3

Basic Concepts

The preceding chapters informally discussed a number of software architectural notions, such as software components and connectors; their configurations in a given system; relationships between a system's requirements, its architecture, and its implementation; and software product lines. These ideas served to provide context for the field of software architecture, situate it within other facets of software engineering, and motivate its unique role and importance. Indeed, we have generally avoided definitions of terms in the hope that the reader will more readily recognize many of the discussed concepts within his own experience, or be able to relate the architectural concepts to those with which he is already familiar.

While informal terms have a useful role in introducing concepts, the uncertainties that result can hinder deeper understanding. Hence, the objective of this chapter is to define the key terms and ideas from the field of software architecture, providing a uniform basis for their discussion in the remainder of the book. Key elements of architecture-centric design and their interrelationships, basic techniques and processes for developing a software system's architecture, and the relevant stakeholders and their roles in architecture-based software development are presented and illustrated with simple examples.

Outline of Chapter 3

3.1 TERMINOLOGY

We begin presentation of the field's key terms with an exploration of software architecture itself and several concepts tied to it, such as architectural degradation. The major constituent elements of architectures are then explored, including components, connectors, and configurations. Two important types of potentially deep architectural knowledge—architectural patterns and styles—are then defined and examined.

3.1.1 Architecture

At its essence, software architecture is defined quite simply, as follows.

> ***Definition.* A software system's *architecture* is the set of principal design decisions made about the system.**

Put another way, software architecture is the blueprint for a software system's construction and evolution. The notion of *design decision* is central to software architecture and to all of the concepts based on it. For example, the special notion of *product family architectures* was briefly introduced Chapter 1. Product family architectures are anchored on the idea of reference architecture, defined as follows.

> ***Definition.* A *reference architecture* is the set of principal design decisions that are simultaneously applicable to multiple related systems, typically within an application domain, with explicitly defined points of variation.**

Design decisions encompass every aspect of the system under development, including:

- Design decisions related to system *structure*—for example, "The architectural elements should be organized and composed exactly like this . . ."
- Design decisions related to *functional behavior*—for example, "Data processing, storage, and visualization will be performed in strict sequence."
- Design decisions related to *interaction*—for example, "Communication among all system elements will occur only using event notifications."
- Design decisions related to the system's *nonfunctional properties*—for example, "The system's dependability will be ensured by replicated processing modules."
- Design decisions related to the system's *implementation*—for example, "The user interface components will be built using the Java Swing toolkit."

Note that in the preceding discussion and examples we have used terms such as *element* and *module*, which we have not yet defined. You can think of them simply as building blocks (akin to bricks in the construction of physical structures) from which the architecture is composed. We soon will address them more rigorously.

Another important term that appears in the above definitions is *principal*. It implies a degree of importance and topicality that grants a design decision architectural status, that is, that makes it an *architectural design decision*. It also implies that not all design decisions are architectural. In fact, many of the design decisions made in the process of engineering a system (for example, the details of the selected algorithms or data structures) will not impact a system's architecture. How one defines principal will depend on the system goals. Ultimately, the system's stakeholders (including, but not restricted only to the architect)

will decide which design decisions are or are not important enough to include in the architecture.

Based on this, note that architecture is at least in part determined by context, that nontechnical considerations may end up driving it, and that different sets of stakeholders may deem different sets of design decisions principal (that is, architectural). We discuss this further below and elsewhere in the text.

Another observation following from the definition of software architecture is that every set of principal design decisions can be thought of as a different architecture. Over the lifetime of a system, these design decisions will be made and unmade; they will change, evolve, "fork," converge, and so on. As the architecture of a large, complex, long-lived system is developed and evolved, the corresponding set of architectural design decisions will be changed hundreds of times. The outcome will, in effect, be hundreds of different (though related) architectures, rather than a single architecture. In that sense, architecture has a temporal aspect.

Prescriptive Architecture versus Descriptive Architecture

At any time, t, during the process of engineering a software system, that system's architects will have made a set of architectural design decisions, P, that reflect their intent. These design decisions comprise the system's *prescriptive architecture*. In other words, these design decisions represent the prescription for the system's construction. The prescriptive architecture is thus the system's as-intended or as-conceived architecture. The prescriptive architecture need not necessarily exist in any tangible form. For example, it may be entirely in the architects' minds. Alternatively, the prescriptive architecture may have been captured in a notation such as an architecture description language (such as those presented in Chapter 6) or another form of documentation.

It is important to note that documenting the prescriptive architecture is not enough. The reader should recall that architecture is not just a phase in the process of developing a software system, but rather forms the critical underpinning for the system from its inception through retirement. Thus, at any point during this process, the architectural design decisions that are part of the prescriptive architecture (that is, of the set P) will putatively be refined and *realized* with a set of artifacts, A. These artifacts may include refinements of architectural design decisions in a notation such as the Unified Modeling Language or their implementations in a programming language. The artifacts may also include models of architectural styles and patterns used in the architecture, previously existing off-the-shelf software components that will be used in the desired system, implementation frameworks and middleware infrastructures that will aid the system's construction, specifications of standards to which the architecture needs to adhere, and so on.

While many of the artifacts in A may have existed prior to and independently of the architecture under consideration, each embodies certain design decisions that the architects find desirable and relevant to the current system. The full set of principal design decisions, D, embodied in the set of artifacts, A, is referred to as the system's *descriptive architecture*. The descriptive architecture is referred to as such because it describes how the system has been realized. The descriptive architecture is thus the system's as-realized architecture.

The reader should note that, in the early stages of a system's life cycle, the number of artifacts that realize the architecture will typically be smaller than during later stages when more of the system's architecture has been elaborated and possibly implemented.

In fact, if we consider time t_1 to indicate the inception of the initial set P_1 of architectural design decisions for a given system, the sets A_1, and thus D_1, may be empty. This will typically be the case in so-called greenfield development, where the system is designed and implemented from scratch. In the case of brownfield development, many artifacts partially realizing the architecture for a given system may exist before the architecture is even conceived; in other words, at the start of a project, t_0, the set D_0 is nonempty while P_0 may be an empty set. This is the case if the new system is a member of a closely related application family, the architectural styles and/or patterns that will be used are known ahead of time, the middleware platforms and/or implementation languages have been selected, and so on. Another situation in which P_0 may be empty while D_0 is large is development involving a legacy system whose architectural intent has been lost over time. Such discrepancies between sets P and D can be indications of problems with a system's software architecture, and will be discussed further below.

Prescriptive and Descriptive Architectures in Action: An Example

Let us illustrate the above discussion with a simple example. At this point, the reader is not expected to understand all the nuances and implications of this example, but only to follow along with the argument as it pertains to prescriptive and descriptive architecture.

Figure 3-1 shows a graphical view of the prescriptive architecture of a simple logistics application. In other words, the diagram in the figure is a model of the architecture in a graphical architecture description language. The architecture is designed using a set of components, which implement the application's functionality, and connectors, which enable the components to call each other's operations and exchange data. (Components and connectors will be defined more formally later in this chapter.)

This application controls the routing of cargo from a set of incoming Delivery Ports to a set of Warehouses via a set of Vehicles. The Cargo Router component tries to optimize the use of vehicles and deliver the cargo to its proper destination (that is, warehouse). The Clock component provides the time and helps the ports, vehicles, and warehouses to synchronize as necessary. Finally, the Graphics Binding component provides a graphical user interface that allows a human operator to assess the state of the system during its execution. The components in the system interact via three connectors—Clock Conn, Router Conn, and Graphics Conn—by exchanging requests and replies. The lines connecting the component and connector elements in the architecture represent the elements' interaction paths.

The cargo routing application is useful in this discussion for three reasons. First, its simplicity suggests that its architects should have been able to evaluate all of the architectural decisions and their trade-offs before the application was implemented. Second, the architecture was carefully implemented with the help of an architectural framework (see Chapter 9 for a discussion of architectural frameworks), which allowed the application's developers to realize all of the architectural design decisions directly in the code. Third, aside from a graphical user interface (GUI) library, no other off-the-shelf functionality was used in the implementation, which allowed for a carefully controlled mapping between the architecture and its implementation (that is, between the sets P and D).

A graphical view of the descriptive (that is, as-realized) architecture of the cargo routing application is depicted in Figure 3-2. While there are many artifacts in the set A of this application (including the architectural style used to realize the architecture, the implementation framework, the off-the-shelf GUI toolkit, and the implementation code

itself), we have deliberately elided many of these and extracted the simplified view of the descriptive architecture, *D*, from them.

