# 2 Software Architecture Concepts

One of the problems encountered when we talk about architecture for software systems is that the terminology has been loosely borrowed from other disciplines (such as building architecture or naval architecture) and is widely used, inconsistently, in a variety of situations. For example, the term *architecture* is used to refer to the internal structure of microprocessors, the internal structure of machines, the organization of networks, the structure of software programs, and many other things.

This chapter defines and reviews some of the core concepts that underpin the discussion in the remainder of the book: *software architecture*, *architectural elements*, *stakeholders*, and *architectural descriptions*.

## Software Architecture

Computers can be found everywhere in modern society—not just in data centers or on desks but also in cars, washing machines, and credit cards. Whether they are big or small, simple or complex, all computer systems are made up of the same three fundamental parts: hardware (e.g., processors, memory, disks, network cards); software (e.g., programs or libraries); and data, which may be either transient (in memory) or persistent (on disk or ROM).

When you try to understand a computer system, you are interested in what its individual parts actually do, how they work together, and how they interact with the world around them—in other words, its *architecture*. The best definition of architecture that we have read, and the one widely accepted in the architectural community, comes from work done in the Software Architecture group of the Software Engineering Institute (SEI) at Carnegie-Mellon University in Pittsburgh.

**DEFINITION** The **architecture** of a software-intensive 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.

Let's look at two key parts of this definition in a bit more detail, namely, a system's *structures* and its *externally visible properties*.

## System Structures

There are two types of system structure of interest to the software architect: *static* (design-time organization) and *dynamic* (runtime organization).

1. The *static structures* of a system tell you what the design-time form of a system is—that is, what its elements are and how they fit together.

**DEFINITION** The **static structures** of a software system define its internal design-time elements and their arrangement.

Internal design-time software elements might be modules, object-oriented classes or packages, database stored procedures, services, or any other self-contained code unit. Internal data elements include classes, relational database entities/tables, and data files. Internal hardware elements include computers or their constituent parts such as disk or CPU and networking elements such as cables, routers, or hubs.

The static arrangement of these elements defines—depending on the context—the associations, relationships, or connectivity between these elements. For software modules, for example, there may be static relationships such as a hierarchy of elements (module *A* is built from modules *B* and *C*) or dependencies between elements (module *A* relies on the services of module *B*). For classes, relational entities, or other data elements, relationships define how one data item is linked to another one. For hardware, the relationships define the required physical interconnections between the various hardware elements of the system.

2. The system's *dynamic structures* show how the system actually works—that is, what happens at runtime and what the system does in response to external (or internal) stimulus.

**DEFINITION** The **dynamic structures** of a software system define its runtime elements and their interactions.

These internal interactions may be flows of information between elements (element *A* sends messages to element *B*) or the parallel or sequential execution of internal tasks (element *X* invokes a routine on element *Y*), or they may be expressed in terms of the effect they have on data (data item *D* is created, updated many times, and finally destroyed).

## Externally Visible System Properties

External properties manifest themselves in two different ways: *externally visible behavior* (what the system does) and *quality properties* (how the system does it).

1. *Externally visible behavior* tells you what a system does from the viewpoint of an external observer.

**DEFINITION** The **externally visible behavior** of a software system defines the functional interactions between the system and its environment.

These external interactions form a set similar to the ones we considered for dynamic structure. This includes flows of information in and out of the system, the way that the system responds to external stimuli, and the published "contract" or API that the architecture has with the outside world.

External behavior may be modeled by treating the system as a black box so that you don't know anything about its internals (if you make request *P* to a system built in compliance with the architecture, you are returned response *Q*). Alternatively, it may consider changes to internal system state in response to external stimuli (submitting a request *R* causes the creation of an internal data item *D*).

2. *Quality properties* tell you how a system behaves from the viewpoint of an external observer (often referred to as its nonfunctional characteristics).

**DEFINITION** A **quality property** is an externally visible, nonfunctional property of a system such as performance, security, or scalability.

There is a whole range of external architectural characteristics that may be of interest: How does the system perform under load? What is the peak throughput given certain hardware? How is the information in the system protected from malicious use? How often is it likely to break? How easy is it to manage, maintain, and enhance? How easily can it be used by people who are disabled? Which of these characteristics are relevant depends on your circumstances and on the concerns and priorities of your stakeholders.

## External Properties and Internal Organization

Let's explore these concepts in more detail by means of a simple example.

**EXAMPLE** An airline reservation system supports a number of different transactions to book airline seats, update or cancel them, transfer them, upgrade them, and so forth. Figure 2–1 shows the context for this system.

The *externally visible behavior* of the system (what it does) is its response to the transactions that can be submitted by customers, such as booking a seat, updating a reservation, or canceling a booking. The *quality properties* of the system (how it does it) include the average response time for a transaction under a specified load, the maximum throughput the system can support, system availability, and the time required to repair defects.

Faced with these requirements, there are a number of ways that an architect could design a system for it. Over the next few pages we outline two possible architectural approaches for this system.

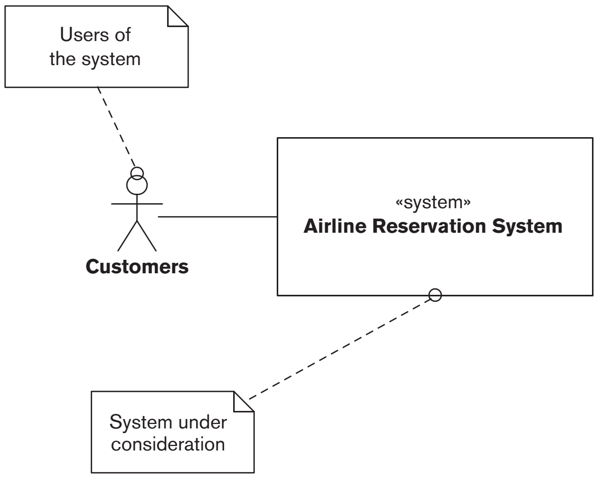

**FIGURE 2–1** CONTEXT DIAGRAM FOR AN AIRLINE BOOKING SYSTEM

The architect could design a solution for the airline reservation system based around a **client/server** or **two-tier** approach. (In fact, client/server is an example of an *architectural style*, as we will see in Part II.) In this approach, shown in Figure 2–2, a number of clients (which present information to customers and accept their input) communicate with a central server (which stores the data in a relational database) via a wide-area network (WAN).

