There are 11 topologies (specialized deployment patterns) described for WebSphere version 6, including the "single machine topology (stand-alone server)," "reverse proxy topology," "vertical scaling topology," "horizontal scaling topology," and "horizontal scaling with IP sprayer topology."

There are also published work assignment patterns. These take the form of often-used team structures. For example, patterns for globally distributed Agile projects include these:

- *Platform.* In software product line development, one site is tasked with developing reusable core assets of the product line, and other sites develop applications that use the core assets.
- *Competence center.* Work is allocated to sites depending on the technical or domain expertise located at a site. For example, user interface design is done at a site where usability engineering experts are located.
- *Open source.* Many independent contributors develop the software product in accordance with a technical integration strategy. Centralized control is minimal, except when an independent contributor integrates his code into the product line.

13.3 Relationships between Tactics and Patterns

Patterns and tactics together constitute the software architect's primary tools of the trade. How do they relate to each other?

Patterns Comprise Tactics

As we said in the introduction to this chapter, tactics are the "building blocks" of design from which architectural patterns are created. Tactics are atoms and patterns are molecules. Most patterns consist of (are constructed from) several different tactics, and although these tactics might all serve a common purpose—such as promoting modifiability, for example—they are often chosen to promote *different* quality attributes. For example, a tactic might be chosen that makes an availability pattern more secure, or that mitigates the performance impact of a modifiability pattern.

Consider the example of the layered pattern, the most common pattern in all of software architecture (virtually all nontrivial systems employ layering). The layered pattern can be seen as the amalgam of several tactics—increase semantic coherence, abstract common services, encapsulate, restrict communication paths, and use an intermediary. For example:

- *Increase semantic coherence*. The goal of ensuring that a layer's responsibilities all work together without excessive reliance on other layers is achieved by choosing responsibilities that have semantic coherence. Doing so binds responsibilities that are likely to be affected by a change. For example, responsibilities that deal with hardware should be allocated to a hardware layer and not to an application layer; a hardware responsibility typically does not have semantic coherence with the application responsibilities.
- *Restrict dependencies*. Layers define an ordering and only allow a layer to use the services of its adjacent lower layer. The possible communication paths are reduced to the number of layers minus one. This limitation has a great influence on the dependencies between the layers and makes it much easier to limit the side effects of replacing a layer.

Without any one of its tactics, the pattern might be ineffective. For example, if the restrict dependencies tactic is not employed, then any function in any layer can call any other function in any other layer, destroying the low coupling that makes the layering pattern effective. If the increase semantic coherence tactic is not employed, then functionality could be randomly sprinkled throughout the layers, destroying the separation of concerns, and hence ease of modification, which is the prime motivation for employing layers in the first place.

Table 13.12 shows a number of the architectural patterns described in the book *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*, by Buschmann et al., and shows which modifiability tactics they employ.

Using Tactics to Augment Patterns

A pattern is described as a solution to a class of problems in a general context. When a pattern is chosen and applied, the context of its application becomes very specific. A documented pattern is therefore underspecified with respect to applying it in a specific situation.

To make a pattern work in a given architectural context, we need to examine it from two perspectives:

- The inherent quality attribute tradeoffs that the pattern makes. Patterns exist to achieve certain quality attributes, and we need to compare the ones they promote (and the ones they diminish) with our needs.
- Other quality attributes that the pattern isn't directly concerned with, but which it nevertheless affects, and which are important in our application.

TABLE 13.12 Architecture Patterns and Corresponding Tactics ([Bachmann 07])

	Modifiability									
	Increase Cohesion		Reduce Coupling					Defer Binding Time		
Pattern	Increase Semantic Coherence	Abstract Common Services	Encapsulate	Use a Wrapper	Restrict Comm. Paths	Use an Intermediary	Raise the Abstraction Level	Use Runtime Registration	Use Startup-Time Binding	Use Runtime Binding
Layered	X	X	X		X	X	X			
Pipes and Filters	X		X		X	X			X	
Blackboard	X	X			X	X	X	X		X
Broker	X	X	X		X	X	X	X		
Model View Controller	X		X			X				X
Presentation Abstraction Control	X		X			X	X			
Microkernel	X	X	X		X	X				
Reflection	X		X							

To illustrate these concerns in particular, and how to use tactics to augment patterns in general, we'll use the broker pattern as a starting point.

The broker pattern is widely used in distributed systems and dates back at least to its critical role in CORBA-based systems. Broker is a crucial component of any large-scale, dynamic, service-oriented architecture.

Using this pattern, a client requesting some information from a server does not need to know the location or APIs of the server. The client simply contacts the broker (typically through a client-side proxy); this is illustrated in the UML sequence diagram in Figure 13.16.

Weaknesses of the Broker Pattern. In Section 13.2 we enumerated several weaknesses of the broker pattern. Here we will examine these weaknesses in more detail. The broker pattern has several weaknesses with respect to certain quality attributes. For example:

- *Availability*. The broker, if implemented as suggested in Figure 13.6, is a single point of failure. The liveness of servers, the broker, and perhaps even the clients need to be monitored, and repair mechanisms must be provided.

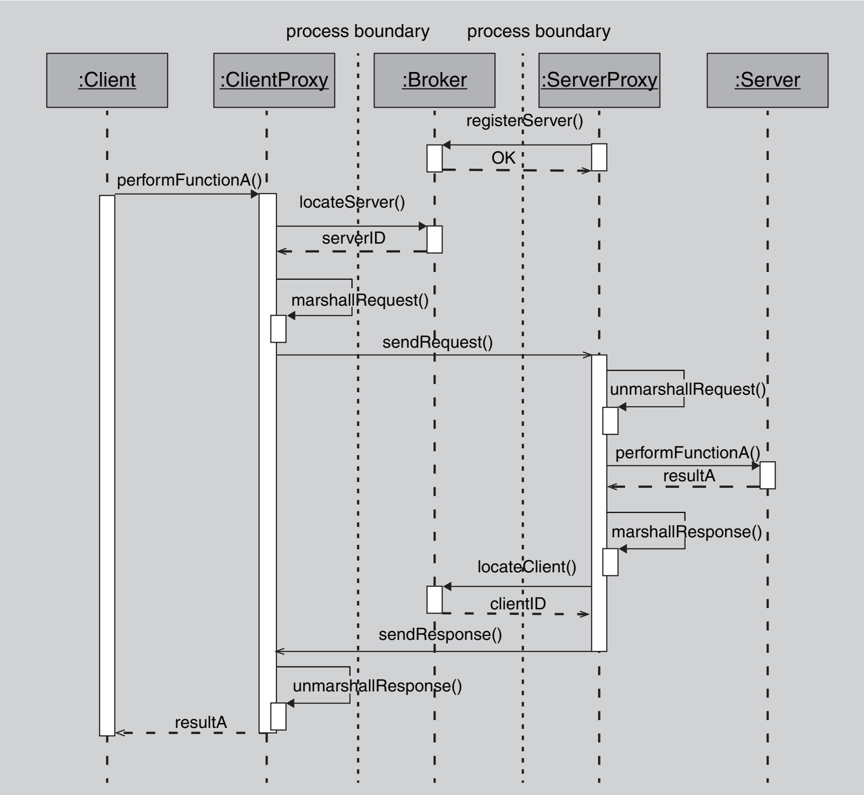


FIGURE 13.16 A sequence diagram showing a typical client-server interaction mediated by a broker

- *Performance*. The levels of indirection between the client (requesting the information or service) and the server (providing the information or service) add overhead, and hence add latency. Also, the broker is a potential performance bottleneck if direct communication between the client and server is not desired (for example, for security reasons).
- *Testability*. Brokers are employed in complex multi-process and multi-processor systems. Such systems are typically highly dynamic. Requests and responses are typically asynchronous. All of this makes testing and debugging such systems extremely difficult. But the description of the broker pattern provides no testing functionality, such as testing interfaces, state or activity capture and playback capabilities, and so forth.

