What Is Software Architecture?

*with Linda Northrop*

*If a project has not achieved a system architecture, including its rationale, the project should not proceed to full-scale system development. Specifying the architecture as a deliverable enables its use throughout the development and maintenance process. — Barry Boehm [Boehm 95]*

In Chapter 1, we explained that architecture plays a pivotal role in allowing an organization to meet its business goals. Architecture commands a price (the cost of its careful development), but it pays for itself handsomely by enabling the organization to achieve its system goals and expand its software capabilities. Architecture is an asset that holds tangible value to the developing organization beyond the project for which it was created.

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 in addition to the value returned to the enterprise in the ways described in Chapter 1.

### 2.1 What Software Architecture Is and What It Isn’t

Figure 2.1, taken from a system description for an underwater acoustic simulation, purports to describe that system’s “top-level architecture” and is precisely

*Note:* Linda Northrop is a program director at Carnegie Mellon University’s Software Engineering Institute.
the kind of diagram most often displayed to help explain an architecture. Exactly what can we tell from it?

- The system consists of four elements.
- Three of the elements—Prop Loss Model (MODP), Reverb Model (MODR), and Noise Model (MODN)—might have more in common with each other than with the fourth—Control Process (CP)—because they are positioned next to each other.
- All of the elements apparently have some sort of relationship with each other, since the diagram is fully connected.

Is this an architecture? Assuming (as many definitions do) that architecture is a set of components (of which we have four) and connections among them (also present), this diagram seems to fill the bill. However, even if we accept the most primitive definition, what can we not tell from the diagram?

- What is the nature of the elements? What is the significance of their separation? Do they run on separate processors? Do they run at separate times? Do the elements consist of processes, programs, or both? Do they represent ways in which the project labor will be divided, or do they convey a sense of runtime separation? Are they objects, tasks, functions, processes, distributed programs, or something else?
- What are the responsibilities of the elements? What is it they do? What is their function in the system?
- What is the significance of the connections? Do the connections mean that the elements communicate with each other, control each other, send data to each other, use each other, invoke each other, synchronize with each other, share
some information-hiding secret with each other, or some combination of
these or other relations? What are the mechanisms for the communication?
What information flows across the mechanisms, whatever they may be?

- What is the significance of the layout? Why is CP on a separate level? Does
it call the other three elements, and are the others not allowed to call it?
Does it contain the other three in an implementation unit sense? Or is there
simply no room to put all four elements on the same row in the diagram?

We must raise these questions because unless we know precisely what the
elements are and how they cooperate to accomplish the purpose of the system,
diagrams such as these are not much help and should be regarded skeptically.

This diagram does not show a software architecture, at least not in any use-
ful way. The most charitable thing we can say about such diagrams is that they
represent a start. We now define what does constitute a software architecture:

The software architecture of a program or computing system is the structure
or structures of the system, which comprise software elements, the externally
visible properties of those elements, and the relationships among them.¹

“Externally visible” properties are those assumptions other elements can
make of an element, such as its provided services, performance characteristics,
fault handling, shared resource usage, and so on. Let’s look at some of the impli-
cations of this definition in more detail.

First, architecture defines software elements. The architecture embodies
information about how the elements relate to each other. This means that it spe-
cifically omits certain information about elements that does not pertain to their
interaction. Thus, an architecture is foremost an abstraction of a system that sup-
presses details of elements that do not affect how they use, are used by, relate to,
or interact with other elements. In nearly 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—those having to do solely with internal implementa-
tion—are not architectural.

Second, the definition makes clear that systems can and do comprise more
than one structure and that no one structure can irrefutably claim to be the archi-
tecture. For example, all nontrivial projects are partitioned into implementation
units; these units are given specific responsibilities and are frequently the basis of
work assignments for programming teams. This type of element comprises pro-
grams and data that software in other implementation units can call or access, and
programs and data that are private. In large projects, these elements are almost

¹ This is a slight change from the first edition. There the primary building blocks were
called “components,” a term that has since become closely associated with the component-
based software engineering movement, taking on a decidedly runtime flavor. “Element”
was chosen here to convey something more general.
certainly subdivided for assignment to subteams. This is one kind of structure often used to describe a system. It is very static in that it focuses on the way the system's functionality is divided up and assigned to implementation teams.

Other structures are much more focused on the way the elements interact with each other at runtime to carry out the system's function. Suppose the system is to be built as a set of parallel processes. The processes that will exist at runtime, the programs in the various implementation units described previously that are strung together sequentially to form each process, and the synchronization relations among the processes form another kind of structure often used to describe a system.

Are any of these structures alone the architecture? No, although they all convey architectural information. The architecture consists of these structures as well as many others. This example shows that since architecture can comprise more than one kind of structure, there is more than one kind of element (e.g., implementation unit and processes), more than one kind of interaction among elements (e.g., subdivision and synchronization), and even more than one context (e.g., development time versus runtime). By intention, the definition does not specify what the architectural elements and relationships are. Is a software element an object? A process? A library? A database? A commercial product? It can be any of these things and more.

Third, the definition implies that every computing system with software has a software architecture because every system can be shown to comprise elements and the relations among them. In the most trivial case, a system is itself a single element—uninteresting and probably nonuseful 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. Unfortunately, an architecture can exist independently of its description or specification, which raises the importance of architecture documentation (described in Chapter 9) and architecture reconstruction (discussed in Chapter 10).