The *static structure* (design-time organization) for this client/server architecture consists of the client programs (which in this example are further broken down into presentation, business logic, database, and network layers), the server, and the connections between them. The *dynamic structure* (runtime organization) is based on a request/response model: Requests are submitted by a client to the server over the WAN, and responses are returned by the server to the client.

Alternatively, the architect could take a **three-tier** or **thin-client** approach, where only the presentation processing is performed on the clients, with the business logic and database access performed in an application server, as shown in Figure 2–3.

The *static structure* for this architecture consists of the client programs (which in this example are further broken down into presentation and network layers), the application server (here, business logic, database, and network layers), the database server, and the connections between

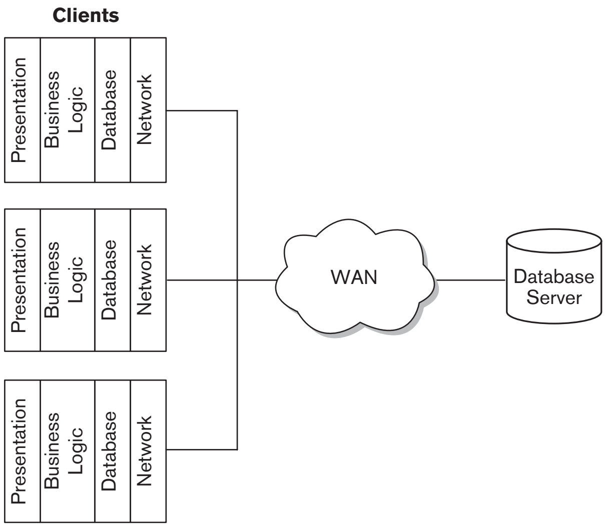

**FIGURE 2–2** Client/Server Architecture for an Airline Booking System

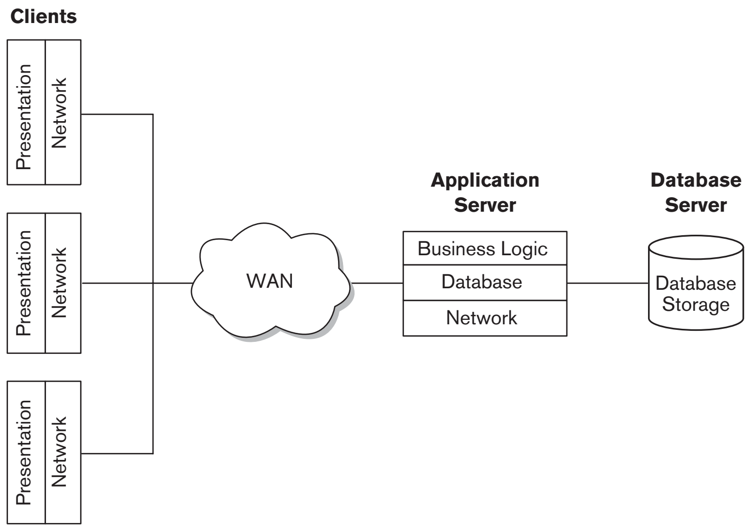

**FIGURE 2–3** THIN-CLIENT ARCHITECTURE FOR AN AIRLINE BOOKING SYSTEM

them. The *dynamic structure* is based on a three-tier request/response model: Requests are submitted by a client to the application server over the WAN, the application server submits requests to the database server if necessary, and responses are returned by the application server to the client.

The architect might identify the two-tier approach as appropriate for the architecture because of its relative operational simplicity, because it can be developed quickly by the organization's software developers, because it can be delivered at lower cost than other options, or for a range of other reasons.

Alternatively, the architect may consider the three-tier approach to be right for the architecture because it provides better options for scalability as workload increases, because less powerful client hardware is needed, because it may offer better security, or for other reasons.

Whichever approach the architect considers to be most appropriate, she chooses it because it provides the best match between the system properties promised by the approach and the requirements of the system.

In this example, there are two possible solutions to the problem, based around a client/server approach and a three-tier approach, respectively. We call these *candidate architectures*.

**Definition** A **candidate architecture** for a system is a particular arrangement of static and dynamic structures that has the potential to exhibit the system's required externally visible behaviors and quality properties.

Although the candidate architectures have different static and dynamic structures, each must be able to meet the system's overall requirements to process airline bookings in a timely and efficient manner. However, although all candidate architectures are believed to share the same important externally visible behaviors (in this case, responses to booking transactions) and general quality properties (such as acceptable response time, throughput, availability, and time to repair), they are likely to differ in the specific set of quality properties that each exhibits (such as one being easier to maintain but more expensive to build than another).

In each case, the extent to which the candidate actually exhibits these behaviors and properties must be determined by further analysis of its static and dynamic structures. For example, the client/server model might meet the functional requirements better because it supports functionally richer clients; the three-tier model might deliver better throughput and response time because it is more loosely coupled.

It is part of the architect's role to derive the static and dynamic structures for each of the candidate architectures, understand the extent to which they exhibit the required behaviors and quality properties, and select the best one. Of course, what is meant by "best" may not always be clear; we will return to this issue in Part II.

We can capture the relationship between the externally visible properties of a system and its internal structure and organization as follows.

