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 publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States, please contact:

International Sales
international@pearson.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.
First printing, September 2012

Contents

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.

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.

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.

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.

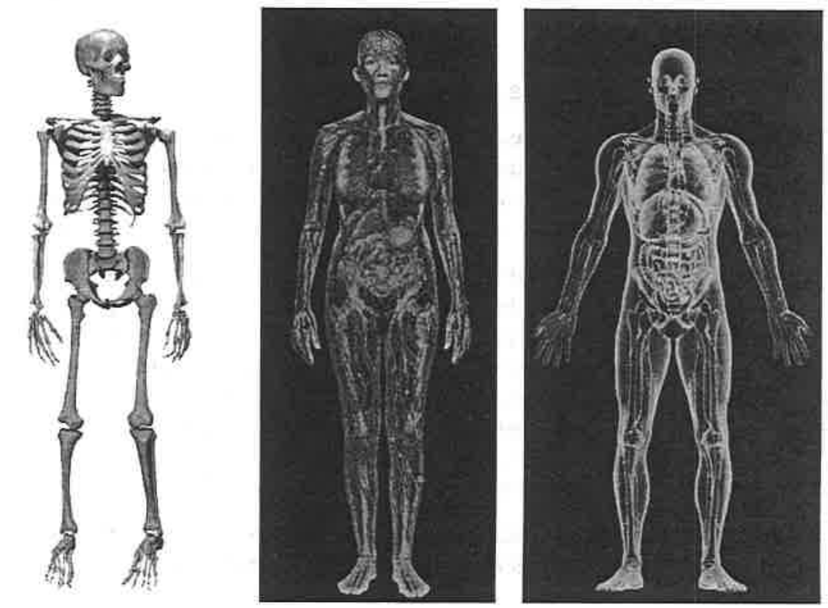

FIGURE 1.1 Physiological structures (Getty images: Brand X Pictures [skeleton], Don Farrall [woman], Mads Abildgaard [man])

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

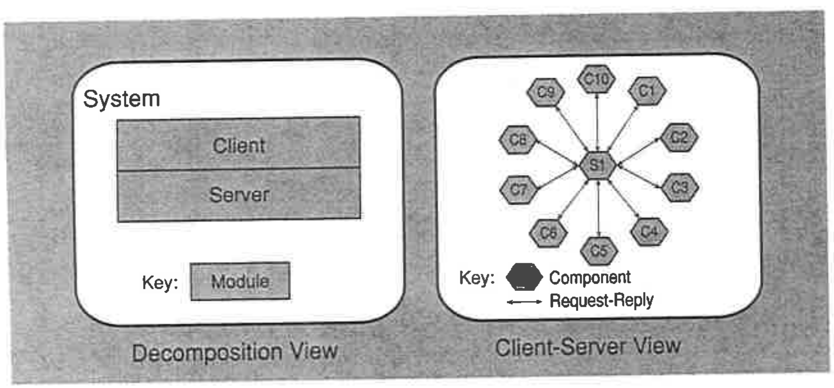

FIGURE 1.2 Two views of a client-server system

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.