Fourth, the behavior of each element is part of the architecture insofar as that behavior can be observed or discerned from the point of view of another element. Such behavior is what allows elements to interact with each other, which is clearly part of the architecture. This is another reason that the box-and-line drawings that are passed off as architectures are not architectures at all. They are simply box-and-line drawings—or, to be more charitable, they serve as cues to provide more information that explains what the elements shown actually do. 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 observer's mind and relies on information that is not present. We
do not mean that the exact behavior and performance of every element must be 
documented in all circumstances; however, to the extent that an element's behav-
ior influences how another element must be written to interact with it or influ-
ences the acceptability of the system as a whole, this behavior is part of the 
software architecture.

Finally, the definition is indifferent as to whether the architecture for a sys-
tem is a good one or a bad one, meaning that it will allow or prevent the system 
from meeting its behavioral, performance, and life-cycle requirements. 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 hoping for 
the best—so this raises the importance of architecture evaluation (Chapters 11 
and 12) and architecture design (Chapter 7).

2.2 Other Points of View

Software architecture is a growing but still young discipline; hence, it has no sin-
gle, accepted definition. On the other hand, there is no shortage of definitions. 
Most of those commonly circulated are consistent in their themes—structure, ele-
ments, and connections among them—but they vary widely in the details and are 
not interchangeable.

The study of software architecture has evolved by observation of the design 
principles that designers follow and the actions that they take when working on 
real systems. It is an attempt to abstract the commonalities inherent in system 
design, and as such it must account for a wide range of activities, concepts, meth-
ods, approaches, and results. For that reason, other definitions of architecture are 
present in the software engineering community, and because you are likely to 
encounter some of them, you should understand their implications and be able to 
discuss them. A few of the most often heard definitions follow.

- Architecture is high-level design. This is true enough, in the sense that a horse 
is a mammal, but the two are not interchangeable. Other tasks associated 
with design are not architectural, such as deciding on important data struc-
tures that will be encapsulated. The interface to those data structures is 
decidedly an architectural concern, but their actual choice is not.

- Architecture is the overall structure of the system. This common refrain 
implies (incorrectly) that systems have but one structure. We know this to be 
false, and, if someone takes this position, it is usually entertaining to ask 
which structure they mean. The point has more than pedagogic significance. 
As we will see later, the different structures provide the critical engineering 
leverage points to imbue a system with the quality attributes that will render 
it a success or failure. The multiplicity of structures in an architecture lies at 
the heart of the concept.
• Architecture is the structure of the components of a program or system, their interrelationships, and the principles and guidelines governing their design and evolution over time. This is one of a number of process-centered definitions that include ancillary information such as principles and guidelines. Many people claim that architecture includes a statement of stakeholder needs and a rationale for how those needs are met. We agree that gathering such information is essential and a matter of good professional practice. However, we do not consider them part of the architecture per se any more than an owner's manual for a car is part of the car. Any system has an architecture that can be discovered and analyzed independently of any knowledge of the process by which the architecture was designed or evolved.

• Architecture is components and connectors. Connectors imply a runtime mechanism for transferring control and data around a system. Thus, this definition concentrates on the runtime architectural structures. A UNIX pipe is a connector, for instance. This makes the non-runtime architectural structures (such as the static division into responsible units of implementation discussed earlier) second-class citizens. They aren't second class but are every bit as critical to the satisfaction of system goals. When we speak of "relationships" among elements, we intend to capture both runtime and non-runtime relationships.

At the root of all the discussion about software architecture is a focus on reasoning about the structural system issues. And although architecture is sometimes used to mean a certain architectural pattern, such as client-server, and sometimes refers to a field of study, such as a book about architecture, it is most often used to describe structural aspects of a particular system. That is what we have attempted to capture in our definition.

2.3 Architectural Patterns, Reference Models, and Reference Architectures

Between box-and-line sketches that are the barest of starting points and full-fledged architectures, with all of the appropriate information about a system filled in, lie a host of intermediate stages. Each stage represents the outcome of a set of architectural decisions, the binding of architectural choices. Some of these intermediate stages are very useful in their own right. Before discussing architectural structures, we define three of them.

1. An architectural pattern is a description of element and relation types together with a set of constraints on how they may be used. A pattern can be thought of as a set of constraints on an architecture—on the element types and their patterns of interaction—and these constraints define a set or family of architectures.
that satisfy them. For example, client-server is a common architectural pattern. Client and server are two element types, and their coordination is described in terms of the protocol that the server uses to communicate with each of its clients. Use of the term client-server implies only that multiple clients exist; the clients themselves are not identified, and there is no discussion of what functionality, other than implementation of the protocols, has been assigned to any of the clients or to the server. Countless architectures are of the client-server pattern under this (informal) definition, but they are different from each other. An architectural pattern is not an architecture, then, but it still conveys a useful image of the system—it imposes useful constraints on the architecture and, in turn, on the system.

One of the most useful aspects of patterns is that they exhibit known quality attributes. This is why the architect chooses a particular pattern and not one at random. Some patterns represent known solutions to performance problems, others lend themselves well to high-security systems, still others have been used successfully in high-availability systems. Choosing an architectural pattern is often the architect's first major design choice.