- The externally visible behavior of a system (what it does) is determined by the combined functional behavior of its internal elements.
- The quality properties of a system such as performance, scalability, and resilience (how it does it) arise from the quality properties of its internal elements. (Typically, a system's overall quality property is only as good as the property of its worst-behaving or weakest internal element.)

## The Importance of Software Architecture

Every computer system, large or small, is made up of pieces that are linked together. There may be a small number of these pieces, or perhaps only one, or there may be dozens or hundreds; and this linkage may be trivial, or very complicated, or somewhere in between.

Furthermore, every system is made up of pieces that interact with each other and the outside world in a deterministic (predictable) way. Again, the behavior may be simple and easily understood, or it may be so convoluted that no one person can understand every aspect of it. However, this behavior is still there and still (in theory at least) describable.

In other words, every system has an architecture, in the same way that every building, bridge, and battleship has an architecture—and every human body has a physiology.

This is such an important concept that we will state it formally as a principle here.

---

**PRINCIPLE** Every computer system has an architecture, whether or not it is documented and understood.

---

The architecture of a system is an intrinsic, fundamental property that is present whether or not it has been documented and is understood. It follows that every system has precisely one architecture—although, as we will see, it can be represented in a number of ways.

## ARCHITECTURAL ELEMENTS

Throughout this book, we standardize the term *architectural element* to refer to the pieces from which systems are built.

---

**DEFINITION** An **architectural element** (or just element) is a fundamental piece from which a system can be considered to be constructed.

---

The nature of an architectural element depends very much on the type of system you are considering and the context within which you are considering its elements. Programming libraries, subsystems, deployable software units (e.g., Enterprise Java Beans and Active X controls), reusable software products (e.g., database management systems), or entire applications may form architectural elements in an information system, depending on the system being built.

An architectural element should possess the following key attributes:

- A clearly defined set of *responsibilities*
- A clearly defined *boundary*
- A set of clearly defined *interfaces*, which define the *services* that the element provides to the other architectural elements

Architectural elements are often known informally as *components* or *modules*, but these terms are already widely used with established specific meaning. In particular, the term *component* tends to suggest the use of a programming-level component model (such as J2EE or .NET), while *module* tends to suggest a programming language construct. Although these are valid architectural elements in some contexts, they won't be the type of fundamental system element used in others.

For this reason, we deliberately don't use these terms from now on. Instead, we use the term *element* throughout the book to avoid confusion (following the lead of others, including Perry and Wolf [PERR92] and Bass, Clements, and Kazman [BASS03]—see the Further Reading section at the end of this chapter for more details).

## Stakeholders

Traditional software development has been driven by the need of the delivered software to meet the requirements of users. Although the definition of the term *user* varies, all software development methods are based around this principle in one way or another.

However, the people affected by a software system are not limited to those who use it. Software systems are not just used: They have to be *built* and *tested*, they have to be *operated*, they may have to be *repaired*, they are usually *enhanced*, and of course they have to be *paid for*. Each of these activities involves a number—possibly a significant number—of people in addition to the users. Each of these groups of people has its own requirements, interests, and needs to be met by the software system.

We refer collectively to these people as *stakeholders*. Understanding the role of the stakeholder is fundamental to understanding the role of the architect in the development of a software product or system. We define a stakeholder as follows.

**Definition** A **stakeholder** in a software architecture is a person, group, or entity with an interest in or concerns about the realization of the architecture.

The definition comes from IEEE Standard 1471 [IEEE00] on architectural description, which we discuss in more depth in Part II. For now, let's look at a couple of key concepts from this definition.

### People, Groups, and Entities

First of all, consider the phrase "person, group or entity." As we shall see in this book, those with an interest in an architecture stretch far more widely than just its developers, or even its developers and users. A much broader community than this is affected by the realization of the architecture, such as those who have to support it, deploy it, or pay for it.

Specifying the architecture is a key opportunity for the stakeholders to direct its shape and direction. You will find, however, that some stakeholders are more interested in their roles than others, for a variety of reasons that have little to do with architecture. Part of your role, therefore, is to engage and galvanize, to persuade people of the importance of their involvement, and to obtain their commitment to the task.

A stakeholder often represents a class of person, such as user or developer, rather than an individual. This presents some problems because it may not be possible to capture and reconcile the needs of all members of the class (all users, all developers) in the time available. Furthermore, you may not have the stakeholders at hand (e.g., when developing a new product). In either case, you need to select some representative stakeholders who will speak for the group. We'll come back to this in Part II.

## Interests and Concerns

Now consider the phrase "interest in or concerns about." This criterion is—deliberately—a broad one, and its interpretation is entirely specific to individual projects.

We use the term *concern* here, incidentally, not just because the IEEE standard uses it but also because it is particularly appropriate to the process of architecture. As you will see when you start to develop your architecture, you are engaged in a process of discovery as much as one of capture—in other words, this early in the system development lifecycle, your stakeholders may not yet know precisely what their requirements are.

**DEFINITION** A **concern** about an architecture is a requirement, an objective, an intention, or an aspiration a stakeholder has for that architecture.

Many concerns will be common among stakeholders, but some concerns will be distinct and may even conflict. Resolving such conflicts in a way that leaves stakeholders satisfied can be a significant challenge.

