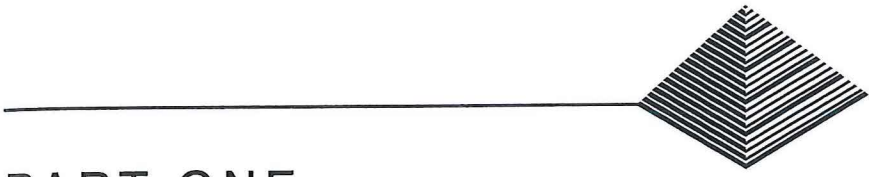

PART ONE

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.

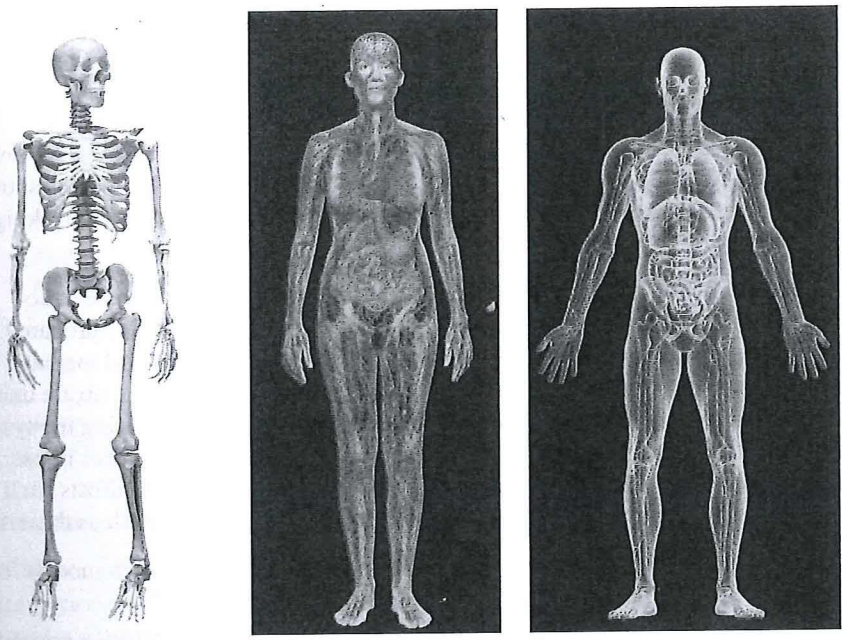

FIGURE 1.1 Physiological structures (Getty images: Brand X Pictures [skeleton], Don Farrall [woman], Mads Abildgaard [man])

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

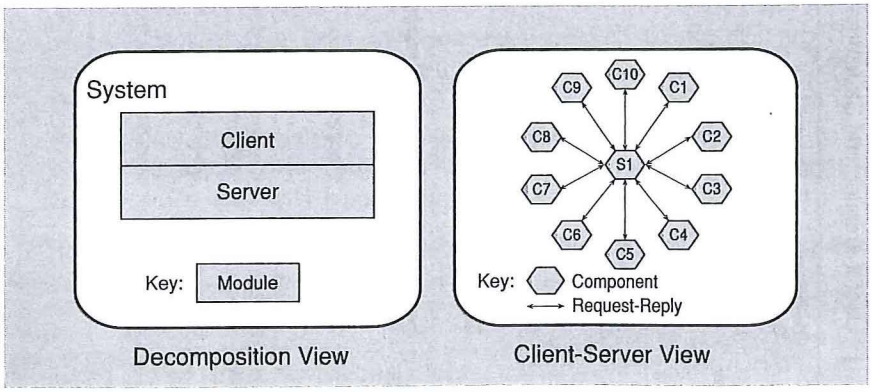

FIGURE 1.2 Two views of a client-server system

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.

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?