- *Security*. Because the broker pattern is primarily used when the system spans process and processor boundaries—such as on web-based systems—security is a legitimate concern. However, the broker pattern as presented does not offer any means to authenticate or authorize clients or servers, and provides no means of protecting the communication between clients and servers.

Of these quality attributes, the broker pattern is mainly associated with poor performance (the well-documented price for the loose coupling it brings to systems). It is largely unconcerned with the other quality attributes in this list; they aren't mentioned in most published descriptions. But as the other bullets show, they can be unacceptable "collateral damage" that come with the broker's benefits.

Improving the Broker Pattern with Tactics. How can we use tactics to plug the gaps between the "out of the box" broker pattern and a version of it that will let us meet the requirements of a demanding distributed system? Here are some options:

- The increase available resources performance tactic would lead to multiple brokers, to help with performance and availability.
- The maintain multiple copies tactic would allow each of these brokers to share state, to ensure that they respond identically to client requests.
- Load balancing (an application of the scheduling resources tactic) would ensure that one broker is not overloaded while another one sits idle.
- Heartbeat, exception detection, or ping/echo would give the replicated brokers a way of notifying clients and notifying each other when one of them is out of service, as a means of detecting faults.

Of course, each of these tactics brings a tradeoff. Each complicates the design, which will now take longer to implement, be more costly to acquire, and be more costly to maintain. Load balancing introduces indirection that will add latency to each transaction, thus giving back some of the performance it was intended to increase. And the load balancer is a single point of failure, so it too must be replicated, further increasing the design cost and complexity.

13.4 Using Tactics Together

Tactics, as described in Chapters 5–11, are design primitives aimed at managing a single quality attribute response. Of course, this is almost never true in practice; every tactic has its main effect—to manage modifiability or performance or safety, and so on—and it has its side effects, its tradeoffs. On the face of it, the situation for an architect sounds hopeless. Whatever you do to improve one quality attribute endangers another. We are able to use tactics profitably because we can gauge the direct and side effects of a tactic, and when the tradeoff is acceptable, we employ the tactic. In doing so we gain some benefit in our quality attribute of interest while giving up something else (with respect to a different quality attribute and, we hope, of a much smaller magnitude).

This section will walk through an example that shows how applying tactics to a pattern can produce negative effects in one area, but how adding other tactics can bring relief and put you back in an acceptable design space. The point is to show the interplay between tactics that you can use to your advantage. Just as some combinations of liquids are noxious whereas others yield lovely things like strawberry lemonade, tactics can either make things worse or put you in a happy design space. Here, then, is a walkthrough of tactic mixology.

Consider a system that needs to detect faults in its components. A common tactic for detecting faults is ping/echo. Let us assume that the architect has decided to employ ping/echo as a way to detect failed components in the system. Every tactic has one or more side effects, and ping/echo is no different. Common considerations associated with ping/echo are these:

- *Security*. How to prevent a ping flood attack?
- *Performance*. How to ensure that the performance overhead of ping/echo is small?
- *Modifiability*. How to add ping/echo to the existing architecture?

We can represent the architect's reasoning and decisions thus far as shown in Figure 13.17.

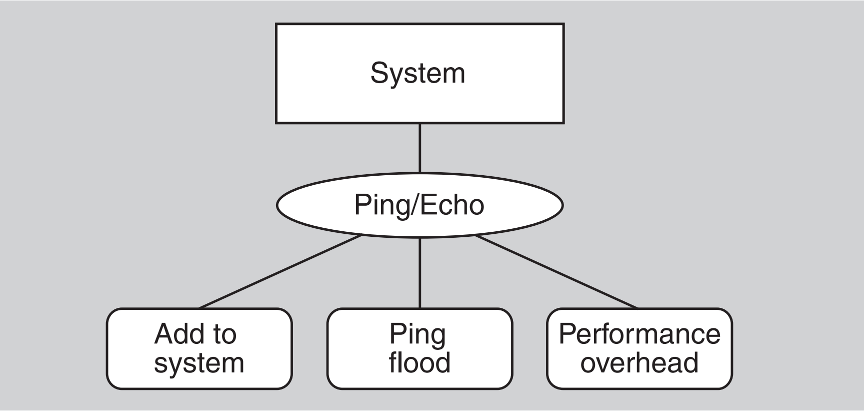


FIGURE 13.17 Partial availability decisions

Suppose the architect determines that the performance tradeoff (the overhead of adding ping/echo to the system) is the most severe. A tactic to address the performance side effect is *increase available resources*. Considerations associated with increase available resources are these:

- *Cost*. Increased resources cost more.
- *Performance*. How to utilize the increased resources efficiently?

This set of design decisions can now be represented as shown in Figure 13.18.

Now the architect chooses to deal with the resource utilization consequence of employing increase available resources. These resources must be used efficiently or else they are simply adding cost and complexity to the system. A tactic that can address the efficient use of resources is the employment of a *scheduling policy*. Considerations associated with the scheduling policy tactic are these:

- *Modifiability*. How to add the scheduling policy to the existing architecture?
- *Modifiability*. How to change the scheduling policy in the future?

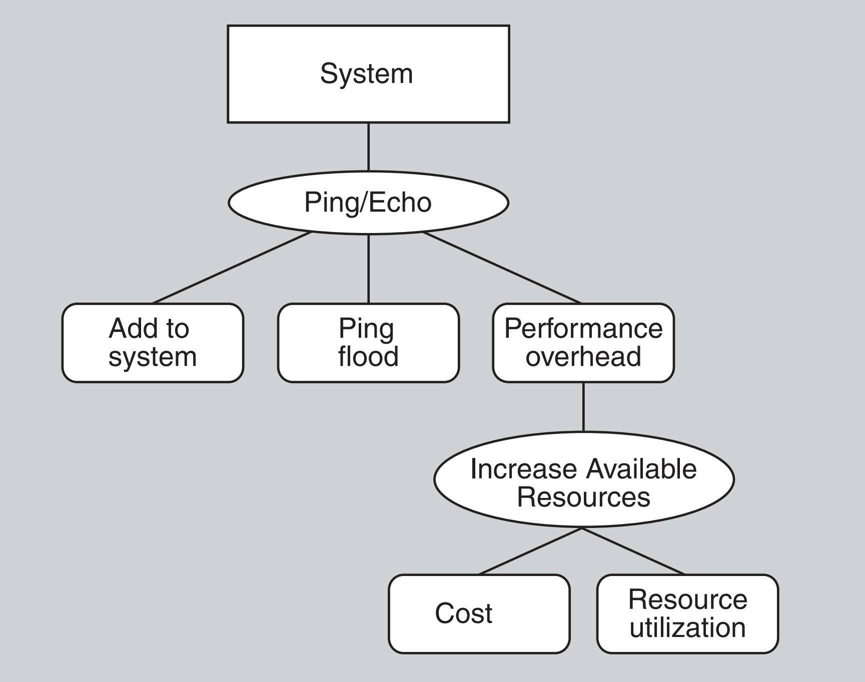


FIGURE 13.18 More availability decisions

The set of design decisions that includes the scheduling policy tactic can now be represented as in Figure 13.19.

Next the architect chooses to deal with the modifiability consequence of employing a scheduling policy tactic. A tactic to address the addition of the scheduler to the system is to *use an intermediary*, which will insulate the choice of scheduling policy from the rest of the system. One consideration associated with use an intermediary is this:

- *Modifiability*. How to ensure that all communication passes through the intermediary?