It can be noticed immediately that the extracted view of the descriptive architecture is not identical to the prescriptive architecture: the Vehicle component is not connected to the Clock Conn connector, while the Router Conn and Clock Conn connectors are connected, allowing two-hop routing of requests and replies in the architecture. It is thus possible for the Clock component to interact directly with the Cargo Router component via the two connectors. The application-specific reasons behind these changes are unimportant to this discussion. What is important is the fact that the programmers and architects together discovered that, even in the case of such a simple application, they had not properly thought through all of the architectural design decisions. It is, then, reasonable to expect that such issues would only be exacerbated in larger, more complex systems. In such systems, the prescriptive and descriptive architectures may not look nearly as similar to one another. This is an issue that will be revisited several times throughout the book.

Let us take this discussion a step further. We can ask several questions about the prescriptive and descriptive architectures of the cargo routing application. Again, the reader is not yet expected to be able to provide complete answers to these questions. Instead, the purpose is to illustrate certain issues that will be revisited and sensitize the reader to the difficulties inherent in the architectural design of even moderately complex software systems.

1. Which architecture is "correct?" The prescriptive architecture reflects the architects' intent, but the descriptive architecture reflects the added experience of actually implementing the system.
2. Are the two architectures consistent with one another? In this example, the differences were relatively easy to spot. However, architectural inconsistencies can be much more complex and/or insidious.
3. What criteria are used to establish the consistency between the two architectures? Again, in this example a simple structural comparison sufficed, but more sophisticated techniques may be needed.
4. On what information is the answer to the preceding questions based? The two diagrams in Figure 3-1 and Figure 3-2 represent very small subsets of different concerns that may be captured and/or visualized in an architectural model. Even if these two diagrams were identical, the amount of architectural information they represent is insufficient to make any guarantees about the two architectures' relationship.

Architectural Degradation

During the lifespan of a typical software system, a large number of prescriptive and descriptive architectures will be created. Each corresponding pair of such architectures represents the system's software architecture at a given time, *t*. As the set of principal design decisions, *P*, the set of artifacts, *A*, and the corresponding principal design decisions, *D*, embodied in the artifacts, grow, so the system's software architecture becomes more complete. The system's stakeholders will decide the state at which *P* and *A* (and thus *D*) can be considered sufficiently complete and sufficiently consistent with one another for the system to be released into operation.

In an ideal scenario, the two sets of architectural design decisions, *P* and *D*, would always be identical; in other words, *D* would always be a perfect realization of *P*. However,

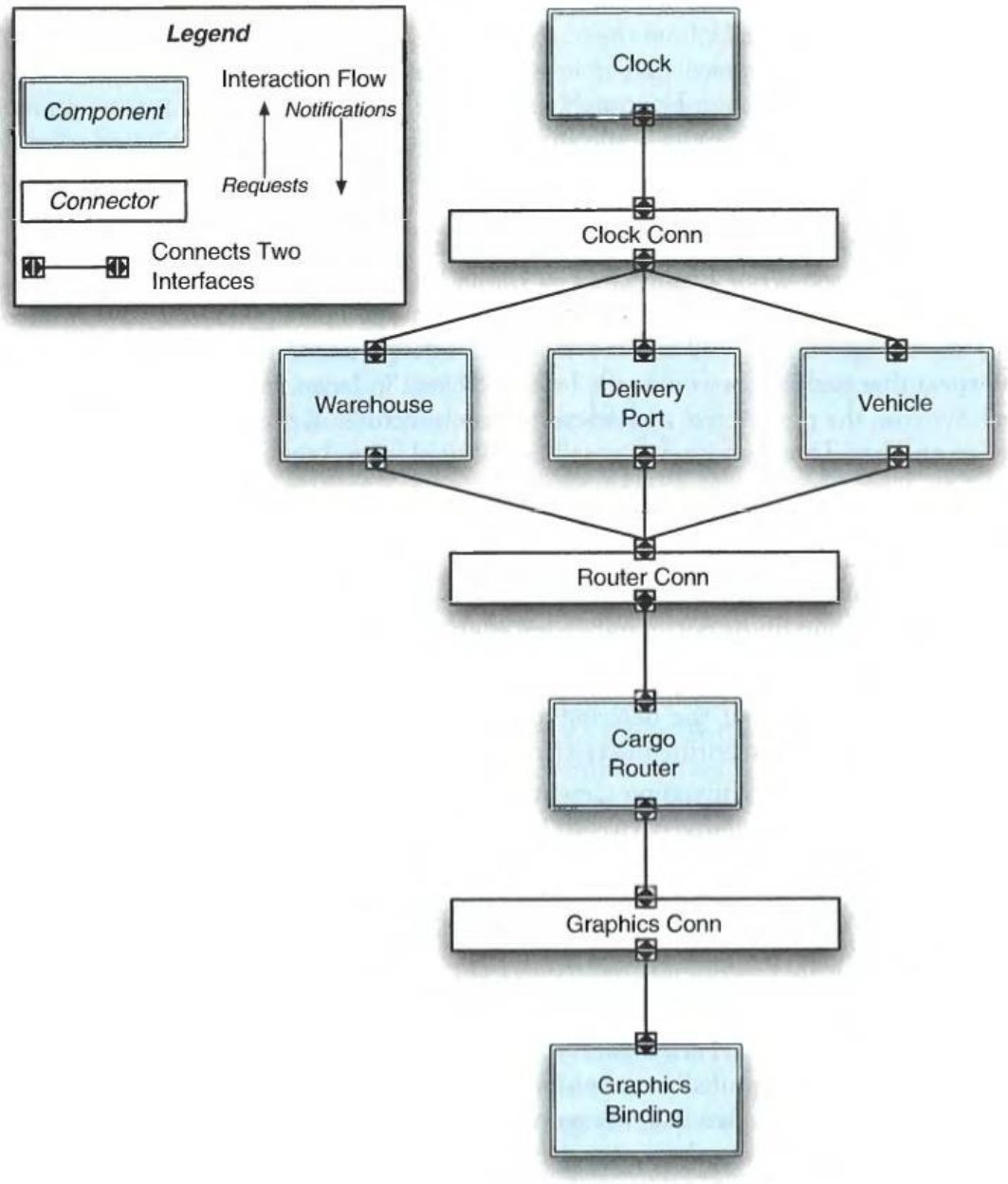


Figure 3-1. *A high-level graphical view of the prescriptive (that is, as-designed or as-intended) architecture of the cargo routing application.*

this need not be the case. For example, an off-the-shelf component or middleware platform will likely embody a number of design decisions that may, in turn, impact the architectural design decisions made for the system under construction. Thus, the exact relationship between sets P and D may vary depending on the system in question, the requirements imposed by system stakeholders, the point in the system's lifespan, and so on. However, it is imperative that the stakeholders, and architects in particular, understand this relationship and be precise about the allowed differences between P and D.

Note that, given sets of design decisions in P and D at time t, it is possible for these sets to remain stable to time $t + n$. This simply means that though the set A may have enlarged as implementation progresses, the addition of more artifacts or the further development of