The term architectural style has also been widely used to describe the same concept.

2. A reference model is a division of functionality together with data flow between the pieces. A reference model is a standard decomposition of a known problem into parts that cooperatively solve the problem. Arising from experience, reference models are a characteristic of mature domains. Can you name the standard parts of a compiler or a database management system? Can you explain in broad terms how the parts work together to accomplish their collective purpose? If so, it is because you have been taught a reference model of these applications.

3. A reference architecture is a reference model mapped onto software elements (that cooperatively implement the functionality defined in the reference model) and the data flows between them. Whereas a reference model divides the functionality, a reference architecture is the mapping of that functionality onto a system decomposition. The mapping may be, but by no means necessarily is, one to one. A software element may implement part of a function or several functions.

Reference models, architectural patterns, and reference architectures are not architectures; they are useful concepts that capture elements of an architecture. Each is the outcome of early design decisions. The relationship among these design elements is shown in Figure 2.2.

People often make analogies to other uses of the word architecture, about which they have some intuition. They commonly associate architecture with physical structure (buildings, streets, hardware) and physical arrangement. A building architect must design a building that provides accessibility, aesthetics, light, maintainability, and so on. A software architect must design a system that provides concurrency, portability, modifiability, usability, security, and the like, and that reflects consideration of the tradeoffs among these needs.
Analogies between buildings and software systems should not be taken too far, as they break down fairly quickly. Rather, they help us understand that the viewer’s perspective is important and that structure can have different meanings depending on the motivation for examining it. A precise definition of software architecture is not nearly as important as what investigating the concept allows us to do.

2.4 Why Is Software Architecture Important?

Chapter 1 covered the importance of architecture to an enterprise. In this chapter, we focus on why architecture matters from a technical perspective. In that context, there are fundamentally three reasons for software architecture’s importance.

1. 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 mutual understanding, negotiation, consensus, and communication.

2. Early design decisions. Software architecture manifests the earliest design decisions about a system, and these early bindings carry weight far out of proportion to their individual gravity with respect to the system’s remaining development, its deployment, and its maintenance life. It is also the earliest point at which design decisions governing the system to be built can be analyzed.

3. Transferable abstraction of a system. Software architecture constitutes a relatively small, intellectually graspable model for how a system is structured and how its elements work together, and this model is transferable across systems. In particular, it can be applied to other systems exhibiting similar quality attribute and functional requirements and can promote large-scale re-use.

We will address each of these points in turn.
ARCHITECTURE IS THE VEHICLE FOR
STAKEHOLDER COMMUNICATION

Each stakeholder of a software system—customer, user, project manager, coder, tester, and so on—is concerned with different system characteristics that are affected by the architecture. For example, the user is concerned that the system is reliable and available when needed; the customer is concerned that the architecture can be implemented on schedule and to budget; the manager is worried (as well as 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 (see the sidebar What Happens When I Push This Button?). 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 Part Three, both depends on this level of communication and enhances it.

“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 so 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 of 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 its graphical rendition) had sparked debate. It is natural to think of architecture as the basis for communication among some of the stakeholders besides architects and developers. Managers, for example, use it 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 system understanding?

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 even though he was tired and wanted to go home, it was the first slide on architecture that 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 the sidebar Their Solution Just Won't Work in Chapter 11, we describe an architecture evaluation exercise in which 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. 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 can ameliorate this, but for a variety of reasons they are not always created or available.
In their absence, a specification of the architecture often triggers questions and improves clarity. It is probably more prudent to recognize this than to resist it. In Chapter 11, we point out that one of the benefits of an architecture evaluation is the clarification and prioritization of requirements.

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 would not have felt like a fish out of water, asking his question at a clearly inappropriate moment. Of course, he could always go into real estate.

— PCC

ARCHITECTURE MANIFESTS THE EARLIEST SET OF DESIGN DECISIONS

Software architecture represents a system's earliest set of design decisions. These early decisions are the most difficult to get correct and the hardest to change later in the development process, and they have the most far-reaching effects.

The Architecture Defines Constraints on Implementation. An implementation exhibits an architecture if it conforms to the structural design decisions described by the architecture. This means that the implementation must be divided into the prescribed elements, the elements must interact with each other in the prescribed fashion, and each element must fulfill its responsibility to the others as dictated by the architecture.

Resource allocation decisions also constrain implementations. These decisions may be invisible to implementors working on individual elements. The constraints permit a separation of concerns that allows management decisions to make the best use of personnel and computational capacity. Element builders must be fluent in the specification of their individual elements but not in architectural tradeoffs. Conversely, architects need not be experts in all aspects of algorithm design or the intricacies of the programming language, but they are the ones responsible for the architectural tradeoffs.

The Architecture Dictates Organizational Structure. Not only does architecture prescribe the structure of the system being developed, but that structure becomes engrained in the structure of the development project (and sometimes, as mentioned in Chapter 1, the structure of the entire organization). The normal method for dividing up the labor in a large system is to assign different groups different portions of the system to construct. This is called the work breakdown structure of a system. Because the system architecture includes the highest-level decomposition of the system, it is typically used as the basis for the work breakdown structure, which 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 minutiae such as how the project intranet is organized and how many team picnics there are. Teams communicate with each other in terms of the interface specifications to the major elements. The maintenance activity, when launched, will also reflect the software structure, with teams formed to maintain specific structural elements.