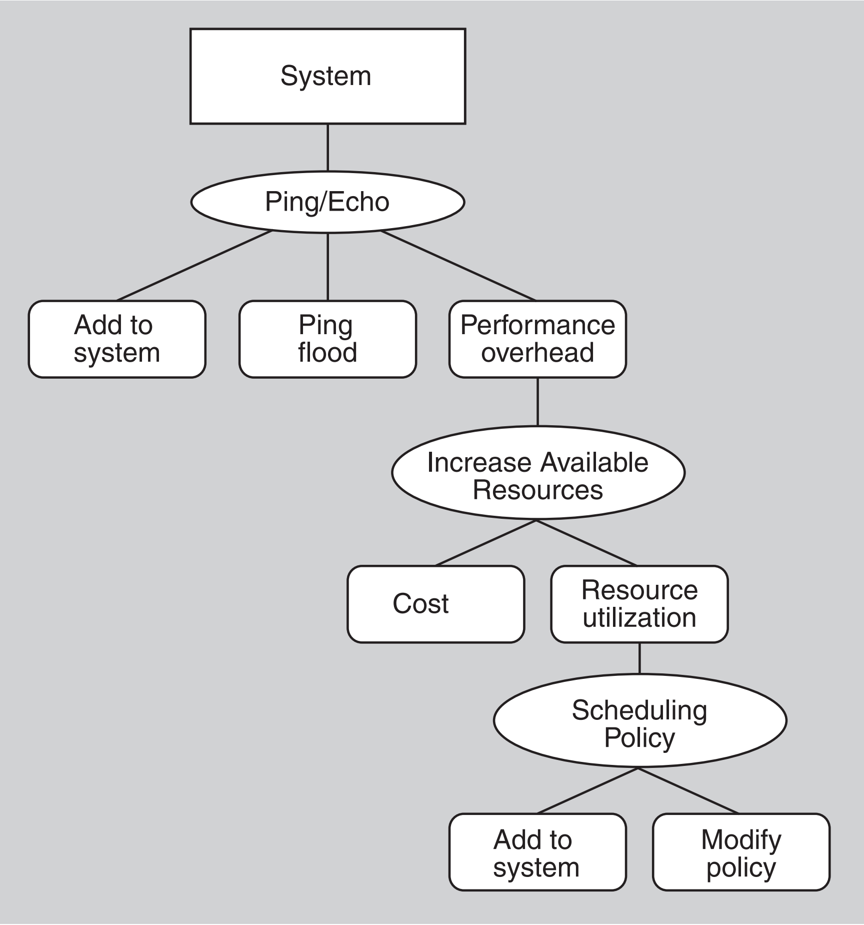


FIGURE 13.19 Still more availability decisions

We can now represent the tactics-based set of architectural design decisions made thus far as in Figure 13.20.

A tactic to address the concern that all communication passes through the intermediary is *restrict dependencies*. One consideration associated with the restrict dependencies tactic is this:

- *Performance*. How to ensure that the performance overhead of the intermediary is not excessive?

This design problem has now become recursive! At this point (or in fact, at any point in the tree of design decisions that we have described) the architect might determine that the performance overhead of the intermediary is small enough that no further design decisions need to be made.

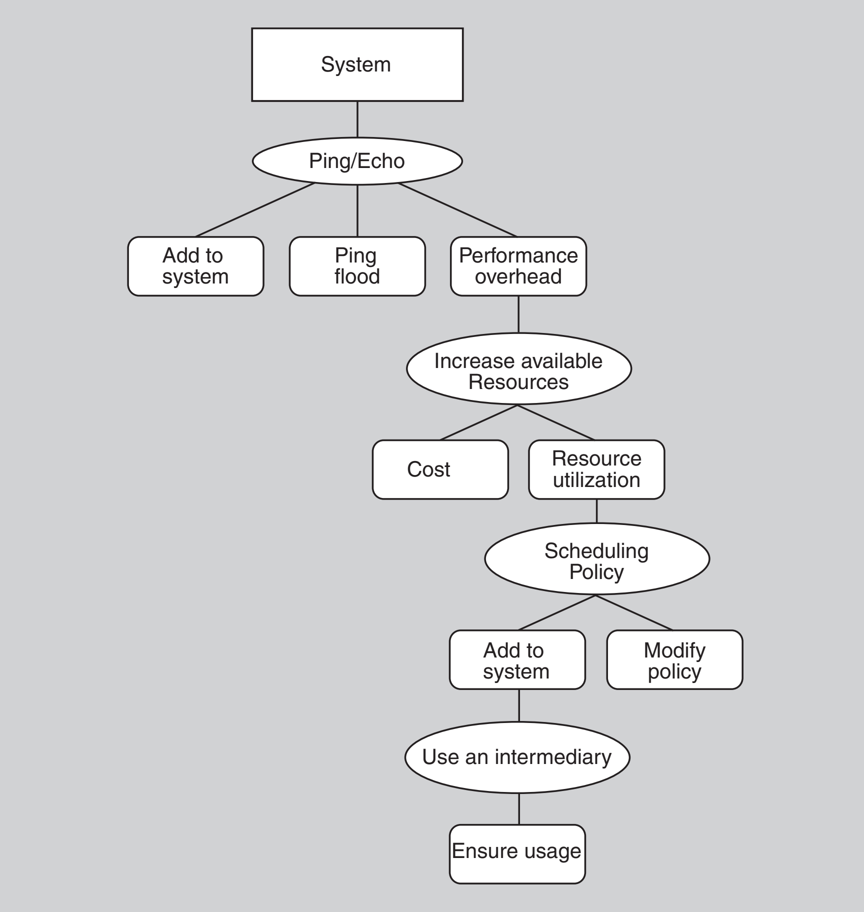


FIGURE 13.20 As far as we go with availability decisions

Applying successive tactics is like moving through a game space, and it's a little like chess: Good players are able to see the consequences of the move they're considering, and the very good players are able to look several moves ahead. In Chapter 17 we'll see the activity of design treated as an exercise of "generate and test": propose a design and test it to see if it's satisfactory. Applying tactics to an existing design solution, such as a pattern, is one technique for generating a design for subsequent testing.

13.5 Summary

An architectural pattern

- is a package of design decisions that is found repeatedly in practice,
- has known properties that permit reuse, and
- describes a *class* of architectures.

Because patterns are (by definition) found repeatedly in practice, one does not *invent* them; one *discovers* them.

Tactics are simpler than patterns. Tactics typically use just a single structure or computational mechanism, and they are meant to address a single architectural force. For this reason they give more precise control to an architect when making design decisions than patterns, which typically combine multiple design decisions into a package. Tactics are the "building blocks" of design from which architectural patterns are created. Tactics are atoms and patterns are molecules.

An architectural pattern establishes a relationship between:

- *A context*. A recurring, common situation in the world that gives rise to a problem.
- *A problem.* The problem, appropriately generalized, that arises in the given context.
- *A solution.* A successful architectural resolution to the problem, appropriately abstracted.

Complex systems exhibit multiple patterns at once.

Patterns can be categorized by the dominant type of elements that they show: module patterns show modules, component-and-connector patterns show components and connectors, and allocation patterns show a combination of software elements (modules, components, connectors) and nonsoftware elements. Most published patterns are C&C patterns, but there are module patterns and allocation patterns as well. This chapter showed examples of each type.

A pattern is described as a solution to a class of problems in a general context. When a pattern is chosen and applied, the context of its application becomes very specific. A documented pattern is therefore underspecified with respect to applying it in a specific situation. We can make a pattern more specific to our problem by augmenting it with tactics. Applying successive tactics is like moving through a game space, and is a little like chess: the consequences of the next move are important, and looking several moves ahead is helpful.

13.6 For Further Reading

There are many existing repositories of patterns and books written about patterns. The original and most well-known work on object-oriented design patterns is by the "Gang of Four" [Gamma 94].

The Gang of Four's discussion of patterns included patterns at many levels of abstraction. In this chapter we have focused entirely on architectural patterns. The patterns that we have presented here are intended as representative examples. This chapter's inventory of patterns is in no way meant to be exhaustive. For example, while we describe the SOA pattern, entire repositories of SOA patterns (refinements of the basic SOA pattern) have been created. A good place to start is www.soapatterns.org.