**EXAMPLE** Some of the important attributes of a software development project are often shown as a triangle whose corners represent cost, quality, and time to market. Ideally we would like a project to have high quality, zero cost, and immediate delivery, but we know this isn't possible. The *quality triangle* in Figure 2–4 shows that it is necessary to make

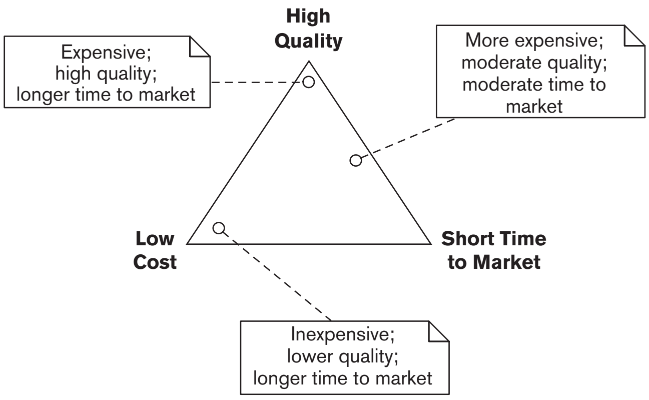
High Quality
Expensive; high quality; longer time to market
More expensive; moderate quality; moderate time to market
Low Cost
Short Time to Market
Inexpensive; lower quality; longer time to market

**FIGURE 2–4** The Quality Triangle

compromises between these three attributes, and the best you are likely to achieve is two out of three.

For example, a high-quality system tends to take longer to build and to cost more. Conversely, it is often possible to reduce the initial development time but, assuming costs are kept roughly constant, this comes at the expense of reducing the quality of the delivered software.

One or more of these attributes is likely to be important to different stakeholders, and it is the architect's job to understand which of these attributes is important to whom and to reach an acceptable compromise when necessary. We'll talk more about how to do this in Part II.

## The Importance of Stakeholders

Stakeholders (explicitly or implicitly) drive the whole shape and direction of the architecture, which is developed solely for their benefit and to serve their needs. Stakeholders ultimately make or direct the fundamental decisions about scope, functionality, operational characteristics, and structure of the eventual product or system—under the guidance of the architect, of course. Without stakeholders, there would be no point in developing the architecture because there would be no need for the system it will turn into, nor would there be anyone to build it, deploy it, run it, or pay for it.

**Principle** Architectures are created solely to meet stakeholder needs.

It follows that if a system does not meet the needs of its stakeholders, it cannot be considered a success—no matter how well it conforms to good architectural practice. In other words, architectures must be evaluated with respect to stakeholder needs as well as abstract architectural and software engineering principles.

**PRINCIPLE** A good architecture is one that successfully meets the objectives, goals, and needs of its stakeholders.

Part II explores the concept of stakeholders in more detail and explains how they can be classified, identified, selected, and engaged in the development of the architecture.

## ARCHITECTURAL DESCRIPTIONS

An architecture for a software system can be an incredibly complex thing. Part of the architect's role is to describe this complexity to the people who need to understand it. The architect does this by means of an *architectural description*.

**DEFINITION** An **architectural description (AD)** is a set of products that documents an architecture in a way its stakeholders can understand and demonstrates that the architecture has met their concerns.

"Products" in this context consists of a range of things—particularly *architectural models*, but also *scope definition*, *constraints*, and *principles*. We discuss each of these in more detail in Part II.

A description of an architecture has to present the essence of the architecture and its detail at the same time—in other words, it must provide an overall picture that summarizes the whole system, but it also must decompose into enough detail that it can be validated and the described system can be built.

Although it is true that every system has an architecture, it is unfortunately not true that every system has an AD. Even if an architecture is documented, it may be documented only in part, or the documentation may be out of date or unused.

Strictly speaking, therefore, our definition describes a *good* AD. However, an AD that its stakeholders cannot understand or that doesn't demonstrate to them that their concerns have been met is really not worth having—in fact, it can be more of a liability than an asset. The AD needs to contain all of (and ideally only) the information needed to communicate the architecture effectively to those stakeholders who need to understand it.

**Principle** Although every system has an architecture, not every system has an architecture that is effectively communicated via an architectural description.

Of course, the chances of project success are far less if the AD is inadequate.

**Example** The AD for the airline reservation system referred to earlier focused strongly on the static structure (the key hardware and software elements) and to a lesser extent on its external behavior (the requests that users could make). Because most users would have a customer at a sales desk or on the end of a telephone, quick response time and system reliability are paramount.

If the AD for such a system does not consider the quality properties of the system in any detail—in particular, if there is no clear definition of response-time requirements nor any performance models—it is quite likely that when the system is deployed, it will deliver poor performance, particularly under peak load.

The solution to this is to identify a group of users who can agree on what the performance requirements are, and then the architect can balance these against what analysis and testing reveal is practically possible. This helps avoid the significant amount of enhancement and tuning inevitably required when performance problems emerge later in the lifecycle.

The architect writes the AD and is also one of its major users. You use the AD as a memory aid, a basis for analysis, a record of decisions, and so on. However, you are only one of the users of the AD. To a lesser or greater extent, all of the other stakeholders need to understand the architecture (or at least parts of it) as it relates to them. If the AD does not help with this, it has failed.

**Principle** A good architectural description is one that effectively and consistently communicates the key aspects of the architecture to the appropriate stakeholders.

Nowadays there is a plethora of techniques, models, architecture description languages, and other ways to document architectures. Choosing the right ones for a particular system development is a significant challenge in its own right; you need to take into account the characteristics of the system and the skills and capabilities of its stakeholders.

Part II explores the concept of ADs in more detail, and Parts III and IV explain the different elements of an AD and how to create them.

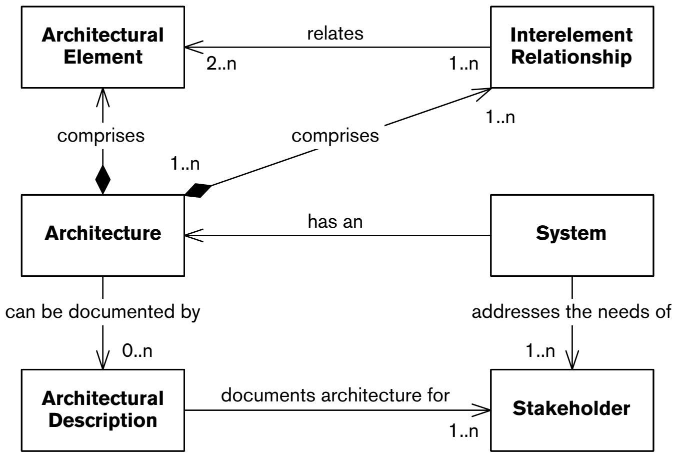

**FIGURE 2–5** Core Concept Relationships

## Interrelationships between the Core Concepts

The important relationships between our core concepts are illustrated in the UML class diagram in Figure 2–5. The diagram brings out the following relationships between the concepts we have discussed so far.