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 them can become expensive. Tracking progress on a collection of tasks being distributed also becomes much more difficult.

Once the architecture has been agreed on, then, it becomes almost impossible, for managerial and business reasons, to modify it. This is one argument (among many) for carrying out a comprehensive evaluation before freezing the software architecture for a large system.

The Architecture Inhibits or Enables 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. Chapter 5 will delve into the relationship between architectures and quality in more detail, but for now keep the following in mind:

- If your system requires high performance, you need to manage the time-based behavior of elements and the frequency and volume of inter-element communication.
- If modifiability is important, you need to assign responsibilities to elements such that changes to the system do not have far-reaching consequences.
- If your system must be highly secure, you need to manage and protect inter-element communication and which elements are allowed to access which information. You may also need to introduce specialized elements (such as a trusted kernel) into the architecture.
- If you believe scalability will be needed in your system, you have to carefully localize the use of resources to facilitate the introduction of higher-capacity replacements.
- If your project needs to deliver incremental subsets of the system, you must carefully manage inter-component usage.
- If you want the elements of your system to be re-usable in other systems, 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. It is important to understand, however, that architecture alone cannot guarantee functionality or quality. Poor downstream design or implementation decisions can always undermine an adequate architectural design. Decisions at all stages of
the life cycle—from high-level design to coding and implementation—affect system quality. Therefore, quality is not completely a function of architectural design. To ensure quality, a good architecture is necessary, but not sufficient.

Predicting System Qualities by Studying the Architecture. Is it 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? If the answer were no, choosing an architecture would be a hopeless task—random selection 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. Architecture evaluation techniques such as the Architecture Tradeoff Analysis Method of Chapter 11 support top-down insight into the attributes of software product quality that is made possible (and constrained) by software architectures.

The Architecture Makes It Easier to Reason about and Manage Change. The software development community is coming to grips with the fact that roughly 80 percent of a typical software system's cost occurs after initial deployment. A corollary of this statistic is that most systems that people work on are in this phase. Many if not most programmers and designers never work on new development—they work under the constraints of the existing body of code. Software systems change over their lifetimes; they do so often and often 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. A nonlocal change requires multiple element modifications but leaves the underlying architectural approach intact. An architectural change affects the fundamental ways in which the elements interact with each other—the pattern of the architecture—and will probably require changes all over the system. Obviously, local changes are the most desirable, and so an effective architecture is one in which the most likely changes are also the easiest 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 are in the job description for an architect. Reasoning about the architecture can provide the insight necessary to make decisions about proposed changes.

The Architecture Helps in Evolutionary Prototyping. Once an architecture has been defined, it can be analyzed and prototyped as a skeletal system. This aids the development process in two ways.

1. The system is executable early in the product's life cycle. Its fidelity increases as prototype parts are replaced by complete versions of the software. These prototype parts can be a lower-fidelity version of the final functionality, or they can be surrogates that consume and produce data at the appropriate rates.
2. A special case of having the system executable early is that potential performance problems can be identified early in the product's life cycle.

Each of these benefits reduces the risk in the project. 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.

**The Architecture Enables More Accurate Cost and Schedule Estimates.**
Cost and schedule estimates are an important management tool to enable the manager to acquire the necessary resources and to understand whether a project is in trouble. Cost estimations based on an understanding of the system pieces are, inherently, more accurate than those based on overall system knowledge. As we have said, the organizational structure of a project is based on its architecture. Each team will be able to make more accurate estimates for its piece than a project manager will and will feel more ownership in making the estimates come true. Second, the initial definition of an architecture means that the requirements for a system have been reviewed and, in some sense, validated. The more knowledge about the scope of a system, the more accurate the estimates.

**ARCHITECTURE AS A TRANSFERABLE, RE-USABLE MODEL**
The earlier in the life cycle re-use is applied, the greater the benefit that can be achieved. While code re-use is beneficial, re-use at the architectural level provides tremendous leverage for systems with similar requirements. Not only code can be re-used but so can the requirements that led to the architecture in the first place, as well as the experience of building the re-used architecture. When architectural decisions can be re-used across multiple systems, all of the early decision consequences we just described are also transferred.

**Software Product Lines Share a Common Architecture.** A software product line or family is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way. Chief among these core 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 by making design decisions that apply across the family early and by making other decisions that apply only to individual members late. 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 shows 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 14, and case studies of product lines are given in Chapters 15 and 17.

**Systems Can Be Built Using Large, Externally Developed Elements.** Whereas earlier software paradigms 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. It 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.

One key aspect of architecture is its organization of element structure, interfaces, and operating concepts. The most significant principle of this organization is *interchangeability*. 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 reliable physical measurements, this 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, subsystems, and compatible communications interfaces all depend on the principle of interchangeability. However, there is much about software development through composition that remains unresolved. When the components that are candidates for importation and re-use 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.

**Less Is More: It Pays to Restrict the Vocabulary of Design Alternatives.** As useful architectural patterns and design patterns are collected, it becomes clear that, although computer programs 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 when it comes to program cooperation and interaction. That is, we wish to minimize the design complexity of the system we are building. Advantages to this approach include enhanced re-use, more regular and simpler designs that are more easily understood and communicated, more capable analysis, shorter selection time, and greater interoperability.