Some good references for pattern-oriented architecture are [Buschmann 96], [Hanmer 07], [Schmidt 00], and [Kircher 03].

A good place to learn more about the map-reduce pattern is Google's foundational paper on it [Dean 04].

Map-reduce is the tip of the spear of the so-called "NoSQL" movement, which seeks to displace the relational database from its venerable and taken-for-granted status in large data-processing systems. The movement has some of the revolutionary flavor of the Agile movement, except that NoSQL advocates are claiming a better (for them) technology, as opposed to a better process. You can easily find NoSQL podcasts, user forums, conferences, and blogs; it's also discussed in Chapter 26.

[Bachmann 07] discusses the use of tactics in the layered pattern and is the source for some of our discussion of that.

The passage in this chapter about augmenting ping/echo with other tactics to achieve the desired combination of quality attributes is based on the work of Kiran Kumar and TV Prabhakar [Kumar 10a] and [Kumar 10b].

[Urdangarin 08] is the source of the work assignment patterns described in Section 13.2.

The Adventure Builder system shown in Figures 13.11 and 13.15 comes from [AdvBuilder 10].

13.7 Discussion Questions

1. What's the difference between an *architectural* pattern, such as those described in this chapter and in the Pattern-Oriented Software Architecture series of books, and *design* patterns, such as those collected by the Gang of Four in 1994 and many other people subsequently? Given a pattern, how would you decide whether it was an architectural pattern, a design pattern, a code pattern, or something else?
2. SOA systems feature dynamic service registration and discovery. Which quality attributes does this capability enhance and which does it threaten? If you had to make a recommendation to your boss about whether your company's SOA system should use external services it discovers at runtime, what would you say?
3. Write a complete pattern description for the "competence center" work assignment pattern mentioned in Section 13.2.
4. For a data set that is a set of web pages, sketch a map function and a reduce function that together provide a basic search engine capability.
5. Describe how the layered pattern makes use of these tactics: abstract common services, encapsulate, and use an intermediary.

This page intentionally left blank

14

Quality Attribute Modeling and Analysis

> *Do not believe in anything simply because you have heard it . . . Do not believe in anything merely on the authority of your teachers and elders. Do not believe in traditions because they have been handed down for many generations. But after observation and analysis, when you find that anything agrees with reason and is conducive to the good and benefit of one and all, then accept it and live up to it.*
>
> —Prince Gautama Siddhartha

In Chapter 2 we listed thirteen reasons why architecture is important, worth studying, and worth practicing. Reason 6 is that the analysis of an architecture enables early prediction of a system's qualities. This is an extraordinarily powerful reason! Without it, we would be reduced to building systems by choosing various structures, implementing the system, measuring the system for its quality attribute responses, and all along the way hoping for the best. Architecture lets us do better than that, much better. We can analyze an architecture to see how the system or systems we build from it will perform with respect to their quality attribute goals, even before a single line of code has been written. This chapter will explore how.

The methods available depend, to a large extent, on the quality attribute to be analyzed. Some quality attributes, especially performance and availability, have well-understood and strongly validated analytic modeling techniques. Other quality attributes, for example security, can be analyzed through checklists. Still others can be analyzed through back-of-the-envelope calculations and thought experiments.

Our topics in this chapter range from the specific, such as creating models and analyzing checklists, to the general, such as how to generate and carry out the thought experiments to perform early (and necessarily crude) analysis. Models and checklists are focused on particular quality attributes but can aid in the analysis of any system with respect to those attributes. Thought experiments, on the other hand, can consider multiple quality attributes simultaneously but are only applicable to the specific system under consideration.

14.1 Modeling Architectures to Enable Quality Attribute Analysis

Some quality attributes, most notably performance and availability, have well-understood, time-tested analytic models that can be used to assist in an analysis. By *analytic model*, we mean one that supports quantitative analysis. Let us first consider performance.

Analyzing Performance

In Chapter 12 we discussed the fact that models have parameters, which are values you can set to predict values about the entity being modeled (and in Chapter 12 we showed how to use the parameters to help us derive tactics for the quality attribute associated with the model). As an example we showed a queuing model for performance as Figure 12.2, repeated here as Figure 14.1. The parameters of this model are the following:

- The arrival rate of events
- The chosen queuing discipline
- The chosen scheduling algorithm
- The service time for events
- The network topology
- The network bandwidth
- The routing algorithm chosen

In this section, we discuss how such a model can be used to understand the latency characteristics of an architectural design.

To apply this model in an analytical fashion, we also need to have previously made some architecture design decisions. We will use model-view-controller as our example here. MVC, as presented in Section 13.2, says nothing about its deployment. That is, there is no specification of how the model, the view, and the controller are assigned to processes and processors; that's not part of the pattern's concern. These and other design decisions have to be made to transform a pattern into an architecture. Until that happens, one cannot say anything with authority about how an MVC-based implementation will perform. For this example we will assume that there is one instance each of the model, the view, and the controller, and that each instance is allocated to a separate processor. Figure 14.2 shows MVC following this allocation scheme.

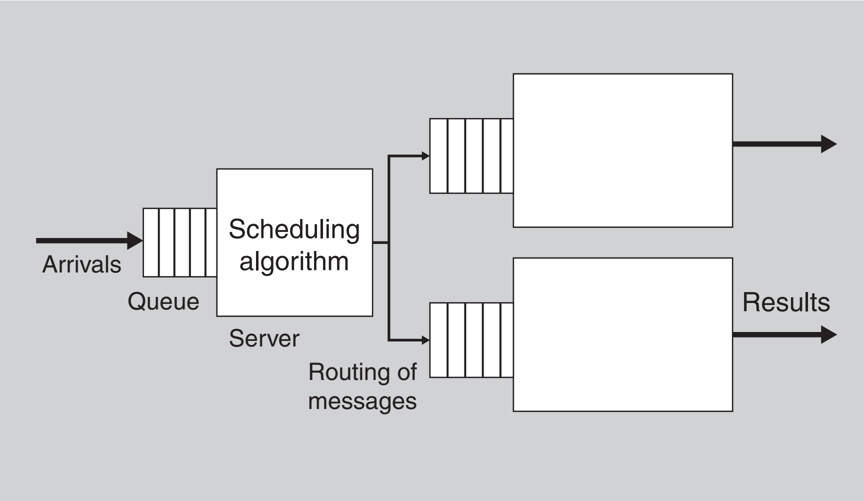


FIGURE 14.1 A queuing model of performance

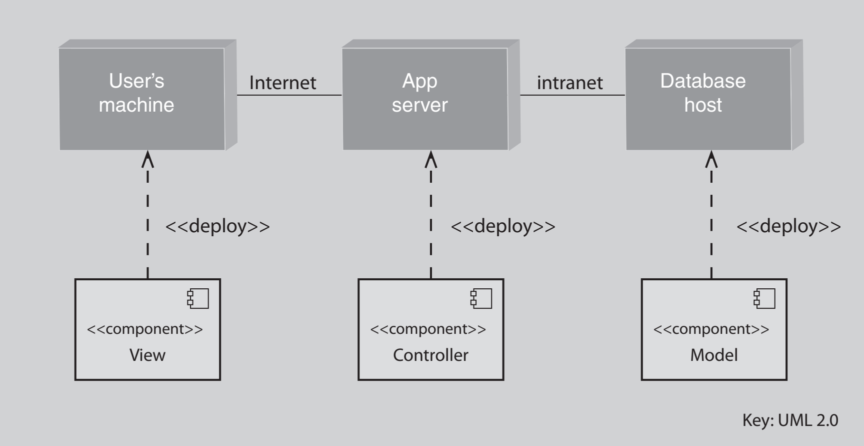


FIGURE 14.2 An allocation view, in UML, of a model-view-controller architecture

Given that quality attribute models such as the performance model shown in Figure 14.1 already exist, the problem becomes how to map these allocation and coordination decisions onto Figure 14.1. Doing this yields Figure 14.3. There are requests coming from users outside the system—labeled as 1 in Figure 14.3—arriving at the view. The view processes the requests and sends some transformation of the requests on to the controller—labeled as 2. Some actions of the controller are returned to the view—labeled as 3. The controller sends other actions on to the model—labeled 4. The model performs its activities and sends information back to the view—labeled 5.

