

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.

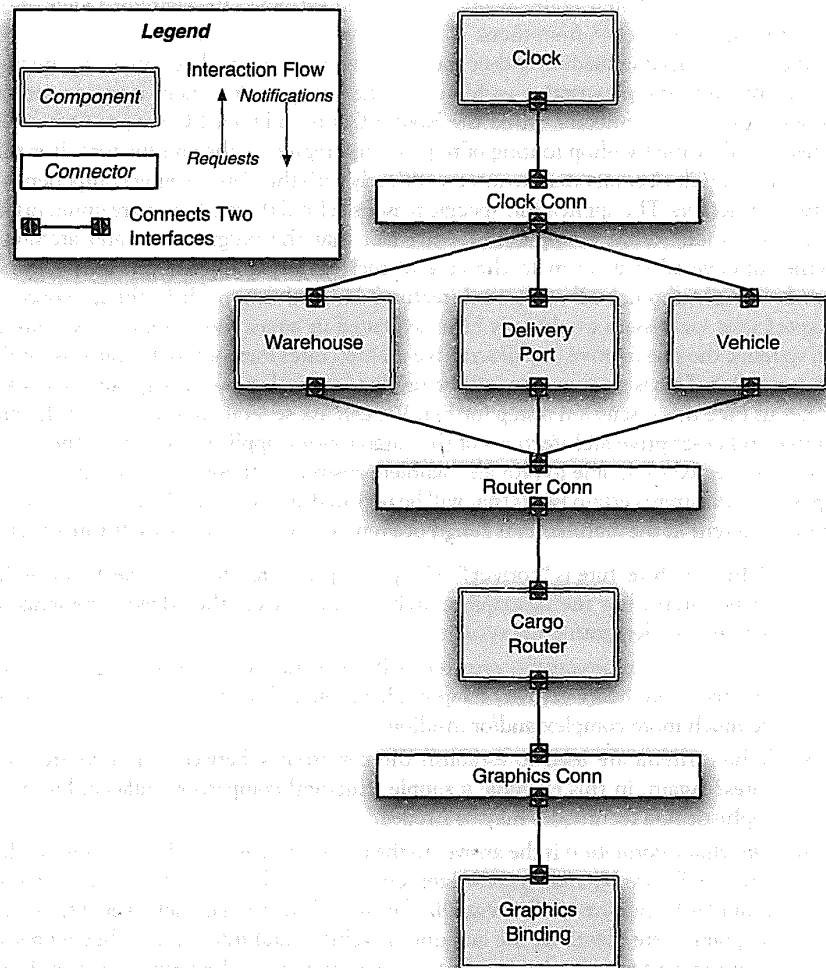
1. Which architecture is "correct?" The prescriptive architecture reflects the architects' intent, but the descriptive architecture reflects the added experience of actually implementing the system.
2. Are the two architectures consistent with one another? In this example, the differences were relatively 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