- A system is built to address the needs, concerns, goals, and objectives of its stakeholders.
- The architecture of a system is comprised of a number of architectural elements and their interelement relationships.
- The architecture of a system can potentially be documented by an AD (fully, partly, or not at all). In fact, there are many potential ADs for a given architecture, some good, some bad.
- An AD documents an architecture for its stakeholders and demonstrates to them that it has met their needs.

## Summary

In this chapter we laid our foundations by defining and discussing some concepts and terms we will be using throughout the rest of the book.

- The *architecture* of a system defines four different aspects: its *static structure*, its *dynamic structure*, its *externally visible behavior*, and its *quality properties*. Each of these aspects is important, although not always addressed. Every computer system has an architecture, even if we don't understand it.

- A *candidate architecture* for a system is one that has the potential to exhibit the system's required externally visible behaviors and quality properties. Most problems have several candidate architectures, and it is the job of the architect to select the best one.
- An *architectural element* is a clearly identifiable, architecturally meaningful piece of a system.
- A *stakeholder* is a person, group, or entity with an interest in or concerns about the realization of the architecture. Stakeholders include users but also many people, such as developers, operators, and acquirers. Architectures are created solely to meet stakeholder needs.
- An *architectural description* is a set of products that documents an architecture in a way its stakeholders can understand and demonstrates that the architecture has met their concerns. Although every system has an architecture, not every system has an effective AD.

## Further Reading

We have aligned our language and concepts in this chapter with the only formal general standard we are aware of in the field of software architecture—IEEE Standard 1471 [IEEE00]. According to its own introduction, this standard "addresses the activities of creation, analysis, and sustainment of architectures of software-intensive systems, and the recording of such architectures in terms of architectural descriptions." Our conceptual model is based on the one presented in the standard.

Much of our thinking on software architecture concepts is based on the work done by the Software Architecture group of the Software Engineering Institute. The book by Bass, Clements, and Kazman [BASS03] is a thorough introduction to the main ideas in the field of software architecture and provides a lot more depth and background on the fundamental concepts than we provide here.

One of the original books on software architecture is by Shaw and Garlan [SHAW96]. This book provides a minimalist and elegant introduction to the fundamental ideas in software architecture, including overviews of an AD, architectural styles, and possible tool support. Even earlier than this, one of the original papers in the software architecture field, by Perry and Wolf [PERR92], is well worth reading for its clear focus on the important elements of the discipline.

Another clear and thought-provoking overview of software architecture appears in a book that is actually the report from a research project funded by the European Commission [JAZA00] (which is unfortunately out of print at present). The first chapter provides an interesting introduction to the field.

If you want to take a wider view of software architecture, a useful book may be the cross-disciplinary *Art of Systems Architecting* [MAIE00]. This book is novel in that it introduces and discusses the idea of architecture (and "architecting") being a set of principles and techniques valid across all complex systems domains. A particular emphasis is placed on architecture heuristics, and a set of interesting heuristics is provided. Examples are taken from buildings, manufacturing, social systems, IT, and collaborative systems.

# 3

# Viewpoints and Views

When you start the daunting task of designing the architecture of your system, you will find that you have some difficult architectural questions to answer.

- What are the main functional elements of your architecture?
- How will these elements interact with one another and with the outside world?
- What information will be managed, stored, and presented?
- What physical hardware and software elements will be required to support these functional and information elements?
- What operational features and capabilities will be provided?
- What development, test, support, and training environments will be provided?

A common temptation—one you should strongly avoid—is to try to answer all of these questions by means of a single, heavily overloaded, all-encompassing model. This sort of model (and we've all seen them) will probably use a mixture of formal and informal notations to describe a number of aspects of the system on one huge sheet of paper: the functional structure, software layering, concurrency, intercomponent communication, physical deployment environment, and so on. Let's see what happens when we try to use an all-encompassing model in our AD, by means of an example (see page 28).

As the example shows, this sort of AD is really the worst of all worlds. Many writers on software architecture have pointed out that it simply isn't possible to describe a software architecture by using a single model. Such a model is hard to understand and is unlikely to clearly identify the architecture's most

**Example** Although the airline reservation system we introduced in Chapter 2 is conceptually fairly simple, in practice some aspects of this system make it very complicated indeed.

- The system's data is distributed across a number of systems in different physical locations.
- A number of different types of data entry devices must be supported.
- The system must be able to present some information in different languages.
- The system must be able to print tickets and other documents on a wide range of printers.
- The plethora of international regulation complicates the picture even further.

After some discussion, the architect draws up a first-cut architecture for the system, which attempts to represent all of its important aspects in a single diagram. This model includes the full range of data entry devices (including various dumb terminals, desktop PCs, and wireless devices), the multiple physical systems on which data is stored or replicated data is maintained, and some of the printing devices that must be supported (the model does not cover remote printing because it is done at a separate facility). The model is heavily annotated with text to indicate, for example, where multilanguage support is required and where data must be audited, archived, or analyzed to support regulatory requirements.

However, no details of the network interfaces between the different components are included—these are abstracted out into a network icon because these are so complex. (In fact, the network design is probably the most complicated aspect of the architecture, requiring support for a number of different and largely incompatible network protocols, routing over public and private networks, synchronous and asynchronous interactions, and varying levels of service reliability and availability.) Furthermore, the model does not address any of the implications of having the same data distributed around multiple systems.