To analyze the model in Figure 14.3, a number of items need to be known or estimated:

- The frequency of arrivals from outside the system
- The queuing discipline used at the view queue
- The time to process a message within the view
- The number and size of messages that the view sends to the controller
- The bandwidth of the network that connects the view and the controller
- The queuing discipline used by the controller
- The time to process a message within the controller
- The number and size of messages that the controller sends back to the view
- The bandwidth of the network used for messages from the controller to the view
- The number and size of messages that the controller sends to the model
- The queuing discipline used by the model
- The time to process a message within the model
- The number and size of messages the model sends to the view
- The bandwidth of the network connecting the model and the view

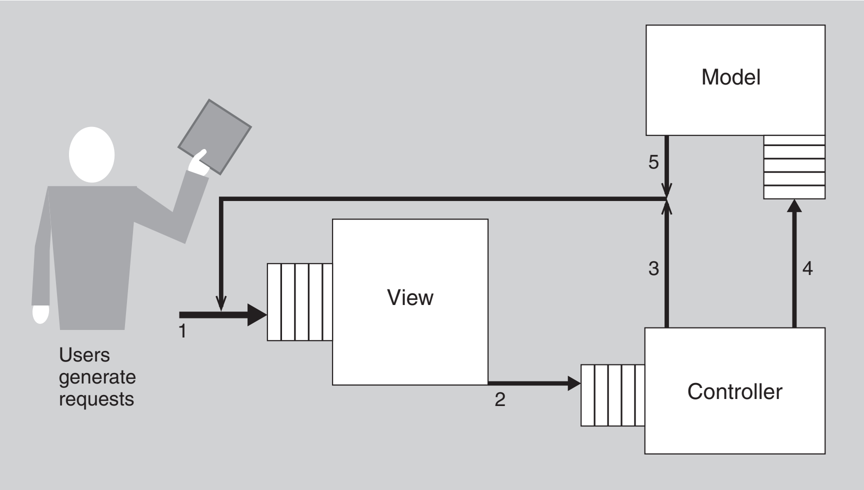


FIGURE 14.3 A queuing model of performance for MVC

Given all of these assumptions, the latency for the system can be estimated. Sometimes well-known formulas from queuing theory apply. For situations where there are no closed-form solutions, estimates can often be obtained through simulation. Simulations can be used to make more-realistic assumptions such as the distribution of the event arrivals. The estimates are only as good as the assumptions, but they can serve to provide rough values that can be used either in design or in evaluation; as better information is obtained, the estimates will improve.

A reasonably large number of parameters must be known or estimated to construct the queuing model shown in Figure 14.3. The model must then be solved or simulated to derive the expected latency. This is the cost side of the cost/benefit of performing a queuing analysis. The benefit side is that as a result of the analysis, there is an estimate for latency, and "what if" questions can be easily answered. The question for you to decide is whether having an estimate of the latency and the ability to answer "what if" questions is worth the cost of performing the analysis. One way to answer this question is to consider the importance of having an estimate for the latency prior to constructing either the system or a prototype that simulates an architecture under an assumed load. If having a small latency is a crucial requirement upon which the success of the system relies, then producing an estimate is appropriate.

Performance is a well-studied quality attribute with roots that extend beyond the computer industry. For example, the queuing model given in Figure 14.1 dates from the 1930s. Queuing theory has been applied to factory floors, to banking queues, and to many other domains. Models for real-time performance, such as rate monotonic analysis, also exist and have sophisticated analysis techniques.

Analyzing Availability

Another quality attribute with a well-understood analytic framework is availability.

Modeling an architecture for availability—or to put it more carefully, modeling an architecture to determine the availability of a system based on that architecture—is a matter of determining the failure rate and the recovery time. As you may recall from Chapter 5, availability can be expressed as

$$\frac{MTBF}{(MTBF + MTTR)}$$

This models what is known as steady-state availability, and it is used to indicate the uptime of a system (or component of a system) over a sufficiently long duration. In the equation, *MTBF* is the *mean time between failure*, which is derived based on the expected value of the implementation's failure probability density function (PDF), and *MTTR* refers to the *mean time to repair*.

Just as for performance, to model an architecture for availability, we need an architecture to analyze. So, suppose we want to increase the availability of a system that uses the broker pattern, by applying redundancy tactics. Figure 14.4

illustrates three well-known redundancy tactics from Chapter 5: *active redundancy*, *passive redundancy*, and *cold spare*. Our goal is to analyze each redundancy option for its availability, to help us choose one.

As you recall, each of these tactics introduces a backup copy of a component that will take over in case the primary component suffers a failure. In our case, a broker replica is employed as the redundant spare. The difference among them is how up to date with current events each backup keeps itself:

- In the case of active redundancy, the active and redundant brokers both receive identical copies of the messages received from the client and server proxies. The internal broker state is synchronously maintained between the active and redundant spare in order to facilitate rapid failover upon detection of a fault in the active broker.
- For the passive redundancy implementation, only the active broker receives and processes messages from the client and server proxies. When using this tactic, checkpoints of internal broker state are periodically transmitted from the active broker process to the redundant spare, using the checkpoint-based rollback tactic.
- Finally, when using the cold spare tactic, only the active broker receives and processes messages from the client and server proxies, because the redundant spare is in a dormant or even powered-off state. Recovery strategies using this tactic involve powering up, booting, and loading the broker implementation on the spare. In this scenario, the internal broker state is rebuilt organically, rather than via synchronous operation or checkpointing, as described for the other two redundancy tactics.

Suppose further that we will detect failure with the *heartbeat* tactic, where each broker (active and spare) periodically transmits a heartbeat message to a separate process responsible for fault detection, correlation, reporting, and recovery. This fault manager process is responsible for coordinating the transition of the active broker role from the failed broker process to the redundant spare.

You can now use the steady state model of availability to assign values for *MTBF* and *MTTR* for each of the three redundancy tactics we are considering. Doing so will be an exercise left to the reader (as you'll see when you reach the discussion questions for this chapter). Because the three tactics differ primarily in how long it takes to bring the backup copy up to speed, *MTTR* will be where the difference among the tactics shows up.

More sophisticated models of availability exist, based on probability. In these models, we can express a probability of failure during a period of time. Given a particular *MTBF* and a time duration *T*, the probability of failure *R* is given by