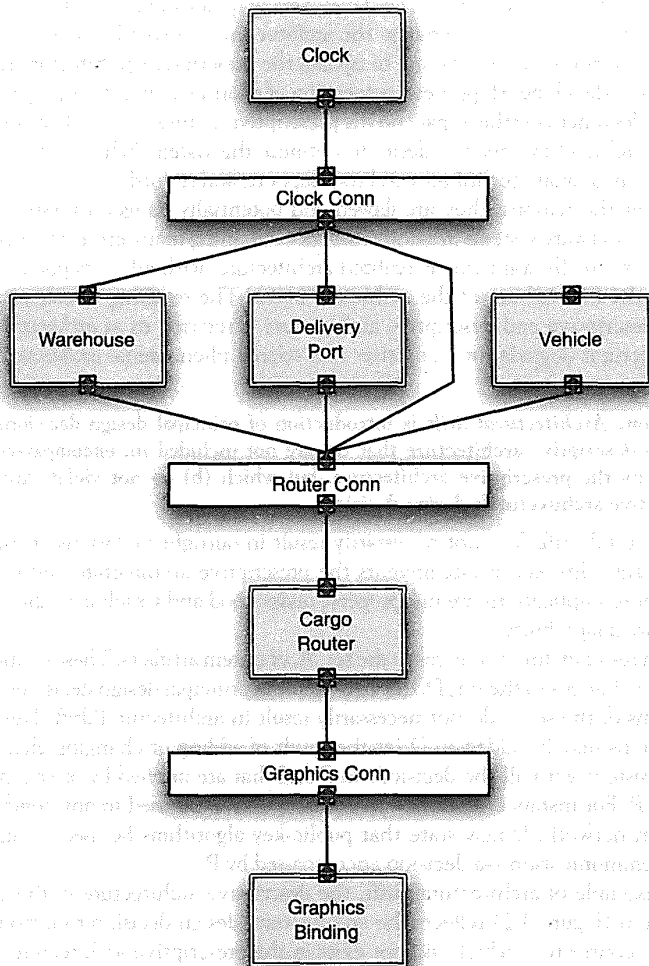


Figure 3-2.
A high-level,
graphical view of
the descriptive
(that is,
as-implemented
or as-realized)
architecture of
the cargo routing
application.

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

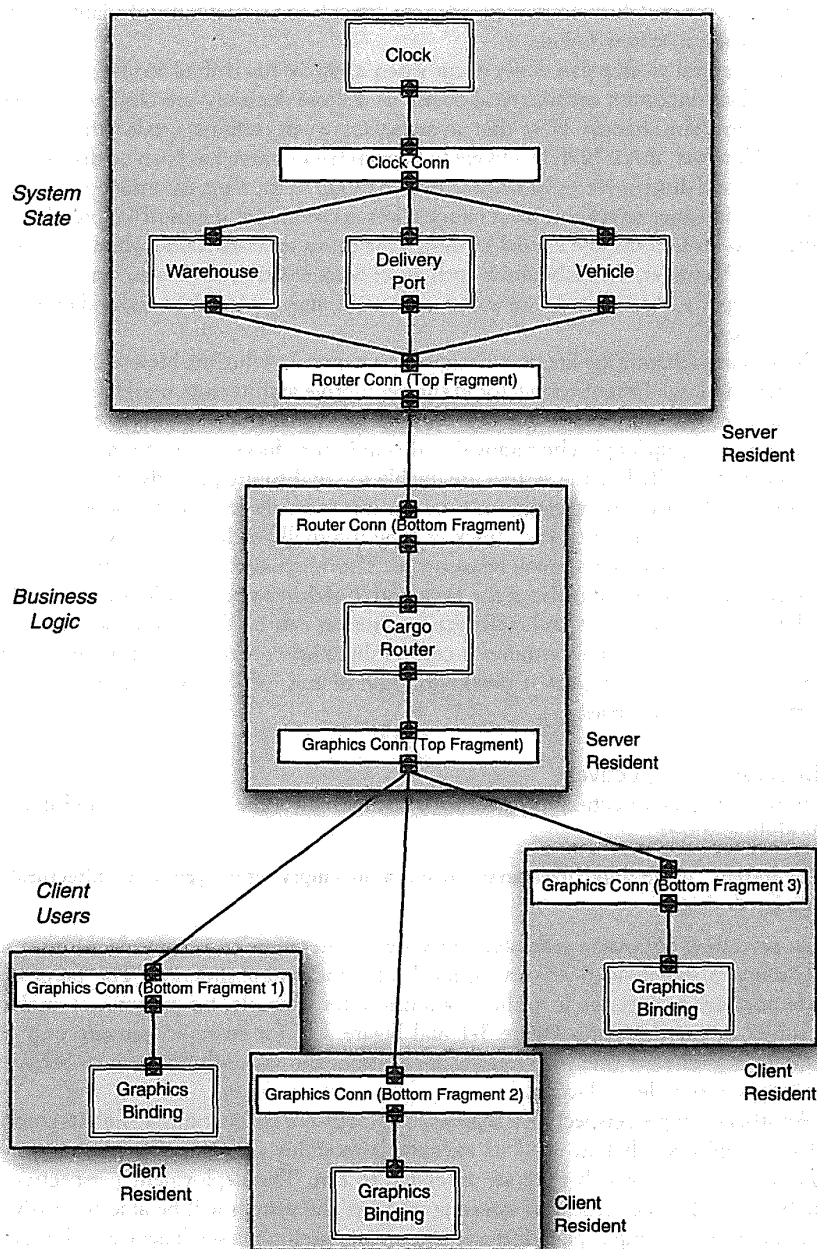
The notion of an architectural perspective is to highlight some aspects of an architecture while eliding others.