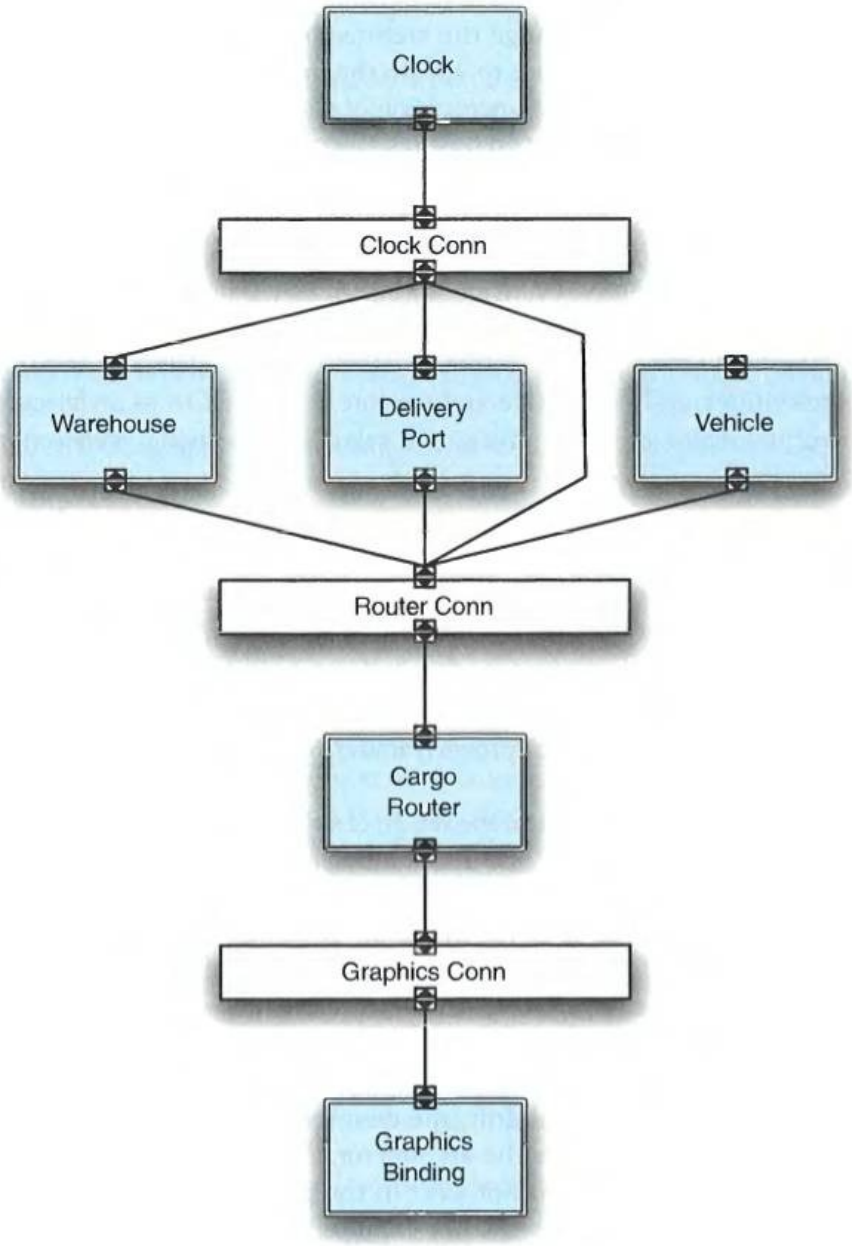


Figure 3-2. *A high-level, graphical view of the descriptive (that is, as-implemented or as-realized) architecture of the cargo routing application.*

existing artifacts did not introduce any new principal design decisions. It is also possible that *P* changes while *D* remains the same (for instance, when architectural design concerns are elaborated but the artifacts realizing the system have not yet been updated); similarly, *D* may change while *P* remains the same.

When a system is initially developed or when the already implemented system is evolved, ideally its prescriptive architecture is first modified appropriately, and then the corresponding changes to the descriptive architecture follow. Unfortunately, this does not always happen in practice. Instead, the system (and thus its descriptive architecture) is often directly modified, without accounting for the impact relative to the prescriptive architecture.

In the cargo routing example, the developers may not have bothered to inform the architects that they decided to change the architecture in several places—even if those changes were warranted. Such failure to update the prescriptive architecture happens for several reasons: developer sloppiness, perception of short deadlines that prevent thinking through and documenting the impact on the prescriptive architecture, lack of a documented prescriptive architecture, need or desire to optimize the system "which can be done only in the code," inadequate techniques and tool support, and so forth.

Whatever the reasons, they are flawed and potentially dangerous, especially if one considers that software systems are notorious for containing many errors (that is, for being "buggy"). Do we really want the as-realized architecture, with all of its potentially latent faults, to be the final arbiter of the architects' *intent*? The resulting discrepancy between a system's prescriptive and descriptive architecture is referred to as *architectural degradation*. Architectural degradation comprises two related phenomena: *architectural drift* and *architectural erosion*.

> ***Definition. Architectural drift* is introduction of principal design decisions into a system's descriptive architecture that (a) are not included in, encompassed by, or implied by the prescriptive architecture, but which (b) do not violate any of the prescriptive architecture's design decisions.**

Architectural drift does not necessarily result in outright violations of the prescriptive architecture. Instead, it circumvents the prescriptive architecture and may involve decisions whose implications are not properly understood and which may affect the given system's future adaptability.

Drift is a result of direct changes to the set, A, of system artifacts. These changes may, in turn, result in changes to the set, D, of corresponding principal design decisions. Note that all expansions of the set D do not necessarily result in architectural drift. New principal design decisions may be added to D (as the result of adding or changing elements of A) that are consistent with all the decisions in P and that are implied by or encompassed by decisions in P. For instance, P may require encryption to be used in any communication over an open network; D may state that public-key algorithms be used to support such encrypted communication—a decision encompassed by P.

As an example of architectural drift, the descriptive architecture in the cargo routing application (Figure 3-2) reflects the architectural design decision to introduce a link between two connectors, which did not exist in the prescriptive architecture. Assuming that the system's original architects did not explicitly prohibit the direct linking of two connectors, adding the link would not violate any of the architectural decisions made in the prescriptive architecture. However, it does not mean that adding this link is a harmless or proper thing to do.

Architectural drift may also cause violations of architectural style rules. In other words, architectural drift reflects the engineers' insensitivity to the system's architecture and can lead to a loss of clarity of form and system understanding (Perry and Wolf 1992). If not properly addressed, architectural drift will eventually result in architectural erosion.

> ***Definition. Architectural erosion* is the introduction of architectural design decisions into a system's descriptive architecture that violate its prescriptive architecture.**

In terms of the architectural design decision sets, P and D, architectural erosion can be thought of as the result of direct changes to the system artifact set, A, which introduces a principal design decision in D that invalidates or violates one or more design decisions

that already exist in *P*. Erosion renders a system difficult to understand and adapt, and also frequently leads to system failure.

Architectural erosion can easily occur when a system has drifted too far, in that it is easy to violate important architectural decisions if those decisions are obscured by many small, intermediate changes. Note that an architecture can certainly erode without previous drift. However, this is both less likely to happen (since there has been no drift yet, the prescriptive and descriptive architectures are consistent at the time the erroneous changes are made) and easier to recover from (since fewer decisions are involved). Note also that architectural erosion may be caused by design decisions that are sound when considered in isolation. However, in combination with other decisions that have been made, but perhaps not properly documented, a new design decision may have unforeseen and undesired consequences.

In the cargo routing application example, the removal of the link between the Vehicle component and its adjacent connector in the descriptive architecture would have resulted in architectural erosion had it not been justified and had the prescriptive architecture not been properly updated. The removal of this link introduces the potential danger that the vehicles controlled by the system are unable to synchronize properly with the rest of the system and deliver their cargo on time. In this case, the system developers realized that Vehicles do not need to keep track of time internally so long as the Cargo Router component gives them appropriate instructions. They discussed this observation with the architects, who agreed and updated the prescriptive architecture accordingly.

Both architectural drift and architectural erosion can be dangerous and expensive, and should be avoided. This requires a certain discipline on the part of architects and developers, but in the long run it saves effort and money, while helping to preserve the important system properties.

Architectural Perspectives

The notion of an architectural perspective is to highlight some aspects of an architecture while eliding others.

> ***Definition.*** **An *architectural perspective* is a nonempty set of types of architectural design decisions.**

As the preceding discussion indicated, software architectures encompass decisions made by a variety of stakeholders at varying levels of detail and abstraction. The purpose of an architectural perspective is to direct attention, for example, for purposes of analysis, to a subset of the decisions. Figure 3-1 and Figure 3-2, for example, provide only the *structural* perspective of the cargo routing application, and say nothing about its behavior, interactions, rationale, and so on.

Another example perspective is *deployment*. A software system cannot fulfill its purpose until it is deployed, that is, until its executable modules are physically placed on the hardware devices on which they are intended to run. The deployment perspective of an architecture can be critical in assessing whether the system will be able to satisfy its requirements. For example, placing many large components on a small device with limited memory and CPU power, or transferring high volumes of data over a network link with low bandwidth will negatively impact the system, much like incorrectly implementing its functionality will. An example deployment perspective on the cargo routing application's architecture is shown in Figure 3-3.