$$R(T) = e(\frac{-T}{MTBF})$$

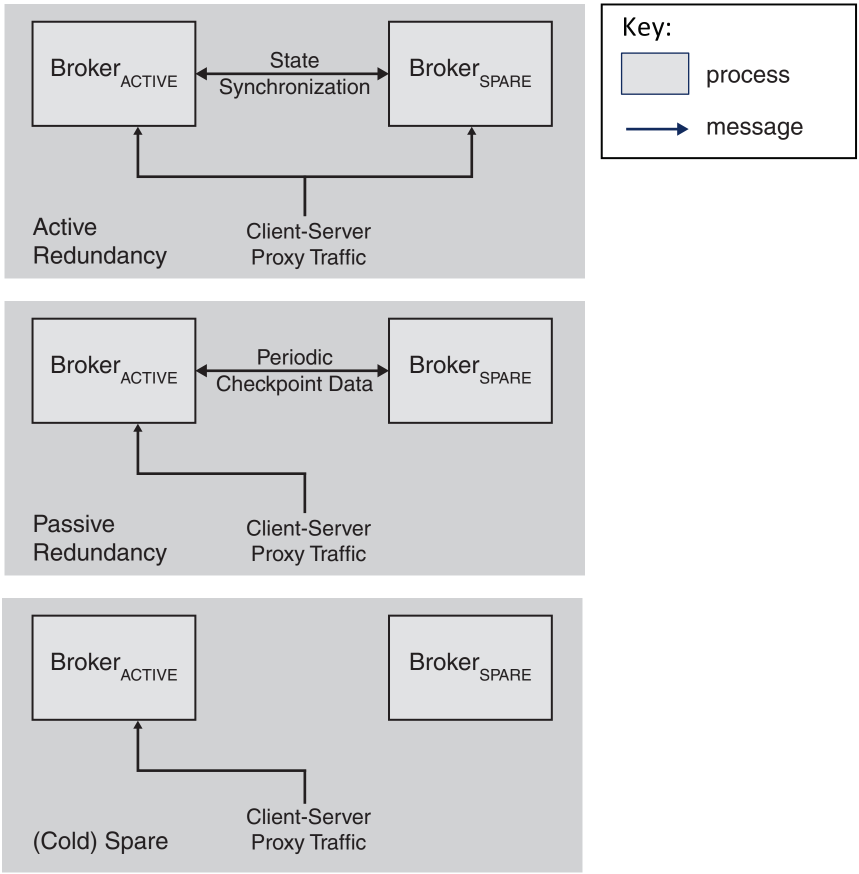


FIGURE 14.4 Redundancy tactics, as applied to a broker pattern

You will recall from Statistics 101 that:

- When two events A and B are independent, the probability that A or B will occur is the sum of the probability of each event: `P(A or B) = P(A) + P(B)`.
- When two events A and B are independent, the probability of both occurring is `P(A and B) = P(A) • P(B)`.
- When two events A and B are dependent, the probability of both occurring is `P(A and B) = P(A) • P(B|A)`, where the last term means "the probability of B occurring, given that A occurs."

We can apply simple probability arithmetic to an architecture pattern for availability to determine the probability of failure of the pattern given the probability of failure of the individual components (and an understanding of their dependency relations). For example, in an architecture pattern employing the passive redundancy tactic, let's assume that the failure of a component (which at any moment might be acting as either the primary or backup copy) is independent of a failure of its counterpart, and that the probability of failure of either is the same. Then the probability that both will fail is `F = (1 - a) **2`, where `a` is the availability of an individual component (assuming that failures are independent).

Still other models take into account different levels of failure severity and degraded operating states of the system. Although the derivation of these formulas is outside the scope of this chapter, you end up with formulas that look like the following for the three redundancy tactics we've been discussing, where the values C2 through C5 are references to the MTBF column of Table 14.1, D2 through D4 refer to the Active column, E2 through E3 refer to the Passive column, and F2 through F3 refer to the Spare column.

- Active redundancy:
  - Availability(MTTR): 1 –((SUM(C2:C5) + D3) × D2)/((C2 × (C2 + C4 + D3) + ((C2 + C4 + D2) × (C3 + C5)) + ((C2 + C4) × (C2 + C4 + D3))))
  - P(Degraded) = ((C3 + C5) × D2)/((C2 × (C2 + C4 + D3) + ((C2 + C4 + D2) × (C3 + C5)) + ((C2 + C4) × (C2 + C4 + D3))))
- Passive redundancy:
  - Availability(MTTR_passive) = 1 – ((SUM(C2:C5) + E3) × E2)/((C2 × (C2 + C4 + E3) + ((C2 + C4 + E2) × (C3 + C5)) + ((C2 + C4) × (C2 + C4 + E3))))
  - P(Degraded) = ((C3 + C5) × E2)/((C2 × (C2 + C4 + E3) + ((C2 + C4 + E2) × (C3 + C5)) + ((C2 + C4) × (C2 + C4 + E3))))
- Spare:
  - Availability(MTTR) = 1 – ((SUM(C2:C5) + F3) × F2)/((C2 × (C2 + C4 + F3) + ((C2 + C4 + F2) × (C3 + C5)) + ((C2 + C4) × (C2 + C4 + F3))))
  - P(Degraded) = ((C3 + C5) × F2)/((C2 × (C2 + C4 + F3) + ((C2 + C4 + F2) × (C3 + C5)) + ((C2 + C4) × (C2 + C4 + F3))))

Plugging in these values for the parameters to the equations listed above results in a table like Table 14.1, which can be easily calculated using any spreadsheet tool. Such a calculation can help in the selection of tactics.

TABLE 14.1 Calculated Availability for an Availability-Enhanced Broker Implementation

			MTTR (Seconds)		
Function	**Failure Severity**	**MTBF (Hours)**	**Active Redundancy (Hot Spare)**	**Passive Redundancy (Warm Spare)**	**Spare (Cold Spare)**
Hardware	1	250,000	1	5	900
	2	50,000	30	30	30
Software	1	50,000	1	5	900
	2	10,000	30	30	30
Availability			0.9999998	0.999990	0.9994

The Analytic Model Space

As we discussed in the preceding sections, there are a growing number of analytic models for some aspects of various quality attributes. One of the quests of software engineering is to have a sufficient number of analytic models for a sufficiently large number of quality attributes to enable prediction of the behavior of a designed system based on these analytic models. Table 14.2 shows our current status with respect to this quest for the seven quality attributes discussed in Chapters 5–11.

TABLE 14.2 A Summary of the Analytic Model Space

Quality Attribute	Intellectual Basis	Maturity/Gaps
Availability	Markov models; statistical models	Moderate maturity; mature in the hardware reliability domain, less mature in the software domain. Requires models that speak to state recovery and for which failure percentages can be attributed to software.
Interoperability	Conceptual framework	Low maturity; models require substantial human interpretation and input.
Modifiability	Coupling and cohesion metrics; cost models	Substantial research in academia; still requires more empirical support in real-world environments.
Performance	Queuing theory; real-time scheduling theory	High maturity; requires considerable education and training to use properly.
Security	No architectural models	
Testability	Component interaction metrics	Low maturity; little empirical validation.
Usability	No architectural models	

As the table shows, the field still has a great deal of work to do to achieve the quest for well-validated analytic models to predict behavior, but there is a great deal of activity in this area (see the "For Further Reading" section for additional papers). The remainder of this chapter deals with techniques that can be used *in addition to* analytic models.

14.2 Quality Attribute Checklists

For some quality attributes, checklists exist to enable the architect to test compliance or to guide the architect when making design decisions. Quality attribute checklists can come from industry consortia, from government organizations, or from private organizations. In large organizations they may be developed in house.

These checklists can be specific to one or more quality attributes; checklists for safety, security, and usability are common. Or they may be focused on a particular domain; there are security checklists for the financial industry, industrial process control, and the electric energy sector. They may even focus on some specific aspect of a single quality attribute: cancel for usability, as an example.

For the purposes of certification or regulation, the checklists can be used by auditors as well as by the architect. For example, two of the items on the checklist of the Payment Card Industry (PCI) are to only persist credit card numbers in an encrypted form and to never persist the security code from the back of the credit card. An auditor can ask to examine stored credit card data to determine whether it has been encrypted. The auditor can also examine the schema for data being stored to see whether the security code has been included.

This example reveals that design and analysis are often two sides of the same coin. By considering the kinds of analysis to which a system will be subjected (in this case, an audit), the architect will be led into making important early architectural decisions (making the decisions the auditors will want to find).

Security checklists usually have heavy process components. For example, a security checklist might say that there should be an explicit security policy within an organization, and a cognizant security officer to ensure compliance with the policy. They also have technical components that the architect needs to examine to determine the implications on the architecture of the system being designed or evaluated. For example, the following is an item from a security checklist generated by a group chartered by an organization of electric producers and distributors. It pertains to embedded systems delivering electricity to your home:

> A designated system or systems shall daily or on request obtain current version numbers, installation date, configuration settings, patch level on all elements of a [portion of the electric distribution] system,

In Search of a Grand Unified Theory for Quality Attributes

How do we create analytic models for those quality attribute aspects for which none currently exist? I do not know the answer to this question, but if we had a basis set for quality attributes, we would be in a better position to create and validate quality attribute models. By *basis set* I mean a set of orthogonal concepts that allow one to define the existing set of quality attributes. Currently there is much overlap among quality attributes; a basis set would enable discussion of tradeoffs in terms of a common set of fundamental and possibly quantifiable concepts. Once we have a basis set, we could develop analytic models for each of the elements of the set, and then an analytic model for a particular quality attribute becomes a composition of the models of the portions of the basis set that make up that quality attribute.