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.

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

Other
Definitions
of Software
Architecture

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

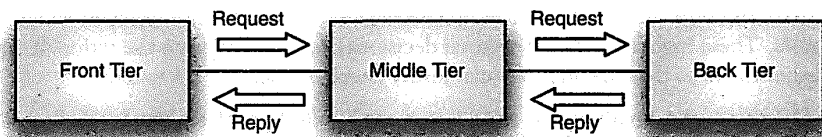
Architectural styles provide general design decisions that both constrain and may need to be refined into additional, usually more specific design decisions in order to be applied to a system. In contrast, an architectural pattern provides a set of specific design decisions that have been identified 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.

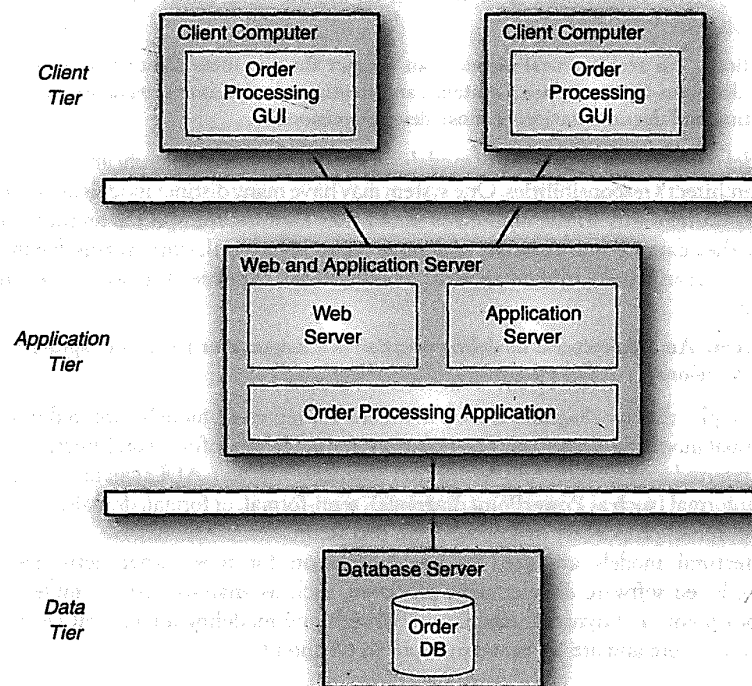
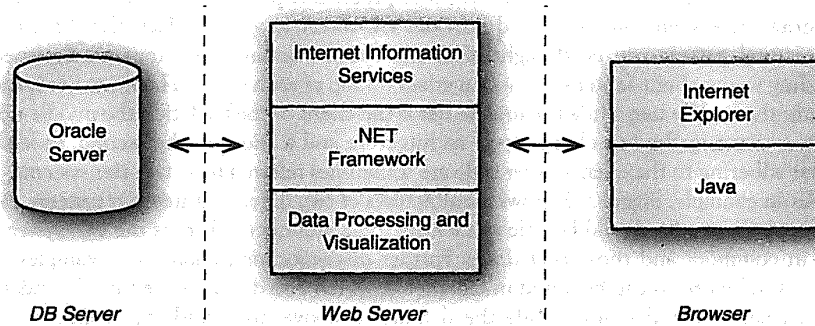
For example, the two diagrams shown in Figure 3-5 represent models captured in two different visual modeling notations. The notations for modeling software architectures are frequently referred to as architecture description languages (ADLs). ADLs can be textual or graphical, informal (such as PowerPoint 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 *redemption*, 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