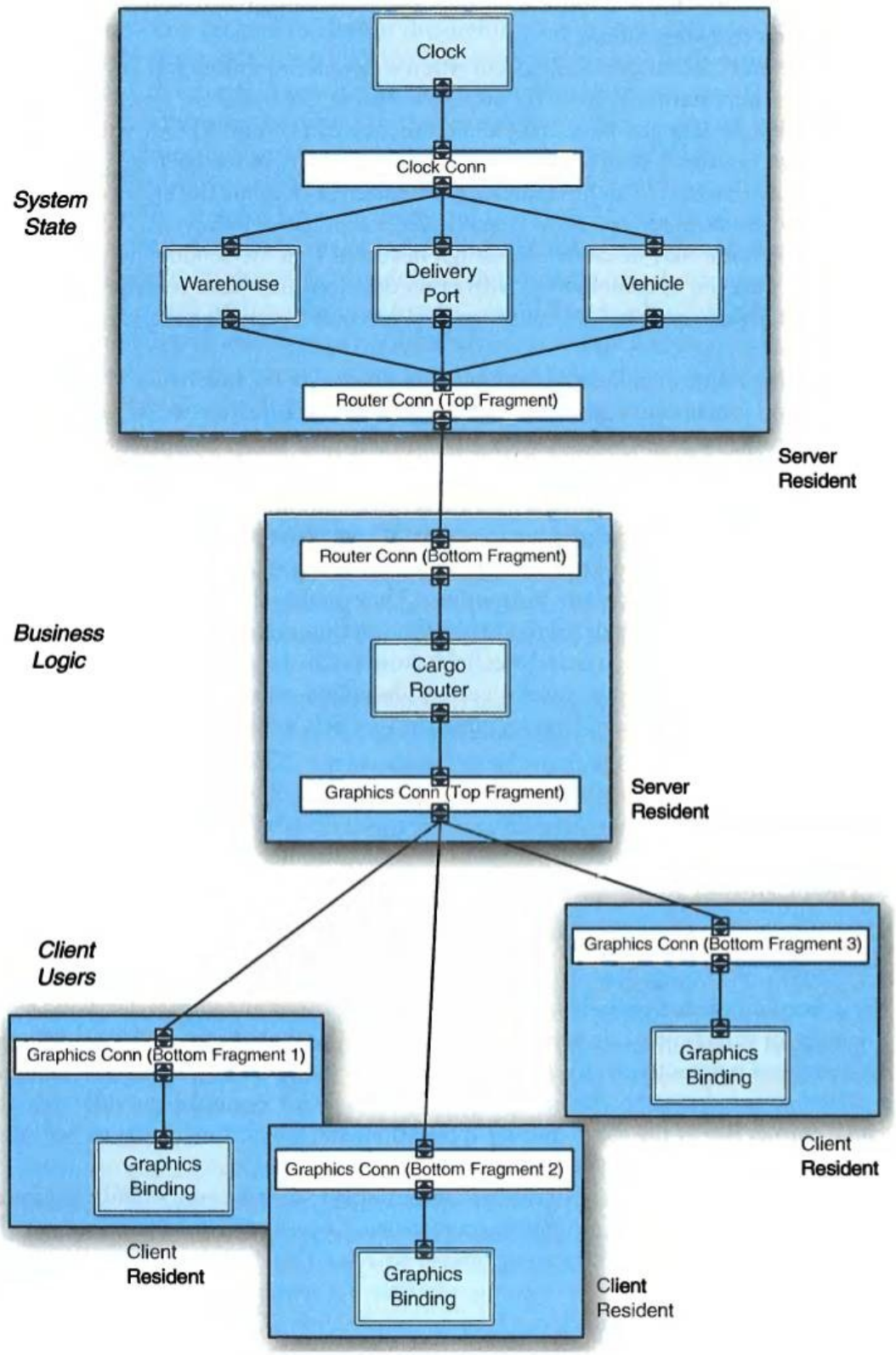


Figure 3-3. *A deployment view of the cargo routing application's architecture distributed across five devices. It is assumed that the Warehouse, Delivery Port, and Vehicle components interact with the corresponding physical objects in order to be able to maintain their state in real-time. However, those interactions are not depicted here.*

To further illustrate, recall that architecture has a temporal dimension, that is, that a system's architecture will likely change over time. However, at any given point in time, the system has only *one* architecture. That architecture may be thought of, modeled, visualized, and discussed from different perspectives. For example, the perspective of the system's modules and their interconnections that is unencumbered with any aspects of the hardware on which the system will run might be referred to as the *structural* view; if we were talking about the prescriptive architecture of a system, then we would say that this is the *structural view of the prescriptive architecture*; if, on the other hand, we were to postulate the hardware topology on which this architecture may (or should) run once implemented, then we would be talking about the *deployment view of the prescriptive architecture*; if we are discussing the distribution of implemented system modules on the different hardware hosts, we would say that this is the *deployment view of the descriptive architecture*; and so on.

Architectural perspectives and views will be discussed in more detail in Chapters 6 and 7. We continue the discussion below with definitions of key architectural building blocks and several other key concepts derived from the concept of software architecture.

Other Definitions of Software Architecture

Dewayne Perry and Alexander Wolf (Perry and Wolf 1992) provide a useful characterization of software architecture as a triple:

Definition. Architecture = {*elements, form, rationale*}

In other words, the architecture defines the system's key elements, and their relationships to each other and to their environment. Furthermore, the architecture reflects the rationale behind the system's structure, functionality, interactions, and resulting properties.

Elements capture the system's building blocks, and help to answer the What questions about the architecture. The questions you may ask about the architecture's elements include the following: What are the system's building blocks? What is a given element's primary purpose in the system? What system services does each element (help to) provide?

Perry and Wolf identify three types of building blocks:

- Processing elements
- Data elements
- Connecting elements

These three types are usually consolidated into two major architectural concepts, components and connectors, which will be discussed below. The one notable exception is the REST architectural style (introduced in Chapter 1 and further discussed in Chapter 11). REST not only gives data elements first-class status, they surpass in importance both processing and connecting elements.

The form captures the way in which the system elements are organized in the architecture. Form represents the structure of individual architectural elements, the manner in which they are composed in the system (that is, the architecture's configuration or topology), the characteristics of their interaction, as well as their relationship to their operating environment (for example, deployment of specific software elements onto specific hardware hosts). The form helps to answer the How questions about the architecture, which may include the following: How is the overall architecture organized? How are the elements composed to accomplish the system's key tasks? How are the elements distributed over a network?

Finally, rationale represents the system designers' intent, assumptions, subtle choices, external (perhaps nontechnical) constraints, selected design patterns and styles, and any other

information that may not be obvious or easily derivable from the architecture. Rationale helps to answer the Why questions about the architecture. These questions may include the following: Why are particular elements used? Why are they combined in a particular way? Why is the system distributed in the given manner?

Although in their seminal paper Perry and Wolf acknowledge the key role of software architecture in a system's evolution, note that their definition does not explicitly capture evolution. However, their definition does suggest that capturing the design rationale is a prerequisite to successfully making any subsequent changes to the system's architecture.

Another useful definition of architecture, which explicitly addresses system evolution, is that provided by the *ANSI/IEEE Standard 1471-2000, Recommended Practice for Architectural Description of Software-Intensive Systems*:

> ***Definition.*** **Architecture is the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution.**

Note that this definition does not specifically refer to software, suggesting that the architecture of a software system is fundamentally similar to architectures of other types of complex systems.

Another interesting, and frequently cited definition of software architecture is attributed to Chris Verhoef (Klusener et al. 2005):

> ***Definition.*** **The software architecture of deployed software is determined by those aspects that are the hardest to change.**

This is a different perspective on how to define software architecture: The definition tells us what *effect* architecture will have on a software system in terms of that system's stability and modifiability. However, the definition does not say what architecture *is*. Furthermore, it is a fair question to ask whether just because something is principal it must be the most difficult to change. In fact, in many cases, architectural design decisions will be explicit enablers of system modification. Examples of this will be provided throughout the book, and in particular in the discussion of architecture-driven software system adaptation discussed in Chapter 14.

3.1.2 Component

The decisions comprising a software system's architecture encompass a rich interplay and composition of many different elements. These elements address key system concerns, including:

- Processing, which may also be referred to as functionality or behavior.
- State, which may also be referred to as information or data.
- Interaction, which may also be referred to as interconnection, communication, coordination, or mediation.

In this section, we will address architectural elements dealing with the first two concerns—processing and data; interaction is covered in the next section.

Elements that encapsulate processing and data in a system's architecture are referred to as *software components*.