What are some of the elements of this basis set? Here are some of my candidates:

- *Time.* Time is the basis for performance, some aspects of availability, and some aspects of usability. Time will surely be one of the fundamental concepts for defining quality attributes.
- *Dependencies among structural elements.* Modifiability, security, availability, and performance depend in some form or another on the strength of connections among various structural elements. Coupling is a form of dependency. Attacks depend on being able to move from one compromised element to a currently uncompromised element through some dependency. Fault propagation depends on dependencies. And one of the key elements of performance analysis is the dependency of one computation on another. Enumeration of the fundamental forms of dependency and their properties will enable better understanding of many quality attributes and their interaction.
- *Access.* How does a system promote or deny access through various mechanisms? Usability is concerned with allowing smooth access for humans; security is concerned with allowing smooth access for some set of requests but denying access to another set of requests. Interoperability is concerned with establishing connections and accessing information. Race conditions, which undermine availability, come about through unmediated access to critical computations.

These are some of my candidates. I am sure there are others. The general problem is to define a set of candidates for the basis set and then show how current definitions of various quality attributes can be recast in terms of the elements of the basis set. I am convinced that this is a problem that needs to be solved prior to making substantial progress in the quest for a rich enough set of analytic models to enable prediction of system behavior across the quality attributes important for a system.

—*LB*

compare these with inventory and configuration databases, and log all discrepancies.

This kind of rule is intended to detect malware masquerading as legitimate components of a system. The architect will look at this item and conclude the following:

- The embedded portions of the system should be able to report their version number, installation date, configuration settings, and patch levels. One technique for doing this is to use "reflection" for each component in the system. Reflection now becomes one of the important patterns used in this system.
- Each software update or patch should maintain this information. One technique for doing this is to have automated update and patch mechanisms. The architecture could also realize this functionality through reflection.
- A system must be designated to query the embedded components and persist the information. This means
  - There must be overall inventory and configuration databases.
  - Logs of discrepancies between current values and overall inventory must be generated and sent to appropriate recipients.
  - There must be network connections to the embedded components. This affects the network topology.

The creation of quality attribute checklists is usually a time-consuming activity, undertaken by multiple individuals and typically refined and evolved over time. Domain specialists, quality attribute specialists, and architects should all contribute to the development and validation of these checklists.

The architect should treat the items on an applicable checklist as requirements, in that they need to be understood and prioritized. Under particular circumstances, an item in a checklist may not be met, but the architect should have a compelling case as to why it is not.

14.3 Thought Experiments and Back-of-the-Envelope Analysis

A thought experiment is a fancy name for the kinds of discussions that developers and architects have on a daily basis in their offices, in their meetings, over lunch, over whiteboards, in hallways, and around the coffee machine. One of the participants might draw two circles and an arrow on the whiteboard and make an assertion about the quality attribute behavior of these two circles and the arrow in a particular context; a discussion ensues. The discussion can last for a long time, especially if the two circles are augmented with a third and one more arrow, or if some of the assumptions underlying a circle or an arrow are still in flux. In this section, we describe this process somewhat more formally.

The level of formality one would use in performing a thought experiment is, as with most techniques discussed in this book, a question of context. If two people with a shared understanding of the system are performing the thought experiment for their own private purposes, then circles and lines on a whiteboard are adequate, and the discussion proceeds in a kind of shorthand. If a third person is to review the results and the third person does not share the common understanding, then sufficient details must be captured to enable the third person to understand the argument—perhaps a quick legend and a set of properties need to be added to the diagram. If the results are to be included in documentation as design rationale, then even more detail must be captured, as discussed in Chapter 18. Frequently such thought experiments are accompanied by rough analyses—back-of-the-envelope analyses—based on the best data available, based on past experiences, or even based on the guesses of the architects, without too much concern for precision.

The purpose of thought experiments and back-of-the-envelope analysis is to find problems or confirmation of the nonexistence of problems in the quality attribute requirements as applied to sunny-day use cases. That is, for each use case, consider the quality attribute requirements that pertain to that use case and analyze the use case with the quality attribute requirements in mind. Models and checklists focus on one quality attribute. To consider other quality attributes, one must model or have a checklist for the second quality attribute and understand how those models interact. A thought experiment may consider several of the quality attribute requirements simultaneously; typically it will focus on just the most important ones.

The process of creating a thought experiment usually begins with listing the steps associated with carrying out the use case under consideration; perhaps a sequence diagram is employed. At each step of the sequence diagram, the (mental) question is asked: What can go wrong with this step with respect to any of the quality attribute requirements? For example, if the step involves user input, then the possibility of erroneous input must be considered. Also the user may not have been properly authenticated and, even if authenticated, may not be authorized to provide that particular input. If the step involves interaction with another system, then the possibility that the input format will change after some time must be considered. The network passing the input to a processor may fail; the processor performing the step may fail; or the computation to provide the step may fail, take too long, or be dependent on another computation that may have had problems. In addition, the architect must ask about the frequency of the input, and the anticipated distribution of requests (e.g., Are service requests regular and predictable or irregular and "bursty"?), other processes that might be competing for the same resources, and so forth. These questions go on and on.

For each possible problem with respect to a quality attribute requirement, the follow-on questions consist of things like these:

- Are there mechanisms to detect that problem?

- Are there mechanisms to prevent or avoid that problem?
- Are there mechanisms to repair or recover from that problem if it occurs?
- Is this a problem we are willing to live with?

The problems hypothesized are scrutinized in terms of a cost/benefit analysis. That is, what is the cost of preventing this problem compared to the benefits that accrue if the problem does not occur?

As you might have gathered, if the architects are being thorough and if the problems are significant (that is, they present a large risk for the system), then these discussions can continue for a long time. The discussions are a normal portion of design and analysis and will naturally occur, even if only in the mind of a single designer. On the other hand, the time spent performing a particular thought experiment should be bounded. This sounds obvious, but every grey-haired architect can tell you war stories about being stuck in endless meetings, trapped in the purgatory of "analysis paralysis."

Analysis paralysis can be avoided with several techniques:

- "Time boxing": setting a deadline on the length of a discussion.
- Estimating the cost if the problem occurs and not spending more than that cost in the analysis. In other words, do not spend an inordinate amount of time in discussing minor or unlikely potential problems.

Prioritizing the requirements will help both with the cost estimation and with the time estimation.

14.4 Experiments, Simulations, and Prototypes

In many environments it is virtually impossible to do a purely top-down architectural design; there are too many considerations to weigh at once and it is too hard to predict all of the relevant technological barriers. Requirements may change in dramatic ways, or a key assumption may not be met: We have seen cases where a vendor-provided API did not work as specified, or where an API exposing a critical function was simply missing.

Finding the sweet spot within the enormous architectural design space of complex systems is not feasible by reflection and mathematical analysis alone; the models either aren't precise enough to deal with all of the relevant details or are so complicated that they are impractical to analyze with tractable mathematical techniques.

The purpose of *experiments*, *simulations*, and *prototypes* is to provide alternative ways of analyzing the architecture. These techniques are invaluable in resolving tradeoffs, by helping to turn unknown architectural parameters into constants or ranges. For example, consider just a few of the questions that might occur when creating a web-conferencing system—a distributed client-server infrastructure with real-time constraints:

- Would moving to a distributed database from local flat files negatively impact feedback time (latency) for users?
- How many participants could be hosted by a single conferencing server?
- What is the correct ratio between database servers and conferencing servers?