Properties of software design follow from the choice of architectural pattern. 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, and/or by helping to surface inconsistencies in requirements specifications.
System Architecture versus Software Architecture

Over the past 5 to 10 years, we have had many occasions to give talks on software architecture. Invariably, a question comes from the audience along the lines of "Why are you talking about software architecture? Isn't system architecture just as important?" or "What is the difference between software architecture and system architecture?"

In fact, there is little difference, as we will see. But we mostly talk about software architecture because we want to stress the crucial nature of the software decisions that an architect makes concerning overall product quality.

In creating a software architecture, system considerations are seldom absent. For example, if you want an architecture to be high performance, you need to have some idea of the physical characteristics of the hardware platforms that it will run on (CPU speed, amount of memory, disk access speed) and the characteristics of any devices that the system interfaces with (traditional I/O devices, sensors, actuators), and you will also typically be concerned with the characteristics of the network (primarily bandwidth). If you want an architecture that is highly reliable, again you will be concerned with the hardware, in this case with its failure rates and the availability of redundant processing or network devices. On it goes. Considerations of hardware are seldom far from the mind of the architect.

So, when you design a software architecture, you will probably need to think about the entire system—the hardware as well as the software. To do otherwise would be foolhardy. No engineer can be expected to make predictions about the characteristics of a system when only part of that system is specified.

But still we persist in speaking about software architecture primarily, and not system architecture. Why is this? Because most of the architect's freedom is in the software choices, not in the hardware choices. It is not that there are no hardware choices to be made, but these may be out of the architect's control (for example, when creating a system that needs to work on arbitrary client machines on the Internet) or specified by others (for reasons of economics, legal issues, or compliance with standards); or they will likely change over time.

For this reason, we feel justified in focusing on the software portion of architecture, for this is where the most fundamental decisions are made, where the greatest freedoms reside, and where there are the greatest opportunities for success (or disaster!).

— RK

An Architecture Permits Template-Based Development. An architecture embodies design decisions about how elements interact that, while reflected in each element's implementation, can be localized and written just once. Templates can be used to capture in one place the inter-element interaction mechanisms. For instance, a template can encode the declarations for an element's public area where results will be left, or can encode the protocols that the element uses to
engage with the system executive. An example of a set of firm architectural decisions enabling template-based development will be discussed in Chapter 8.

**An Architecture Can Be the Basis for Training.** The architecture, including a description of how elements interact to carry out the required behavior, can serve as the 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.

### 2.5 Architectural Structures and Views

The neurologist, the orthopedist, the hematologist, and the dermatologist all have a different view of the structure of a human body. Ophthalmologists, cardiologists, and podiatrists concentrate on subsystems. 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. Together they describe the architecture of the human body.

So it is with software. Modern systems are more than complex enough to make it difficult to grasp them 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.

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. 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, as documented by and used by some system stakeholders. These terms are often used interchangeably, but we will adhere to these definitions.

Architectural structures can by and large be divided into three groups, depending on the broad nature of the elements they show.

- *Module structures.* Here the elements are modules, which are units of implementation. Modules represent a code-based way of considering the system. They are assigned areas of functional responsibility. There is less emphasis on how the resulting software manifests itself at runtime. Module structures allow us to answer questions such as 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?
What modules are related to other modules by generalization or specialization (i.e., inheritance) relationships?

- **Component-and-connector structures.** Here the elements are runtime components (which are the principal units of computation) and connectors (which are the communication vehicles among components). Component-and-connector structures help answer questions such as What are the major executing components and how do they interact? 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? How can the system’s structure change as it executes?

- **Allocation structures.** Allocation structures show the relationship between the software elements and the elements in one or more external environments in which the software is created and executed. They answer questions such as What processor does each software element execute on? In what files is each element stored during development, testing, and system building? What is the assignment of software elements to development teams?

These three structures correspond to the three broad types of decision that architectural design involves:

- How is the system to be structured as a set of code units (modules)?
- How is the system to be structured as a set of elements that have runtime behavior (components) and interactions (connectors)?
- How is the system to relate to nonsoftware structures in its environment (i.e., CPUs, file systems, networks, development teams, etc.)?

**SOFTWARE STRUCTURES**

Some of the most common and useful software structures are shown in Figure 2.3. These are described in the following sections.

**Module.** Module-based structures include the following.

- **Decomposition.** The units are modules related to each other by the "is a submodule of" relation, showing how larger modules are decomposed into smaller ones recursively until they 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 associated products (i.e., interface specifications, code, test plans, etc.). The decomposition structure provides a large part of the system’s modifiability, by ensuring that likely changes fall within the purview of at most a few small modules. It is often used as the basis for the development project’s organization, including the
structure of the documentation, and its integration and test plans. The units in this structure often have organization-specific names. Certain U.S. Department of Defense standards, for instance, define Computer Software Configuration Items (CSCIs) and Computer Software Components (CSCs), which are units of modular decomposition. In Chapter 15, we will see system function groups and system functions as the units of decomposition.

- \textit{Uses}. The units of this important but overlooked structure are also modules, or (in circumstances where a finer grain is warranted) procedures or resources on the interfaces of modules. The units are related by the \textit{uses} relation. One unit uses another if the correctness of the first requires the presence of a correct version (as opposed to a stub) of the second. The uses structure is used to engineer systems that can be easily extended to add functionality or from which useful functional subsets can be easily extracted. The ability to easily subset a working system allows for incremental development, a powerful build discipline that will be discussed further in Chapter 7.