***Definition.* A *software component* is an architectural entity that (1) encapsulates a subset of the system's functionality and/or data, (2) restricts access to that subset via an explicitly defined interface, and (3) has explicitly defined dependencies on its required execution context.**

Put another way, a software component is a locus of computation and state in a system (Shaw et al. 1995). A component can be as simple as a single operation or as complex as an entire system, depending on the architecture, the perspective taken by the designers, and the needs of the given system. The key aspect of any component is that it can be "seen" by its users, whether human or software, from the outside only, and only via the interface it (or, rather, its developer) has chosen to make public. Otherwise it appears as a "black box." Software components are thus embodiments of the software engineering principles of *encapsulation*, *abstraction*, and *modularity*. In turn, this has a number of positive implications on a component's composability, reusability, and evolvability.

Another critical facet of software components that makes them usable and reusable across applications is their explicit treatment of the execution context that a component assumes and on which it depends. The extent of the context captured by a component can include:

- The component's *required* interface, that is, the interface to services provided by other components in a system on which this component depends for its ability to perform its operations.
- The availability of specific resources, such as data files or directories, on which the component relies.
- The required system software, such as programming language run time environments, middleware platforms, operating systems, network protocols, device drivers, and so on.
- The hardware configurations needed to execute the component.

Another aspect of a software component, and one that helps to distinguish it further from the connectors discussed below, is a component's relationship to the specific application to which it belongs. Components are often targeted at the processing and data capture needs of a particular application; that is, they are said to be *application-specific*. For example, Vehicle and Warehouse in the cargo routing system are application-specific components: While they may be useful in other similar systems, they were specifically designed and implemented to address the needs of that application.

This need not be always the case, however. Sometimes components are designed to address the needs of multiple applications within a particular class of applications or problem domain. For example, Web servers are an integral part of any Web-based system; one will probably download, install, and configure an existing Web server rather than develop one's own. Another example involves components such as CTunerDriver, CFrontEnd, and CBackEnd, introduced in the product-line discussion in Chapter 1, Figure 1-6, which are intended to be reused across different systems within the consumer electronics domain.

Finally, certain software components are utilities that are needed and can be reused across numerous applications, without regard for the specific application characteristics or domain. Common examples of reusable utility components are math libraries and GUI toolkits, such as Java's Swing toolkit. Another example of arbitrarily reusable components includes common off-the-shelf applications such as word processors, spreadsheets, or

drawing packages. While they usually provide a large superset of any one system's particular needs, architects may choose to integrate them rather than reimplement the exact needed functionality. As the reader will recall, this is the very reason why a system's prescriptive and descriptive architectures need not be identical.

Other Definitions of Software Component

Clemens Szyperski provides another, widely cited definition of software component (Szyperski 1997). He approaches components from a somewhat different perspective:

> *Definition.* **A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.**

This definition does not tell us what a component is, but rather how it is to be structured and used, both by software developers (by composing the component into a system and deploying it) and by other components (by interacting with it through explicit interfaces). This definition also reflects the vision that, once engineered, the component's interior becomes invisible to the outside world. At the same time, the definition does not capture the role a component plays in a system, so that the same definition could, in fact, be used to define a software connector. Below we will define what a software connector is, and how it fundamentally differs from a component.

Szyperski's definition is similar to several other definitions of software components [for example, see Heineman and Councill's book (Heineman and Councill 2001)] in that it does not separate processing from data components as suggested by Perry and Wolf. It is possible that this failure to separate data from computation in software systems' architectures is a side effect of the popularity of object-orientation and particularly object-oriented languages, in which all system elements are treated as objects, regardless of their purpose. One of the primary goals of software architectures is to illuminate and improve understanding of software systems, however, and it may be argued that different concerns (in this case, processing components and data components) should be treated separately. Even though our definition does identify the dual purpose of software components, we should point out that we have rarely found the two addressed separately (the REST architectural style being one notable exception); instead, most frequently a single component will perform a portion of the system's functionality and maintain a part of its state. The distinction will, then, be made according to the component's primary purpose in the system.

3.1.3 Connector

Components are in charge of *processing* or *data*, or both simultaneously. Another fundamental aspect of software systems is *interaction* among the system's building blocks. Many modern systems are built from large numbers of complex components, distributed across multiple, possibly mobile hosts, and dynamically updated over long time periods. In such systems, ensuring appropriate interactions among the components may become even more important and challenging to developers than the functionality of the individual components. In other words, the interactions in a system become a principal (that is, architectural) concern. Software connectors are the architectural abstraction tasked with managing component interactions.

> *Definition.* **A *software connector* is an architectural element tasked with effecting and regulating interactions among components.**

In traditional desktop software systems, connectors usually have manifested themselves as simple procedure calls or shared data accesses, and have typically been treated as ephemeral or invisible in terms of architecture. This is emblematic of boxes-and-lines diagrams, where the boxes, that is, components, dominate, while connectors are relegated to a minor role and accordingly are represented as lines without identity or any unique or important properties. Furthermore, these simple connectors are usually restricted to enabling the interaction of pairs of components. However, as software systems have become more complex, so have connectors, with their own separate identities, roles, and bodies of implementation-level code, as well as ability to simultaneously service many different components.

Connectors are such a critical, rich, and yet largely underappreciated element of software architectures that we have decided to dedicate Chapter 5 to them. Here, we just briefly illustrate some of the connectors with which the reader may be familiar.

The simplest and most widely used type of connector is *procedure call*. Procedure calls are directly implemented in programming languages, where they typically enable synchronous exchange of data and control between pairs of components: The invoking component (the caller) passes the thread of control, as well as data in the form of invocation parameters, to the invoked component (the callee); after it completes the requested operation, the callee returns the control, as well as any results of the operation, to the caller.

Another very common connector type is *shared data access*. This connector type is manifested in software systems in the form of nonlocal variables or shared memory. Connectors of this type allow multiple software components to interact by reading from and writing to the shared facilities. The interaction is distributed in time, that is, it is asynchronous: The writers need not have any temporal dependencies or place any temporal constraints on the readers and vice versa.

An important class of connectors in modern software systems is *distribution* connectors. These connectors typically encapsulate network library application programming interfaces (APIs) to enable components in a distributed system to interact. A distribution connector is usually coupled with a more basic connector to insulate the interacting components from the system distribution details. Thus, for example, remote procedure call (RPC) connectors couple distribution support with procedure calls.

Many software systems are constructed from preexisting components, which may not have been tailor-made for the given system. In such cases, the components may need help with integrating and interacting with one another. *Adaptor* connectors are employed to this end. Depending on their characteristics and the context within which they are used, wrappers and glue code are two common kinds of adaptor connectors with which the reader may be familiar.

Note that while components mostly provide application-specific services, connectors are typically application-independent. We can discuss the characteristics of a procedure call, distributor, adaptor, and so forth independently of the components they service. Notions that have entered our collective dialect, such as "publish-subscribe," "asynchronous event notification," and "remote procedure call," have associated meanings and characteristics that are largely independent of the context within which they are used. Such connectors can be built without a specific purpose in mind, and then used in applications repeatedly (possibly after some customization).

3.1.4 Configuration

Components and connectors are composed in a specific way in a given system's architecture to accomplish that system's objective. That composition represents the system's configuration, also referred to as topology. We define configurations as follows.

> ***Definition.*** **An *architectural configuration* is a set of specific associations between the components and connectors of a software system's architecture.**

A configuration may be represented as a graph wherein nodes represent components and connectors, and whose edges represent their associations (topology or interconnectivity).

As an example, Figure 3-1 shows the architectural configuration of the example cargo routing system. In the figure, Delivery Port and Cargo Router are examples of components, while Router Conn is a connector between them. The configuration shown in the diagram implies that it is possible for these two components to interact with one another; that is, that there is a possible interaction *path* between the components. However, the displayed information does not guarantee their actual ability to interact. In addition to the appropriate connectivity, the components must have compatible interfaces; note that interfaces are not shown in the diagram in Figure 3-1. Incompatible interfaces are one source of *architectural mismatch*, which we further discuss below.

3.1.5 Architectural Style

As software engineers have built many different systems across a multitude of application domains they have observed that, under given circumstances, certain design choices regularly result in solutions with superior properties. Compared to other possible alternatives, these solutions are more elegant, effective, efficient, dependable, evolvable, and scalable. For example, it has been observed that the following set of design choices ensures effective provision of services to multiple system users in a distributed setting. These choices are intentionally stated here informally, to illustrate the point.