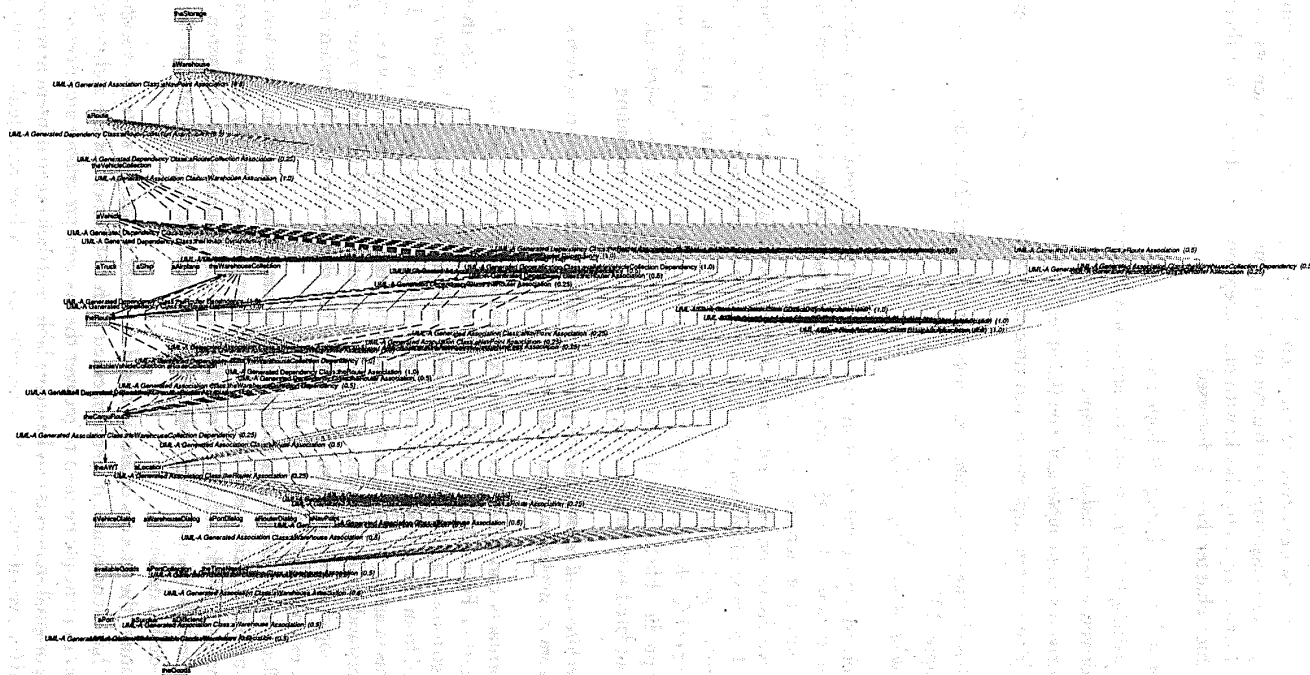


Figure 3-6. Implementation-level view of the cargo routing application.

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.

Designing Architectures

The preceding chapters have laid the foundation for software architecture. Chapter 1 presented motivations for focusing on architectures and the benefits that result. Chapter 2 positioned software architecture with respect to the major activities of software engineering. Chapter 3 took the informal notions of the first two chapters and set them on a firm basis.

So far, however, we have largely neglected the question of *how* architectures are created. Returning to the analogy of building architectures, simply having a set of power tools and a lofty vision for a skyscraper, for example, is not a fully adequate basis for creating a fifty-story building that successfully houses businesses, provides the base for a television broadcast tower, and which effectively integrates with a city's electrical, water, data, and wastewater infrastructures. The small matter of design stands between raw materials and tools and realization of the vision. The design of a skyscraper is the product of a cadre of architects and engineers who labor to meet all the goals for the building while simultaneously satisfying all the constraints placed upon it, and upon them.

But how are designs such as this created? How are engineers and architects taught to approach the problem?

For many serious engineers and building architects, design is viewed as something that cannot be taught as a method. Rather, students are apprenticed: One “sits by the feet of the master” and the hope is that by some mysterious process of osmosis the student eventually acquires some smidgen of the master’s ability. For others, the ability to design is simply something one either is or is not born with. We disagree with both of these views.

First, as human beings we all have the ability to design. We are all innately designers, and in fact engage that ability regularly in our daily lives, from cooking to decorating to painting to writing. It is just how we are created.

Second, design is subject to rigorous, methodical examination, like any other endeavor—there is nothing about it that renders it incapable of study. The outcome

of such study is a set of approaches and techniques for designing that can be described, taught, applied, evaluated, and refined. Examination of existing designs helps identify good strategies for tackling various kinds of problems and helps develop a sense for associated costs and qualities. At a very minimum, students can be taught the use of design tools and simple design methods. Be assured, however, that study of design methods does not imply that there is no place, or need, for creativity. Far from it; one outcome of disciplined study of design is a focusing of design effort and creativity on those aspects of problems that demand it.