- \textit{Layered}. When the uses relations in this structure are carefully controlled in a particular way, a system of layers emerges, in which a layer is a coherent set of related functionality. In a strictly layered structure, layer $n$ may only use the services of layer $n - 1$. Many variations of this (and a lessening of this structural restriction) occur in practice, however. Layers are often designed as abstractions (virtual machines) that hide implementation specifics below from the layers above, engendering portability. We will see layers in the case studies of Chapters 3, 13, and 15.
- **Class**, or *generalization*. 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 (i.e., the classes that other classes inherit from) and parameterized differences which are captured by subclassing. The class structure allows us to reason about re-use and the incremental addition of functionality.

**Component-and-Connector.** These structures include the following.

- **Process**, or *communicating processes*. Like all component-and-connector structures, this one is orthogonal to the module-based structures and deals with the dynamic aspects of a running system. The units here are processes or threads that are connected with each other by communication, synchronization, and/or exclusion operations. The relation in this (and in all component-and-connector structures) is *attachment*, showing how the components and connectors are hooked together. The process structure is important in helping to engineer a system’s execution performance and availability.

- **Concurrency.** 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 “logical threads.” A logical thread is a sequence of computation that can be allocated to a separate physical thread later in the design process. The concurrency structure is used early in design to identify the requirements for managing the issues associated with concurrent execution.

- **Shared data**, or *repository*. This structure comprises components and connectors that create, store, and access persistent data. If the system is in fact structured around one or more shared data repositories, this structure is a good one to illuminate. It shows how data is produced and consumed by runtime software elements, and it can be used to ensure good performance and data integrity.

- **Client-server.** If the system is built as a group of cooperating clients and servers, this is a good component-and-connector structure to illuminate. The components are the clients and servers, and the connectors are protocols and messages they share to carry out the system’s work. This is useful for separation of concerns (supporting modifiability), for physical distribution, and for load balancing (supporting runtime performance).

**Allocation.** Allocation structures include the following.

- **Deployment.** The deployment structure shows how software is assigned to hardware-processing and communication elements. The elements are software (usually a process from a component-and-connector 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 view allows an engineer to reason about performance, data integrity, availability, and security. It is of particular interest in distributed or parallel systems.

- **Implementation.** 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.

- **Work assignment.** This structure assigns responsibility for implementing and integrating the modules to the appropriate development teams. Having a work assignment structure as 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 them to a single team, rather than having them implemented by everyone who needs them.

Table 2.1 summarizes the software structures. The table lists the meaning of the elements and relations in each structure and tells what each structure might be used for.

### TABLE 2.1 Architectural Structures of a System

<table>
<thead>
<tr>
<th>Software Structure</th>
<th>Relations</th>
<th>Useful for</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decomposition</td>
<td>Is a submodule of; shares secret with</td>
<td>Resource allocation and project structuring and planning; information hiding, encapsulation; configuration control</td>
</tr>
<tr>
<td>Uses</td>
<td>Requires the correct presence of</td>
<td>Engineering subsets; engineering extensions</td>
</tr>
<tr>
<td>Layered</td>
<td>Requires the correct presence of; uses the services of; provides abstraction to</td>
<td>Incremental development; implementing systems on top of “virtual machines” portability</td>
</tr>
<tr>
<td>Class</td>
<td>Is an instance of; shares access methods of</td>
<td>In object-oriented design systems, producing rapid almost-alike implementations from a common template</td>
</tr>
<tr>
<td>Client-Server</td>
<td>Communicates with; depends on</td>
<td>Distributed operation; separation of concerns; performance analysis; load balancing</td>
</tr>
<tr>
<td>Process</td>
<td>Runs concurrently with; may run concurrently with; excludes; precedes; etc.</td>
<td>Scheduling analysis; performance analysis</td>
</tr>
</tbody>
</table>

*continued*
TABLE 2.1 Architectural Structures of a System *Continued*

<table>
<thead>
<tr>
<th>Software Structure</th>
<th>Relations</th>
<th>Useful for</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concurrency</td>
<td>Runs on the same logical thread</td>
<td>Identifying locations where resource contention exists, where threads may fork, join, be created or be killed</td>
</tr>
<tr>
<td>Shared Data</td>
<td>Produces data; consumes data</td>
<td>Performance; data integrity; modifiability</td>
</tr>
<tr>
<td>Deployment</td>
<td>Allocated to; migrates to</td>
<td>Performance, availability, security analysis</td>
</tr>
<tr>
<td>Implementation</td>
<td>Stored in</td>
<td>Configuration control, integration, test activities</td>
</tr>
<tr>
<td>Work Assignment</td>
<td>Assigned to</td>
<td>Project management, best use of expertise, management of commonality</td>
</tr>
</tbody>
</table>

Although we often think about a system’s structure in terms of its functionality, there are system properties in addition to functionality, such as physical distribution, process communication, and synchronization, that must be considered at an architectural level. Each structure provides a method for reasoning about some of the relevant quality attributes. The uses structure, for instance, must be *engineered* (not merely recorded) to build a system that can be easily extended or contracted. The process structure is *engineered* to eliminate deadlock and reduce bottlenecks. The module decomposition structure is *engineered* to produce modifiable systems, and so forth. Each structure provides the architect with a different view into the system and a different leverage point for design.