1. Physically separate the software components used to request services from the components that provide the needed services, to allow for proper distribution and scaling up, both in the numbers of service providers and service requesters.
2. Make the service providers unaware of the requesters' identity to allow the providers to service transparently many, possibly changing requesters.
3. Insulate the requesters from one another to allow for their independent addition, removal, and modification. Make the requesters dependent only on the service providers.
4. Allow for multiple service providers to emerge dynamically to off-load the existing providers should the demand for services increase above a given threshold.

Note that the above list does not comprise architectural design decisions for a particular system (or class of systems). Rather, these architectural decisions are applicable to any system that shares the distributed service provision context. These decisions do not specify the components (or component types), the interaction mechanisms among those components, or their specific configuration. The architect will have to elaborate further on these

decisions and turn them into application-specific architectural decisions when designing a system. These higher-level architectural decisions do, however, state the rationale that underlies them, so that the architect can justify choosing them for his system.

> ***Definition.*** **An *architectural style* is a named collection of architectural design decisions that (1) are applicable in a given development context, (2) constrain architectural design decisions that are specific to a particular system within that context, and (3) elicit beneficial qualities in each resulting system.**

The above example is an informal and partial specification of the popular client-server style. Many other styles are in use regularly in software systems, including REST and pipe-and-filter styles introduced in Chapter 1 and others that will be revisited throughout the book. Chapters 4 and 11 provide detailed discussions of architectural styles.

3.1.6 Architectural Pattern

Architectural styles provide general design decisions that both constrain and may need to be refined into additional, usually more specific design decisions in order to be applied to a system. In contrast, an architectural pattern provides a set of specific design decisions that have been identified as effective for organizing certain classes of software systems or, more typically, specific subsystems. These design decisions can be thought of as configurable in that they need to be instantiated with the components and connectors particular to an application. In other words:

> ***Definition.*** **An *architectural pattern* is a named collection of architectural design decisions that are applicable to a recurring design problem, parameterized to account for different software development contexts in which that problem appears.**

On the surface, this definition is reminiscent of the definition of architectural style. In fact, the two notions are similar and it is not always possible to identify a crisp boundary between them. However, in general styles and patterns differ in at least three important ways:

1. ***Scope***: An architectural style applies to a development *context* (for example, "highly distributed systems" or "GUI-intensive") while an architectural pattern applies to a specific design *problem* (for example, "The system's state must be presented in multiple ways" or "The system's business logic must be separated from data management"). A problem is significantly more concrete than a context. Put more concisely, architectural styles are *strategic* while patterns are *tactical* design tools.
2. ***Abstraction***: A style helps to constrain the architectural design decisions one makes about a system. However, styles require human interpretation in order to relate design guidelines captured to reflect the general characteristics of the development context to the design problems pertaining to the specific system at hand. By themselves, styles are too abstract to yield a concrete system design. In contrast, patterns are parameterized architectural fragments that can be thought of as concrete pieces of a design.
3. ***Relationship***: Patterns may not be usable "as is" in that they are parameterized to account for the different contexts in which a given problem appears. This means that a single pattern could be applied to systems designed according to the

Figure 3-4. *Graphical view of the three-tier system architectural pattern.*

guidelines of multiple styles. Conversely, a system designed according to the rules of a single style may involve the use of multiple patterns.

An example pattern that is used widely in modern distributed systems is the *three-tier system* pattern. The three-tier pattern is applicable to many types of systems in which distributed users need to process, store, and retrieve significant amounts of data, such as science (for example, cancer research, astronomy, geology, weather), banking, electronic commerce, and reservation systems across widely different domains (for example, travel, entertainment, medical care). Figure 3-4 shows an informal graphical view of this pattern.

In this pattern, the first tier, often referred to as the *front* or *client* tier, contains the functionality needed to access the system's services, typically by a human user. The front tier would thus contain the system's GUI, and possibly be able to cache data and perform some minor local processing. It is assumed that the front tier is deployed on a standard host (for example, a desktop PC), possibly with limited computing and storage capacity. It is also assumed that the front tier will be replicated to allow independent, simultaneous access to multiple users.

The second tier, also referred to as the *middle*, *application*, or *business logic* tier contains the application's major functionality. The middle tier is in charge of all significant processing, servicing requests from the front tier and accessing and processing data from the back tier. It is assumed that the middle tier will be deployed on a set of powerful and capacious server hosts. However, the number of middle-tier hosts is usually significantly smaller than the number of front-tier hosts.

Finally, the third tier, also referred to as the *back*, *back-end*, or *data* tier contains the application's data access and storage functionality. Typically, this tier will host a powerful database that is capable of servicing many data access requests in parallel.

The interactions among the tiers in principle obey the request-reply paradigm. At the same time, the pattern does not prescribe those interactions further. For example, it may be possible to design and implement a three-tier-compliant system to strictly adhere to synchronous, request-triggered, single-request–single-reply interaction; alternatively, it may be possible to allow multiple requests to result in a single reply, multiple replies to be issued in response to a single request, periodic updates to be issued from the back and middle tiers to the front tier, and so forth.

The three-tier architectural pattern can be used to determine the architecture of a specific distributed software system. The things the architect needs to specify are:

1. Which application-specific user interface, processing, and data access and storage facilities are needed and how they should be organized within each tier.
2. Which mechanisms should be used to enable interaction across the tiers.

Use of architectural styles to solve the same problem requires, in contrast, more attention from the system's architect, and provides less direct support. In fact, the three-tier architectural pattern can be thought of as two specific architectures that are designed according to the client-server style and overlaid on top of each other: The front tier is the client to the middle tier, while the middle tier is the client to the back tier; the middle tier is thus a server in the first client-server architecture and a client in the second. Indeed, systems adhering to the client-server style are sometimes referred to as two-tier systems.

As an example, Figure 3-5 shows architectures of two different three-tier systems. At this point, the reader should be able to identify some architectural traits the two systems have in common, and those that differ. Further discussion and additional examples of architectural patterns can be found in Chapter 4, which also discusses the fuzzy boundary between patterns and styles. While the definitions above distinguish the concepts, in practice the two notions can become blurred.

3.2 MODELS

A software system's architecture is captured in an architectural *model* using a particular modeling *notation*.

> ***Definitions.*** **An architectural *model* is an artifact that captures some or all of the design decisions that comprise a system's architecture. Architectural *modeling* is the reification and documentation of those design decisions.**

A model is a result of the activity of modeling, which constitutes a significant portion of a software architect's responsibilities. One system may have many distinct models associated with it. Models may vary in the amount of detail they capture, the specific architectural perspective they capture (for instance, structural versus behavioral, static versus dynamic, entire system versus a particular component or subsystem), the type of notation they use, and so forth.

> ***Definition.*** **An architectural modeling *notation* is a language or means of capturing design decisions.**

For example, the two diagrams shown in Figure 3-5 represent models captured in two different visual modeling notations. The notations for modeling software architectures are frequently referred to as architecture description languages (ADLs). ADLs can be textual or graphical, informal (such as PowerPoint diagrams), semi-formal, or formal, domain-specific or general-purpose, proprietary or standardized, and so on.

Architectural models are used as the foundation for most other activities in architecture-based software development processes, such as analysis, system implementation, deployment, and dynamic adaptation. Models and modeling are critical facets of software architecture and are discussed in depth in Chapter 6.