Because it is so complex and tries to address a wide mix of concerns in the same diagram, the model fails to engage any of the stakeholders. The users find it too complex and difficult to understand (particularly because of the large number of physical hardware components represented). The technology stakeholders, on the other hand, tend to disregard it because of the detail that is left out, such as the network topology. The legal team members can't use it to satisfy themselves that the regulatory aspects will be adequately handled, and the sponsor finds it completely incomprehensible.

Furthermore, the architect spends an inordinate amount of time keeping it up-to-date—every time a new type of data entry device or printer is discussed, for example, the diagram needs to be updated and reprinted on a very large sheet of paper.

Because of these problems, the diagram soon becomes obsolete and is eventually forgotten. Unfortunately, the issues that the model fails to address do not disappear and thus cause many problems and delays during the implementation and the early stages of live operation.

important features. It tends to poorly serve individual stakeholders because they struggle to understand the aspects that interest them. Worst of all, because of its complexity, a monolithic AD is often incomplete, incorrect, or out-of-date.

**Principle** It is not possible to capture the functional features and quality properties of a complex system in a single comprehensible model that is understandable by and of value to all stakeholders.

We need to represent complex systems in a way that is manageable and comprehensible by a range of business and technical stakeholders. A widely used approach—the only successful one we have found—is to attack the problem from different directions simultaneously. In this approach, the AD is partitioned into a number of separate but interrelated **views**, each of which describes a separate aspect of the architecture. Collectively, the views describe the whole system.

To help you understand what we mean by a view, let's consider the example of an architectural drawing for one of the elevations of an office block. This portrays the building from a particular aspect, typically a compass bearing such as northeast. The drawing shows features of the building that are visible from that vantage point but not from other directions. It doesn't show any details of the interior of the building (as seen by its occupants) or of its internal systems (such as plumbing or air conditioning) that influence the environment its occupants will inhabit. Thus the blueprint is only a partial representation of the building; you have to look at—and understand—the whole set of blueprints to grasp the facilities and experience that the whole building will provide.

Another way that a building architect might represent a new building is to construct a scale model of it and its environs. This shows how the building will look from all sides but again reveals nothing about its interior form or its likely internal environment.

**STRATEGY** A complex system is much more effectively described by using a set of interrelated views, which collectively illustrate its functional features and quality properties and demonstrate that it meets its goals.

Let's take a look at what this approach means for software architecture.

## ARCHITECTURAL VIEWS

An architectural view is a way to portray those aspects or elements of the architecture that are relevant to the concerns the view intends to address—and, by implication, the stakeholders for whom those concerns are important.

This idea is not new, going back at least as far as the work of David Parnas in the 1970s and more recently Dewayne Perry and Alexander Wolf in the early 1990s. However, it wasn't until 1995 that Phillipe Kruchten of the Rational Corporation published his widely accepted written description of viewpoints, *Architectural Blueprints—The 4+1 View Model of Software Architecture*. This suggested four different views of a system and the use of a set of scenarios (use cases) to check their correctness. Kruchten's approach has since evolved to form an important part of the Rational Unified Process (RUP).

More recently, IEEE Standard 1471 has formalized these concepts and brought some welcome standardization of terminology. In fact, our definition of a view is based on and extends the one from the IEEE standard.

**DEFINITION** A **view** is a representation of one or more structural aspects of an architecture that illustrates how the architecture addresses one or more concerns held by one or more of its stakeholders.

When deciding what to include in a view, ask yourself the following questions.

- What class(es) of stakeholder is the view aimed at? A view may be narrowly focused on one class of stakeholder or even a specific individual, or it may be aimed at a larger group whose members have varying interests and levels of expertise.
- How much technical understanding do these stakeholders have? Acquirers and users, for example, will be experts in their subject areas but are unlikely to know much about hardware or software, while the converse may apply to developers or support staff.

- What stakeholder concerns is the view intended to address? How much do the stakeholders know about the architectural context and background to these concerns?
- How much do these stakeholders need to know about this aspect of the architecture? For nontechnical stakeholders such as users, how competent are they in understanding its technical details?

As with the AD itself, one of your main challenges is to get the right level of detail into your views. Provide too much detail, and your audience will be overwhelmed; too little, and you risk your audience making assumptions that may not be valid.

**Strategy** Include in a view only the details that further the objectives of your AD—that is, those details that help explain the architecture to stakeholders or demonstrate that stakeholder concerns are being met.

## Viewpoints

It would be hard work if every time you were creating a view of your architecture you had to go back to first principles to define what should go into it. Fortunately, you don't quite have to do that.

In his introductory paper, Kruchten defined four standard views, namely, Logical, Process, Physical, and Development. The IEEE standard makes this idea generic (and does not specify one set of views or another) by proposing the concept of a *viewpoint*.

The objective of the viewpoint concept is an ambitious one—no less than making available a library of templates and patterns that can be used off the shelf to guide the creation of an architectural view that can be inserted into an AD. We define a viewpoint (again after IEEE Standard 1471) as follows.

**Definition** A **viewpoint** is a collection of patterns, templates, and conventions for constructing one type of view. It defines the stakeholders whose concerns are reflected in the viewpoint and the guidelines, principles, and template models for constructing its views.

Architectural viewpoints provide a framework for capturing reusable architectural knowledge that can be used to guide the creation of a particular type of (partial) AD.

In a relatively unstructured activity like architecture definition, the idea of the viewpoint is very appealing. If we can define a standard approach, a standard language, and even a standard metamodel for describing different aspects of a system, stakeholders can understand any AD that conforms to these standards once familiar with them.

In practice, of course, we haven't achieved this goal yet. There are no universally accepted ways to model software architectures, and every AD uses its own conventions. However, the widespread acceptance of techniques such as entity-relationship models and of modeling languages such as UML takes us some way toward this goal.