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

Individual projects sometimes consider one structure dominant and cast other structures, when possible, in terms of it. Often, but not always, the dominant structure is module decomposition. This is for a good reason: It tends to spawn the project structure. 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 client-server structure of a system would need to consider the process and deployment views
because client-server mechanisms typically involve processes and threads, and physical distribution might involve different control mechanisms than would be used if the processes were colocated on a single machine. If control mechanisms need to be changed, the module decomposition or layered view would need to be considered to determine the extent of the changes.

Not all systems warrant consideration of many architectural structures. The larger the system, the more dramatic the differences between these structures tend to be; however, for small systems we can often get by with less. Instead of working with each of several component-and-connector structures, a single one will do. If there is only one process, then the process structure collapses to a single node and need not be carried through the design. If there is to be no distribution (that is, if there is just one processor), then the deployment structure is trivial and need not be considered further.

Structures represent the primary engineering leverage points of an architecture. Individual structures bring with them the power to manipulate one or more quality attributes. They represent a powerful separation-of-concerns approach for creating the architecture (and, later, for analyzing it and explaining it to stakeholders). And, as we will see in Chapter 9, the structures that the architect has chosen as engineering leverage points are also the primary candidates for the basis for architecture documentation.

WHICH STRUCTURES TO CHOOSE?

We have briefly described a number of useful architectural structures, and there are many more. Which ones should an architect work on? Which ones should the architect document? Surely not all of them.

There is no shortage of advice. In 1995, Philippe Kruchten [Kruchten 95] published a very influential paper in which he described the concept of architecture comprising separate structures and advised concentrating on four. To validate that the structures were not in conflict with each other and together did in fact describe a system meeting its requirements, Kruchten advised using key use cases as a check. This so-called “Four Plus One” approach became popular and has now been institutionalized as the conceptual basis of the Rational Unified Process. Kruchten’s four views follow:

- **Logical.** The elements are “key abstractions,” which are manifested in the object-oriented world as objects or object classes. This is a module view.
- **Process.** This view addresses concurrency and distribution of functionality. It is a component-and-connector view.
- **Development.** This view shows the organization of software modules, libraries, subsystems, and units of development. It is an allocation view, mapping software to the development environment.
- **Physical.** This view maps other elements onto processing and communication nodes and is also an allocation view (which others call the deployment view).
At essentially the same time that Kruchten published his work, Soni, Nord, and Hofmeister [Soni 95] published an influential paper in which they reported the structures put into use across many projects by the software architects in their organization. Their views were conceptual, module interconnection, execution, and code. Once again, these map clearly to the module, component-and-connector, and allocation models.

Other authors followed, and the list of available structures grows ever more rich. Of course, you should not use them all even though most of them will in fact exist in the system you are building. Instead, consider that one of the obligations of the architect is to understand how the various structures lead to quality attributes, and then choose the ones that will best deliver those attributes. This point will be treated at greater length in Chapter 9, on architectural representation.

---

2.6 Summary

This chapter defined software architecture and also introduced the related concepts of reference model, reference architecture, and architectural pattern. We have explained why architecture is a fundamentally useful concept in software engineering, in terms of the early insights it provides into the system, the communication it enables among stakeholders, and the value it provides as a re-usable asset. All of these themes will be expanded in subsequent chapters.

Our definition of architecture makes clear that systems comprise many structures. We showed several of the most commonly used structures and explained how each serves as an engineering leverage point into the design process.

The next chapter is the first case study of the book. Its purpose is to show the utility of different architectural structures in the design of a complex system.

---

2.7 For Further Reading

The early work of David Parnas laid much of the conceptual foundation for what became the study of software architecture (see the sidebar Architecture Déjà Vu). 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].

Software architectural patterns have been extensively catalogued in Pattern-Oriented Software Architecture [Buschmann 96, Schmidt 00].
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 views as used 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 00].

A discussion of architectural mismatch can be found in Garlan et al. [Garlan 95]. Barry Boehm [Boehm 95] discusses the process issues surrounding software architecture.

The Software Engineering Institute's software architecture Web page [SEI ATA] provides a wide variety of software architecture resources and links, including a broad collection of definitions of the term.

Paulish [Paulish 02] discusses the relationship of cost and schedule to the existence of an architecture.

Architecture Déjà Vu

While architecture is undoubtedly a vital part of system development that is enjoying widespread attention at the moment, it must be pointed out that the field is plowing old ground in several areas. In many ways we are "discovering" fundamental principles that were laid out eloquently and convincingly over a quarter-century ago by Fred Brooks, Edsger Dijkstra, David Parnas, and others.

In programming, the term architecture was first used to mean a description of a computer system that applied equally to more than one system. It still carries this meaning today. In 1969, Fred Brooks and Ken Iverson called architecture the "conceptual structure of a computer . . . as seen by the programmer" [Brooks 69]. A few years later, Brooks (crediting G. Blaauw for the term) defined architecture as "the complete and detailed specification of the user interface" [Brooks 75]. A careful distinction was drawn between architecture and implementation. Quoting Blaauw, Brooks writes, "Where architecture tells what happens, implementation tells how it is made to happen." This distinction survives today, and in the era of object-oriented programming, it thrives.