3.3 PROCESSES

As discussed at length in Chapter 2, software architecture is not a software engineering life cycle phase that follows requirements elicitation and precedes low-level design and system implementation. Instead, it permeates, is an integral element of, and is continually

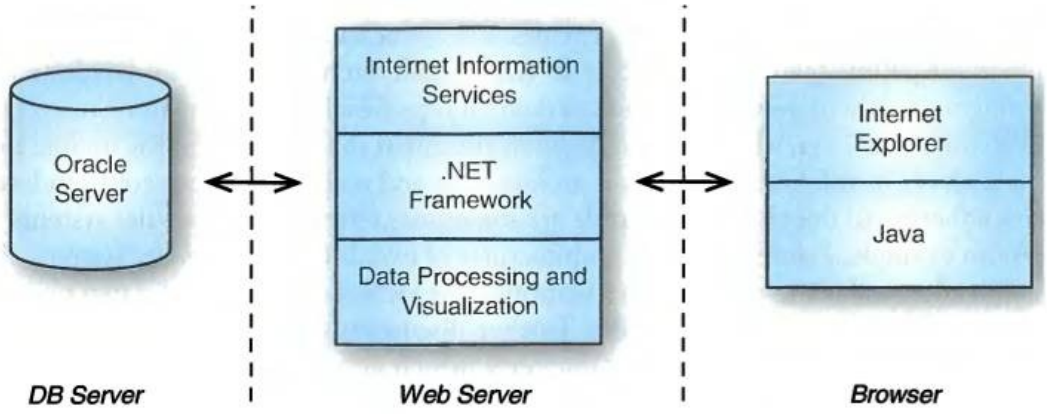
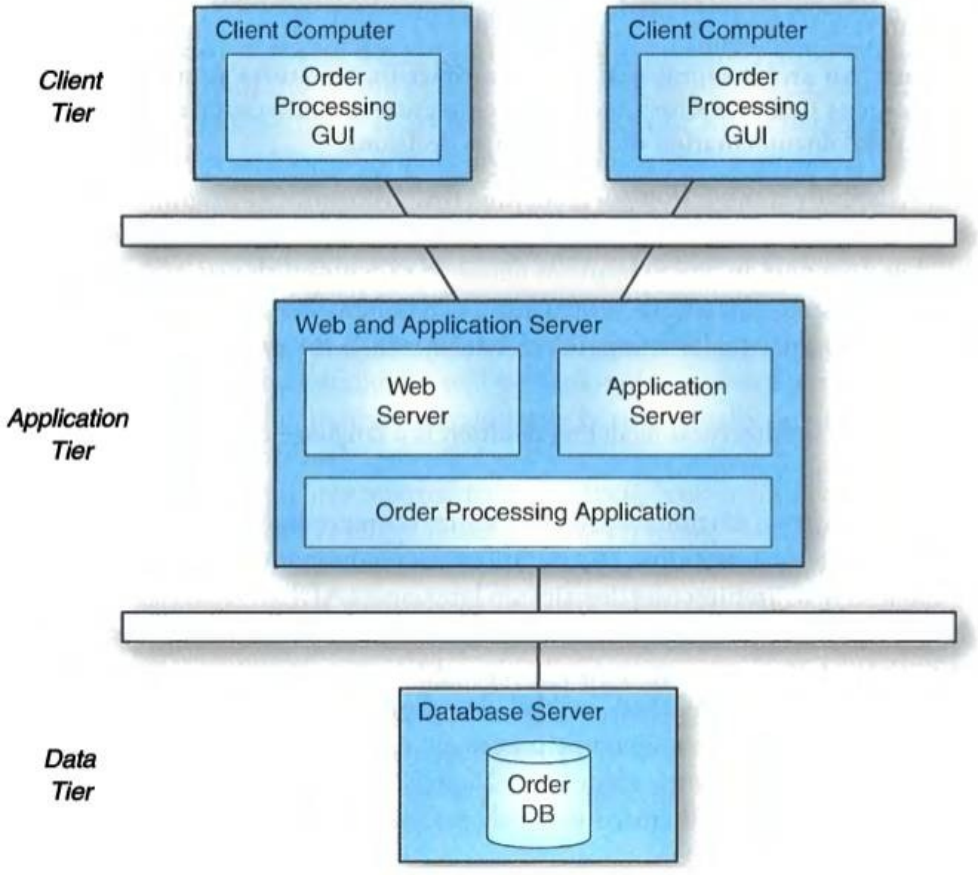


Figure 3-5. *Two example three-tier system architectures. Simply "Googling" the phrase will yield many hits.*



impacted by, all facets of software systems development. In that sense, software architecture helps to anchor the processes associated with different development activities. Each of these activities is covered separately in this book; with one exception, we will only name them here and state in which chapter they are discussed:

- The activity of architectural *design* (Chapter 4)
- Architecture *modeling* (Chapter 6) and *visualization* (Chapter 7)
- Architecture-driven system *analysis* (Chapter 8)
- Architecture-driven system *implementation* (Chapter 9)
- Architecture-driven system *deployment*, run-time *redeployment*, and *mobility* (Chapter 10)
- Architecture-based *design for nonfunctional properties* (Chapter 12), including security and trust (Chapter 13)
- Architectural *adaptation* (Chapter 14)

The one activity on which we will focus in this section is *architectural recovery*, which is again revisited in Chapter 4, in the context of architectural design processes, and in Chapter 8, in the context of architectural analysis.

Recall the above discussion of architectural degradation (that is, architectural drift and/or erosion). If degradation is allowed to occur, a software development organization is likely to be forced to *recover* the system's architecture sooner or later. This happens when, at time *t* during a system's life, changes to the system become too expensive to implement and their effects too unpredictable because the documented prescriptive (that is, as-intended) architecture is so outdated as to be useless or, sometimes even worse, misleading.

> ***Definition. Architectural recovery* is the process of determining a software system's architecture from its implementation artifacts.**

Implementation artifacts can be source code, executable files, Java.class files, and so forth. As an illustration, Figure 3-6 shows the dependencies among the Java objects that implement the cargo routing application introduced in earlier. This diagram has been automatically generated from the application's source code using an off-the-shelf source code analysis tool. At this magnification the figure is used for illustration only; we do not expect the reader to understand its details, other than that the rectangles, which are mostly on the left, represent objects and the lines, extending to the right, represent dependencies between them (for example, Java method calls).

Figure 3-6 reflects many details, including architectural design decisions, low-level design decisions, implementation-level decisions, the Java libraries used by the system, the implementation framework, and so on. From derived artifacts such as this, the system's architecture is recovered by the process of isolating and extracting only the architectural—that is, principal—design decisions.

By its very nature, the process of architectural recovery extracts a system's *descriptive* architecture. That architecture, if complemented with a statement of the architects' original intent, can in principle be used to recover the system's *prescriptive* architecture. However, since the original architects may be unavailable, and their original intent may not have been recorded (or, even if it was recorded originally, may have been repeatedly violated