In any case, it is extremely useful to be able to categorize views according to the types of concerns and architectural elements they present.

**STRATEGY** When developing a view, whether or not you use a formally defined viewpoint, be clear in your own mind what sorts of concerns the view is addressing, what types of architectural elements it presents, and who the viewpoint is aimed at. Make sure that your stakeholders understand these as well.

## INTERRELATIONSHIPS BETWEEN THE CORE CONCEPTS

To put views and viewpoints in context, we can now extend the conceptual model we introduced in Chapter 2 to illustrate how views and viewpoints contribute to the overall picture (see Figure 3–1).

We have added the following relationships to the diagram we originally presented as Figure 2–5.

- A viewpoint defines the aims, intended audience, and content of a class of views and defines the concerns that views of this class will address.
- A view conforms to a viewpoint and so communicates the resolution of a number of concerns (and a resolution of a concern may be communicated in a number of views).
- An AD comprises a number of views.

## THE BENEFITS OF USING VIEWPOINTS AND VIEWS

Using views and viewpoints to describe the architecture of a system benefits the architecture definition process in a number of ways.

- *Separation of concerns*: Describing many aspects of the system via a single representation can cloud communication and, more seriously, can result in independent aspects of the system becoming intertwined in the

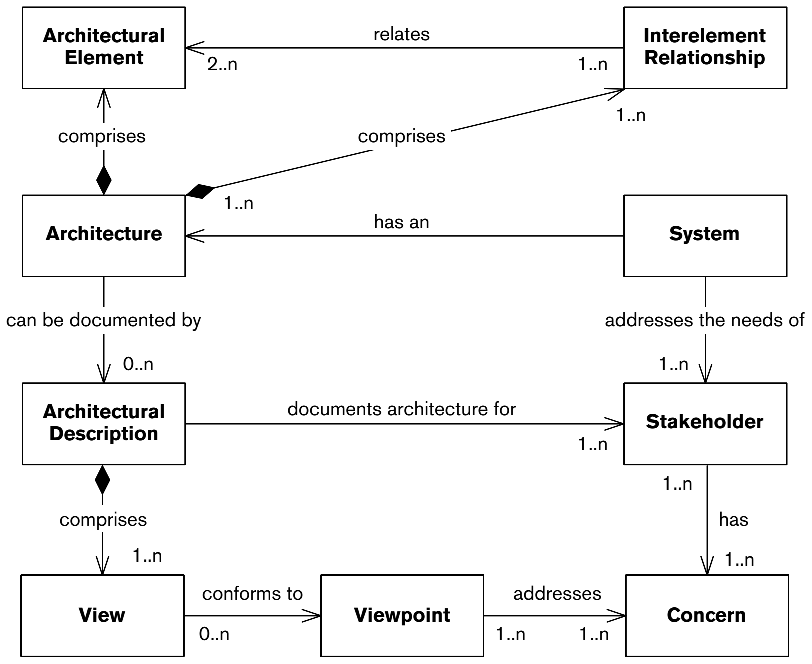

**FIGURE 3–1** VIEWS AND VIEWPOINTS IN CONTEXT

model. Separating different models of a system into distinct (but related) descriptions helps the design, analysis, and communication processes by allowing you to focus on each aspect separately.

- *Communication with stakeholder groups*: The concerns of each stakeholder group are typically quite different (e.g., contrast the primary concerns of end users, security auditors, and help-desk staff), and communicating effectively with the various stakeholder groups is quite a challenge. The viewpoint-oriented approach can help considerably with this problem. Different stakeholder groups can be guided quickly to different parts of the AD based on their particular concerns, and each view can be presented using language and notation appropriate to the knowledge, expertise, and concerns of the intended readership.
- *Management of complexity*: Dealing simultaneously with all of the aspects of a large system can result in overwhelming complexity that no one person can possibly handle. By treating each significant aspect of a system separately, the architect can focus on each in turn and so help conquer the complexity resulting from their combination.

- *Improved developer focus*: The AD is of course particularly important for the developers because they use it as the foundation of the system design. By separating out into different views those aspects of the system that are particularly important to the development team, you help ensure that the right system gets built.

## VIEWPOINT PITFALLS

Of course, the use of views and viewpoints won't solve all of your software architecture problems automatically. Although we have found that using views is really the only way to make the problem manageable, you need to be aware of some possible pitfalls when using the view-based approach.

- *Inconsistency*: Using a number of views to describe a system inevitably brings consistency problems. It is theoretically possible to use architecture description languages to create the models in your views and then cross-check these automatically (much as graphical modeling tools attempt to check structured or object-oriented methods models), but there are no such machine-checkable architecture description languages in widespread use today. This means that achieving cross-view consistency within an AD is an inherently manual process and so, to assist with this, Chapter 22 includes a checklist to help you ensure consistency between the standard viewpoints presented in our catalog in Part III.
- *Selection of the wrong set of views*: It is not always obvious which set of views is suitable for describing a particular system. This is influenced by a number of factors, such as the nature and complexity of the architecture, the skills and experience of the stakeholders (and of the architect), and the time available to produce the AD. There really isn't an easy answer to this problem, other than your own experience and skill and an analysis of the most important concerns that affect your architecture.
- *Fragmentation*: When you start to describe your architecture, one temptation is to create a large number of views. This can lead to your architecture being described by many independent models, each in a separate view, making the AD difficult to follow. Each separate view also involves a significant amount of effort to create and maintain. To avoid fragmentation and minimize the overhead of maintaining unnecessary descriptions, you should eliminate views that do not address significant concerns for the system you are building. In some cases, you may also consider creating hybrid views that combine models from a number of views in the viewpoint set (e.g., creating a combined deployment and concurrency view). Beware, however, of the combined views becoming

difficult to understand and maintain because they address a combination of concerns.