The term architecture is still used today in some communities to refer to the user view of a system, but that is not what we mean by software architecture. The structure(s) contained in a software architecture is invisible to the system's end user. However, the conceptual separation between the what and the how applies. Software architecture is not concerned with how elements do what they do, just as the end user is not concerned with how the system does what it does. The notion of architecture as a common description of a class of systems (i.e., an abstraction, where all the instances are said to exhibit the architecture) remains at the heart of what we call software architecture today.

Also in 1968, Edsger Dijkstra was telling us to be concerned with how software is partitioned and structured as opposed to simply programming to produce a correct result [Dijkstra 68]. He was writing about an operating system
and introduced the idea of a layered structure, in which programs were grouped into layers and programs in one layer could communicate only with programs in adjacent layers. Dijkstra pointed out the elegant conceptual integrity exhibited by such an organization, resulting in increased ease of development and maintenance.

David Parnas advanced this line of observation with his fundamental contributions to software engineering in the early 1970s. In his work, more than anyone else’s, is to be found many of the fundamental tenets and principles behind software architecture, including the following:

- A design principle for how to break a system into elements to increase maintainability and (as we will see in Chapter 5) re-usability. If architecture has a fundamental principle, it is this one, which Parnas called information hiding [Parnas 72].
- The principle of using an element via its interface only, the conceptual basis of all object-based design [Parnas 72].
- An observation of the various structures to be found in software systems, with an admonition not to confuse them—a lesson often forgotten by today’s “architecturists” [Parnas 74].
- Introduction of the uses structure, a principle for controlling the connections between elements in order to increase the extensibility of a system, as well as the ability to field subsets quickly and easily [Parnas 79].
- The principle of detection and handling of errors (now called exceptions) in component-based systems, which is the underlying approach of most modern programming languages [Parnas 72, 76].
- Viewing every program as a member of a family of programs, with principles for taking advantage of the commonalities among the members and ordering the design decisions so that the ones that need to be the easiest to revise are made last. The coarse structuring of the program—part of its architecture—comprises the set of early, family-wide design decisions [Parnas 76].
- Recognition that the structure of a system influences the qualities (such as reliability) of that system [Parnas 76].

Now it is true, and Parnas would agree, that not all of the ideas in his papers were invented by him from whole cloth. About information hiding, for example, he has said that he was writing down what good programmers had been doing for a long time (especially operating systems programmers writing device drivers). However, taken as a body, Parnas’s work is a coherent statement of the theme of software architecture: Structure matters. His insights form the backbone of software architecture as a study area, and no book on the subject would be complete without acknowledging his fundamental contributions.

Recently a colleague and I had a fine architectural discussion about what exactly constitutes the interface to a software element; clearly it is much more than the names of the programs you can call and the parameters they take. My colleague worked out that it is actually the set of assumptions that you
can safely make about the element, and that these assumptions vary according to the context of the element's use. I agreed and pulled out Parnas's paper [Parnas 71] in which he said precisely the same thing. My friend looked a little crestfallen for a moment and then said, "Now I know how Scott felt when he reached the South Pole and found Amundsen's flag already planted. He probably said, 'Oh, damn. And now I've got to eat my dogs.'"

Parnas's flag is planted deeply, and often, in our field. In the next chapter, we will present a case study of an architecture created by Parnas to put his ideas into practical use in a demanding real-world application. Even though it ran its course long ago, we know of no other single project that so clearly laid out and faithfully followed architectural principles such as engineering and maintaining separate structures to achieve quality goals; strict information hiding to achieve re-usable elements and a re-usable architecture; and pains-taking specification of that architecture, its elements, and their relationships.

While Parnas and others laid the foundations, the field has taken its own turns in the interim. Experience with basic ideas leads to the refinement of those ideas, to embellishments rooted in practicalities, and to entirely new concepts. Thus, while Parnas wrote about program families a couple of decades ago, we will see in Chapter 14 that organizational, process, and managerial concerns predominate in the successful development of product lines, their conceptual descendant. While Dijkstra wrote about separation of concerns about a quarter-century ago, objects (the conceptual descendant) have only fairly recently come into their own as a standard, widely accepted design approach. And while Brooks and Blaauw wrote about architecture even longer ago, we've already seen that architectures cannot be understood except in light of the business issues that spawned them, and we will see ways to analyze architectures without waiting for the system to be built.

Today, architecture as a field of study is large and growing larger, primarily because it has left the realm of deep thinkers and visionaries and made the transition into practice. The early ideas have been refined and applied enough so that it is becoming an accepted state-of-the-practice approach to system building.

---PCC

2.8 Discussion Questions

1. Software architecture is often compared to building architecture. What are the strong points of this comparison? What is the correspondence in buildings to software architecture structures and views? To patterns? What are the weaknesses of the comparison? When does it break down?

2. What is the difference between a reference architecture and an architectural pattern? What can you do with one that you cannot do with the other in terms of organizational planning and architectural analysis?
3. Do the architectures in your organization recognize the different views (structures and relations) inherent in architecture? If so, which ones? If not, why not?

4. Is there a different definition of software architecture that you are familiar with? If so, think about the ways in which this definition supports our acid test of an architecture: Does it abstract information away from the system and yet provide enough information to be a basis for analysis, decision making, and risk reduction?