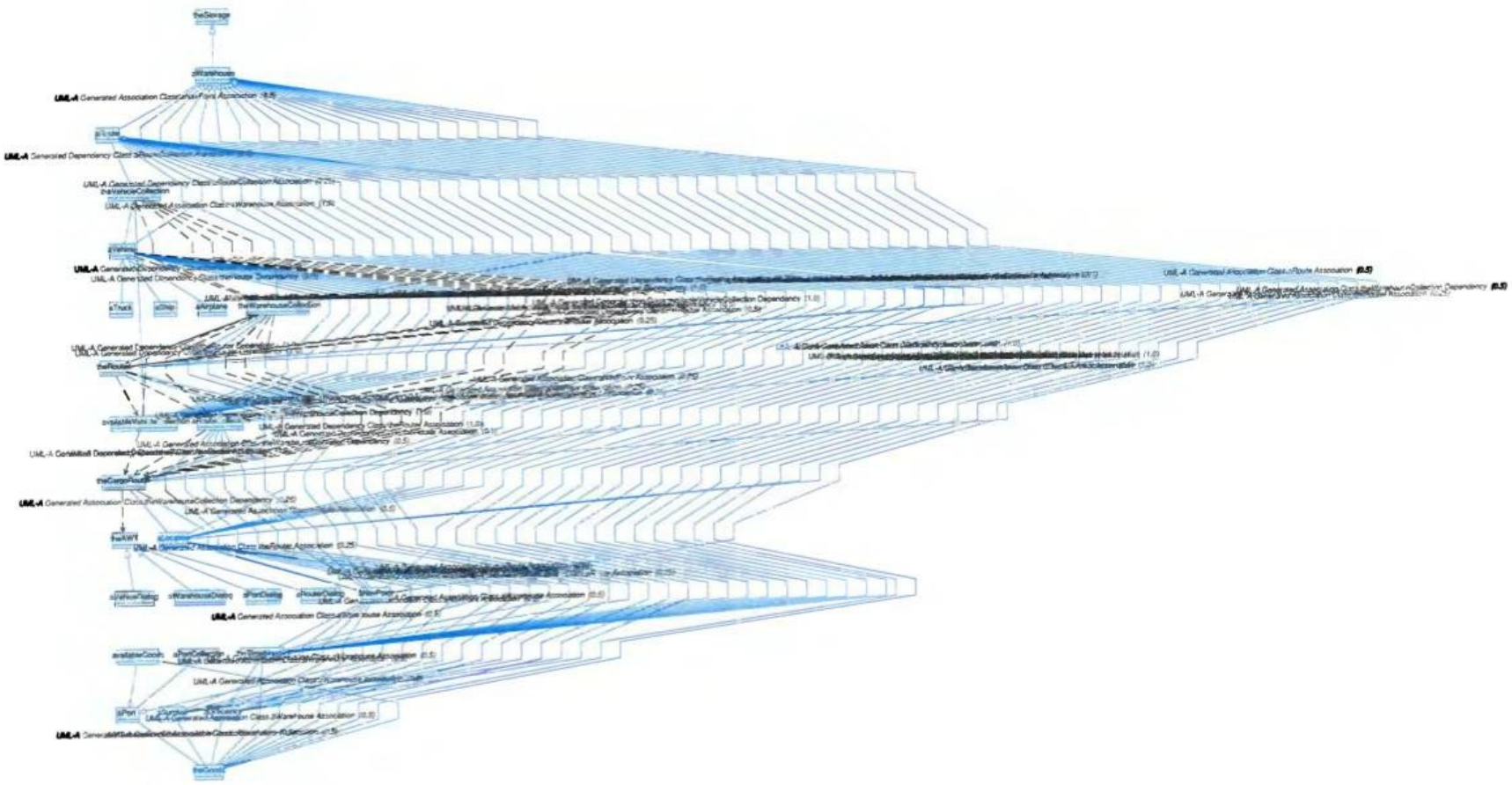


Figure 3-6. *Implementation-level view of the cargo routing application.*

over time), it is often impossible to recover a system's prescriptive architecture. Instead, the recovered descriptive architecture is treated as the closest approximation of the system's prescriptive architecture, and thus the system's architectural evolution clock is effectively reset.

Even though the specifics of the architectural recovery tools and techniques used in practice are beyond the scope of this chapter, the reader should appreciate that recovery is a very time-consuming and complex process. Furthermore, the sheer complexity of most software systems—with their myriad explicit, implicit, intended, and unintended module interdependencies—makes the task of assessing a given implementation's compliance to its purported architecture very difficult. This is why it is critical for software architects and engineers to maintain architectural integrity at *every* step throughout the system's life span. Once the architecture degrades, all subsequent solutions to stem that degradation will be more costly and more error-prone by comparison.

3.4 STAKEHOLDERS

The preceding sections have introduced the reader to the what, the how, and the why of software architecture. This section rounds out the chapter by briefly introducing the who—that is, several key architectural stakeholders.

The software *architect* is one obvious stakeholder. The architect conceives the system's architecture, and then models, assesses, prototypes, and evolves it. Architects maintain a system's conceptual integrity and are thus the system's critical stakeholders.

Software *developers* are the primary consumers of an architect's product (that is, the architecture). They will realize the principal design decisions embodied in the architecture by producing a system implementation.

The role of software *managers* from an architectural standpoint is to provide project oversight and support for the software architects. As will be detailed in Chapter 17, in an organization architects often bear much of the responsibility for a system's success without the accompanying authority. This is why it is critical for architects and managers to work closely together, and for the managers to buy into the key architectural decisions and, if necessary, exert their authority on behalf of the architects. Managers are thus key stakeholders as well.

The bottom-line objective for a given system's *customers*—the ultimate stakeholders—is that a high-quality system satisfying their requirements be delivered on time and within budget. A significant determinant of a project's ability to meet that objective will be the system's architecture. Simply put, an effective architecture will result in a successful project, while an ineffective one will seriously hamper the project's success. A more complete treatment of architectural stakeholders will be provided in Chapter 17.

3.5 END MATTER

Any mature engineering field must be accompanied by a shared, precise understanding of its basic concepts, commonly used models, and processes. This chapter has defined the notions that underlie the field of software architecture. While a less-experienced reader may not yet be able to fully appreciate some of the definitions and accompanying discussion,

in subsequent chapters we return to these concepts as we study them in greater depth. We expect that the reader will also find it useful to return to these concepts and this chapter. In fact, right away, Chapter 4 uses many of the concepts introduced here in the discussion of how software architectures are designed and, in particular, in providing an overview of a large number of commonly used architectural styles. In turn, Chapter 5 will elaborate on the many different classes of software connectors and their properties.

The Business Case

Software architecture is a field of study that is characterized by an unusual diversity of views and understandings of some fundamental concepts. For example, a quick search of the Internet will yield many definitions of architecture. Similarly, there is a diversity of views on the role and importance of connectors, and even whether they deserve a separate treatment from components. The reader will also find arguments (even explicitly embodied in component models such as Microsoft DCOM's component model) that components are exclusively executable entities, which would go against many of the motivations and underpinnings of software architecture.

Simply put, imprecise and inconsistent use of poorly defined—and sometimes undefined—terms is counter to success in a highly competitive environment such as software product development. The ability to build quality products and amass and train a skilled workforce requires precise use of concrete terminology with specific meanings.

A specific instance of this issue would be an organization intending to hire and/or train a software architect. Without a shared technical language, hiring such an individual would be difficult and risky. Without a precise understanding of the foundational concepts in the field of software architecture, training software architects would be expensive and ultimately unproductive.

3.6 REVIEW QUESTIONS

1. Software architects often are asked what the difference is between architecture and design. Given the definition of software architecture provided in this chapter, can you distinguish between the two?
2. What are the key differences between a software component and a software connector?
3. Can connectors simply be treated as special-purpose components? Should they?
4. Should architects and engineers be expected to accept architectural degradation as a fact of life? What, if any, dangers are inherent in doing that?
5. What is the difference between an architectural style and an architectural pattern?
6. Can multiple patterns be used to realize a style?
7. Why should architects concern themselves with the needs of the customers?

3.7 EXERCISES

1. Identify components, connectors, topology in a well-known system's architecture. This can be something Web-based, something described in a publication, or even one of the examples introduced in the book so far, but for which this question has not been answered as part of the chapter.
   a. How can you tell components apart from connectors?
   b. Is the architecture descriptive or prescriptive?
2. The section that defined architectural styles provides a set of architectural guidelines corresponding to client-server systems. Select a distributed application scenario of your choice and show how you would turn these guidelines into specific design decisions. How easy are the guidelines to apply correctly? How easy are they to violate? Try to violate the second guideline and discuss the immediate and potential impact on your architecture.

3. Try to solve the above problem by applying the three-tier system pattern. Compare this solution to the previous one. Discuss which approach was easier to apply and why.

3.8 FURTHER READING

The first explicit treatment of software architecture and the concepts that underlie it was provided by Perry and Wolf (Perry and Wolf 1992). This seminal paper introduced several definitions that have inspired those we have provided in this chapter. Several years later, Shaw and Garlan's book (Shaw and Garlan 1996) provided their collection of definitions, summaries of several industrial and research projects in which the authors were involved, and early architectural insights from those projects. Several subsequent books offered their respective authors' takes on different facets of software architecture: modeling (Hofmeister, Nord, and Soni 1999), evaluation (Clements, Kazman, and Klein 2002), architectural patterns (Buschmann et al. 1996), product lines (Bosch 2000), and component-based system development (Heineman and Council 2001; Szyperski 2002).

A large number of conference and journal articles accompanied these books, and several specialized venues emerged. Initially, the SIGSOFT International Software Architecture Workshop (ISAW) was the primary venue for researchers and practitioners in the field of software architecture to exchange their ideas and develop common understandings. A more narrowly focused workshop series, the Role of Software Architecture for Testing and Analysis (ROSATEA), followed soon thereafter. ISAW was eventually supplanted by the Working IEEE/IFIP Conference on Software Architecture (WICSA). More recently, the European Conference on Software Architecture (ECSA) and the International Conference on the Quality of Software Architectures (QoSA) have emerged. The on-going conference series on Component-Based Software Engineering (CBSE) has naturally had a significant architectural dimension. Finally, the Elsevier Journal on Systems and Software has recently introduced a regular section on software architecture.

While this wealth of books, papers, workshops, conferences, and journals has helped the field of software architecture to mature relatively quickly, it did not result in a convergence of understanding of the principal architectural concepts. One example of this lack of convergence is the use by the software engineering community of a very large and divergent collection of different definitions of software architecture, gathered by the Software Engineering Institute (www.sei.cmu.edu/architecture/definitions.html). While collecting all those definitions may be useful as a sort of historical record, actually relying on all of them is not a hallmark of a mature engineering field. That realization was one of the primary motivators for this book, and this chapter in particular.