## OUR VIEWPOINT CATALOG

Part III of this book presents our catalog of six core viewpoints for information systems architecture: the Functional, Information, Concurrency, Development, Deployment, and Operational viewpoints. Although the viewpoints are (largely) disjoint, we find it convenient to group them as shown in Figure 3–2 .

- The Functional, Information, and Concurrency viewpoints characterize the fundamental organization of the system.
- The Development viewpoint exists to support the system's construction.
- The Deployment and Operational viewpoints characterize the system once in its live environment.

Table 3–1 briefly describes our viewpoints.

Of course, not all of these viewpoints may apply to your architecture, and some will be more important than others. You may not need views of all of these types in your AD, and in some cases there may even be other more specialized viewpoints that you need to identify and add yourself. This means that your first job is to understand the nature of your architecture, the skills and experience of the stakeholders, and the time available and other constraints, and then to come up with an appropriate selection of views.

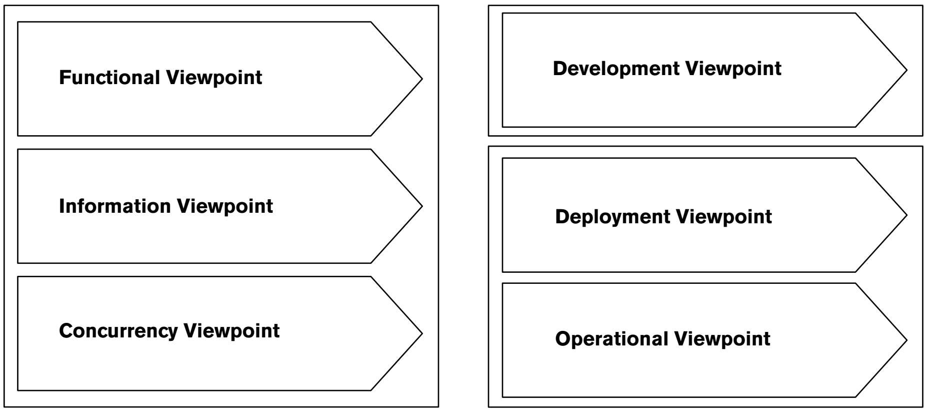

**FIGURE 3–2** VIEWPOINT GROUPINGS

**TABLE 3–1** Viewpoint Catalog

| Viewpoint | Definition |
|---|---|
| Functional | Describes the system's functional elements, their responsibilities, interfaces, and primary interactions. A Functional view is the cornerstone of most ADs and is often the first part of the description that stakeholders try to read. It drives the shape of other system structures such as the information structure, concurrency structure, deployment structure, and so on. It also has a significant impact on the system's quality properties such as its ability to change, its ability to be secured, and its runtime performance. |
| Information | Describes the way that the architecture stores, manipulates, manages, and distributes information. The ultimate purpose of virtually any computer system is to manipulate information in some form, and this viewpoint develops a complete but high-level view of static data structure and information flow. The objective of this analysis is to answer the big questions around content, structure, ownership, latency, references, and data migration. |
| Concurrency | Describes the concurrency structure of the system and maps functional elements to concurrency units to clearly identify the parts of the system that can execute concurrently and how this is coordinated and controlled. This entails the creation of models that show the process and thread structures that the system will use and the interprocess communication mechanisms used to coordinate their operation. |
| Development | Describes the architecture that supports the software development process. Development views communicate the aspects of the architecture of interest to those stakeholders involved in building, testing, maintaining, and enhancing the system. |
| Deployment | Describes the environment into which the system will be deployed, including capturing the dependencies the system has on its runtime environment. This view captures the hardware environment that your system needs (primarily the processing nodes, network interconnections, and disk storage facilities required), the technical environment requirements for each element, and the mapping of the software elements to the runtime environment that will execute them. |
| Operational | Describes how the system will be operated, administered, and supported when it is running in its production environment. For all but the simplest systems, installing, managing, and operating the system is a significant task that must be considered and planned at design time. The aim of the Operational viewpoint is to identify system-wide strategies for addressing the operational concerns of the system's stakeholders and to identify solutions that address these. |

## Summary

Capturing the essence and the detail of the whole architecture in a single model is just not possible for anything other than simple systems. If you try to do this, you will end up with a Frankenstein monster of a model that is unmanageable and does not adequately represent the system to you or any of the stakeholders.

By far the best way of managing this complexity is to produce a number of different representations of all or part of the architecture, each of which focuses on certain aspects of the system, showing how it addresses some of the stakeholder concerns. We call these *views*.

To help you decide what views to produce and what should go into any particular view, you use *viewpoints*, which are standardized definitions of view concepts, content, and activities.

The use of views and viewpoints brings many benefits, such as separation of concerns, improved communication with stakeholders, and management of complexity. However, it is not without its pitfalls, such as inconsistency and fragmentation, and you must be careful to manage these.

In this chapter, we introduced our viewpoint catalog, comprising the Functional, Information, Concurrency, Development, Deployment, and Operational viewpoints, which we describe in detail in Part III.

## Further Reading

A lot of useful guidance on creating ADs using views (including a discussion of when and how to combine views) and thorough guidance for creating the documentation for a wide variety of types of view can be found in Clements et al. [CLEM03]. Other references that help to make sense of viewpoints and views are IEEE Standard 1471 [IEEE00] and Kruchten's "4+1" approach [KRUC95]. One of the earliest explicit references to the need for architectural views appears in Perry and Wolf [PERR92].

Some of the other viewpoint taxonomies that have been developed over the last decade or so—including Kruchten's "4+1"; RM-ODP; the viewpoint set by Hofmeister et al. [HOFM00]; and the set by Garland and Anthony [GARL03]—are described in the Appendix, together with recommendations for further reading in this area.

Part III, where we describe our viewpoint catalog in detail, contains references for specific view-related reading.