These sorts of questions are difficult to answer analytically. The answers to these questions rely on the behavior and interaction of third-party components such as commercial databases, and on performance characteristics of software for which no standard analytical models exist. The approach used for the web-conferencing architecture was to build an extensive testing infrastructure that supported simulations, experiments, and prototypes, and use it to compare the performance of each incremental modification to the code base. This allowed the architect to determine the effect of each form of improvement before committing to including it in the final system. The infrastructure includes the following:

- A client simulator that makes it appear as though tens of thousands of clients are simultaneously interacting with a conferencing server.
- Instrumentation to measure load on the conferencing server and database server with differing numbers of clients.

The lesson from this experience is that experimentation can often be a critical precursor to making significant architectural decisions. Experimentation must be built into the development process: building experimental infrastructure can be time-consuming, possibly requiring the development of custom tools. Carrying out the experiments and analyzing their results can require significant time. These costs must be recognized in project schedules.

14.5 Analysis at Different Stages of the Life Cycle

Depending on your project's state of development, different forms of analysis are possible. Each form of analysis comes with its own costs. And there are different levels of confidence associated with each analysis technique. These are summarized in Table 14.3.

TABLE 14.3 Forms of Analysis, Their Life-Cycle Stage, Cost, and Confidence in Their Outputs

Life-Cycle Stage	Form of Analysis	Cost	Confidence
Requirements	Experience-based analogy	Low	Low–High
Requirements	Back-of-the-envelope	Low	Low–Medium
Architecture	Thought experiment	Low	Low–Medium
Architecture	Checklist	Low	Medium
Architecture	Analytic model	Low–Medium	Medium
Architecture	Simulation	Medium	Medium
Architecture	Prototype	Medium	Medium–High
Implementation	Experiment	Medium–High	Medium–High
Fielded System	Instrumentation	Medium–High	High

The table shows that analysis performed later in the life cycle yields results that merit high confidence. However, this confidence comes at a price. First, the cost of performing the analysis also tends to be higher. But the cost of changing the system to fix a problem uncovered by analysis skyrockets later in the life cycle.

Choosing an appropriate form of analysis requires a consideration of all of the factors listed in Table 14.3: What life-cycle stage are you currently in? How important is the achievement of the quality attribute in question and how worried are you about being able to achieve the goals for this attribute? And finally, how much budget and schedule can you afford to allocate to this form of risk mitigation? Each of these considerations will lead you to choose one or more of the analysis techniques described in this chapter.

14.6 Summary

Analysis of an architecture enables early prediction of a system's qualities. We can analyze an architecture to see how the system or systems we build from it will perform with respect to their quality attribute goals. Some quality attributes, most notably performance and availability, have well-understood, time-tested analytic models that can be used to assist in quantitative analysis. Other quality attributes have less sophisticated models that can nevertheless help with predictive analysis.

For some quality attributes, checklists exist to enable the architect to test compliance or to guide the architect when making design decisions. Quality attribute checklists can come from industry consortia, from government organizations, or from private organizations. In large organizations they may be developed in house. The architect should treat the items on an applicable checklist as requirements, in that they need to be understood and prioritized.

Thought experiments and back-of-the-envelope analysis can often quickly help find problems or confirm the nonexistence of problems with respect to quality attribute requirements. A thought experiment may consider several of the quality attribute requirements simultaneously; typically it will focus on just the most important ones. Experiments, simulations, and prototypes allow the exploration of tradeoffs, by helping to turn unknown architectural parameters into constants or ranges whose values may be measured rather than estimated.

Depending on your project's state of development, different forms of analysis are possible. Each form of analysis comes with its own costs and its own level of confidence associated with each analysis technique.

14.7 For Further Reading

There have been many papers and books published describing how to build and analyze architectural models for quality attributes. Here are just a few examples.

Availability

Many availability models have been proposed that operate at the architecture level of analysis. Just a few of these are [Gokhale 05] and [Yacoub 02].

A discussion and comparison of different black-box and white-box models for determining software reliability can be found in [Chandran 10].

A book relating availability to disaster recovery and business recovery is [Schmidt 10].

Interoperability

An overview of interoperability activities can be found in [Brownsword 04].

Modifiability

Modifiability is typically measured through complexity metrics. The classic work on this topic is [Chidamber 94].

More recently, analyses based on design structure matrices have begun to appear [MacCormack 06].

Performance

Two of the classic works on software performance evaluation are [Smith 01] and [Klein 93].

A broad survey of architecture-centric performance evaluation approaches can be found in [Koziolek 10].

Security

Checklists for security have been generated by a variety of groups for different domains. See for example:

- Credit cards, generated by the Payment Card Industry: www.pcisecuritystandards.org/security_standards/
- Information security, generated by the National Institute of Standards and Technology (NIST): [NIST 09].
- Electric grid, generated by Advanced Security Acceleration Project for the Smart Grid: www.smartgridipedia.org/index.php/ASAP-SG
- Common Criteria. An international standard (ISO/IEC 15408) for computer security certification: www.commoncriteriaportal.org

Testability

Work in measuring testability from an architectural perspective includes measuring testability as the measured complexity of a class dependency graph derived from UML class diagrams, and identifying class diagrams that can lead to code that is difficult to test [Baudry 05]; and measuring controllability and observability as a function of data flow [Le Traon 97].

Usability

A checklist for usability can be found at www.stcsig.org/usability/topics/articles/he-checklist.html

Safety

A checklist for safety is called the Safety Integrity Level: en.wikipedia.org/wiki/Safety_Integrity_Level

Applications of Modeling and Analysis

For a detailed discussion of a case where quality attribute modeling and analysis played a large role in determining the architecture as it evolved through a number of releases, see [Graham 07].

14.8 Discussion Questions

1. Build a spreadsheet for the steady-state availability equation *MTBF* / (*MTBF* + *MTTR*). Plug in different but reasonable values for *MTBF* and *MTTR* for each of the active redundancy, passive redundancy, and cold spare tactics. Try values for *MTBF* that are very large compared to *MTTR*, and also try values for *MTBF* that are much closer in size to *MTTR*. What do these tell you about which tactics you might want to choose for availability?

2. Enumerate as many responsibilities as you can that need to be carried out for providing a "cancel" operation in a user interface. *Hint:* There are at least 21 of them, as indicated in a publication by (*strong hint!*) one of the authors of this book whose last name (*unbelievably strong hint!*) begins with "B."

3. The M/M/1 (look it up!) queuing model has been employed in computing systems for decades. Where in your favorite computing system would this model be appropriate to use to predict latency?

4. Suppose an architect produced Figure 14.5 while you were sitting watching him. Using thought experiments, how can you determine the performance and availability of this system? What assumptions are you making and what conclusions can you draw? How definite are your conclusions?

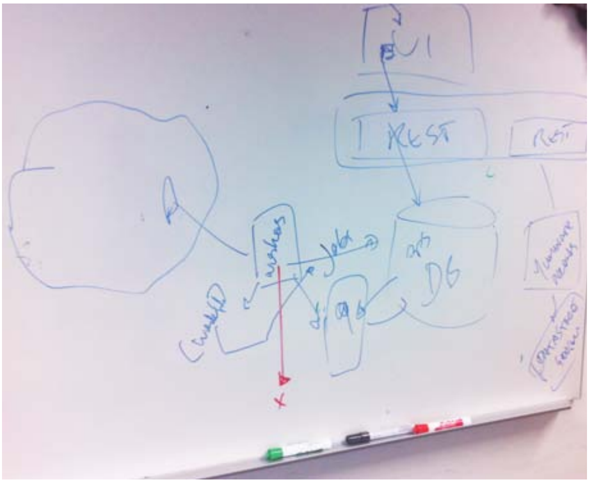


FIGURE 14.5 Capture of a whiteboard sketch from an architect

This page intentionally left blank