The goal of this chapter, therefore, is to set forth a variety of approaches for helping the student learn how to design software systems. We begin with considering some basics of design processes and then move on to foundational conceptual tools; easy to understand, but perhaps hard to apply. To provide a more concrete grip on the task, we then examine in detail a variety of techniques that exploit the most useful software design tool of all, that of refined experience. This discussion briefly reviews the subject of domain-specific software architectures, as introduced in Chapters 1 and 2, putting this particular technique in context with others. The chapter then introduces a wide gamut of architectural patterns and styles that capture and exploit the experience of preceding designs. The section concludes with a discussion of design recovery, the process of making the design of an existing system explicit in such a way that it becomes useful in the tasks of extending or modifying that system or using that architecture as a basis for a new system. This, of course, reinforces one of this text's primary themes: Architecture is the centerpiece of system development and evolution. While designing the architecture is the focus of this chapter, extending and maintaining the architecture is something that occurs throughout a system's lifespan.

When refined experience proves inadequate or is unavailable for solving a new design problem, we describe a variety of techniques for dealing with such design challenges. The techniques presented draw from industrial product design, building architectures, and other sources, as well as from software engineering.

The chapter concludes by briefly returning to the topic of design processes, tying the whole discussion together. Since this is a long chapter, with a multitude of techniques and approaches described, this section helps to put all the details into perspective.

A designer may come to a problem from many starting points—with no prior relevant experience, with much directly relevant experience, or with a goal of simply recovering a design. This chapter provides insights for each.

4 Designing Architectures**4.1 The Design Process****4.2 Architectural Conception****4.2.1 Fundamental Conceptual Tools****4.2.2 The Grand Tool: Refined Experience****4.3 Refined Experience in Action: Styles and Architectural Patterns****4.3.1 Domain Specific Software Architectures****4.3.2 Architectural Patterns****4.3.3 Introduction to Styles****4.3.4 Simple Styles****4.3.5 More Complex Styles****4.3.6 Discussion: Patterns and Styles****4.3.7 Design Recovery****4.4 Architectural Conception In Absence of Experience: Unprecedented Design****4.5 Putting it All Together: Design Processes Revisited****4.5.1 Insights from Requirements****4.5.2 Insights from Implementation****4.6 End Matter****4.7 Review Questions****4.8 Exercises****4.9 Further Reading****Outline of Chapter 4****4.1 THE DESIGN PROCESS**

The typical assumption of software design is that the process can proceed in the general manner of architectural design or engineering design, which can be summarized (Jones 1970) as consisting of the following four stages.

1. Feasibility stage: Identifying a set of feasible concepts
2. Preliminary design stage: Selection and development of the best concept
3. Detailed design stage: Development of engineering descriptions of the concept
4. Planning stage: Evaluating and altering the concept to suit the requirements of production, distribution, consumption and product retirement

Of these processes, Jones comments, "... designing begins (stage 1) by the taking-in of information. From this a set of alternative arrangements for the design as a whole is quickly derived. Stage 2 is to select one of these alternatives for further development. When this design has reached the point of satisfying the chief designer the work is split up for detailed design by many people working in parallel (stages 3 and 4)" [(Jones 1970), p. 24, emphasis added]. Jones here is echoing Vitruvius in stating that good architecture depends on fitness and arrangement.

The reader will quickly recognize that this approach to design is completely consistent with common, broader notions of software engineering processes, such as the waterfall and spiral models of development. It is similarly consistent with more specific approaches to design, such as the Unified Process, Jackson System Development, and Microsoft's Code Complete. Such universal use indicates that it is a very effective strategy in many situations.

The viewpoint represented by this process so pervades the software development world that it is hard to realize that the use of this process represents a choice—that, indeed, other approaches are possible. This is an important recognition because the standard approach does not always work. Even in those situations where it does work, it may not yield the best results. At a minimum, designers must be aware of some of the conditions under which the traditional approach is likely to be insufficient.

We will consider first the conditions under which this process may not work:

- More than anything else, success of the standard process is predicated on the success of the first step: identifying a set of feasible concepts for the overall structure of the system. If the designer is unable to produce such a set, progress stops.
- While not intrinsically part of the standard approach, in history and in typical current practice, the first step (or two) is performed by an individual. As problems and products increase in size and complexity, the probability that any one individual can successfully perform the first steps decreases. Merely saying that those steps should be performed by a design team does not resolve the issue: the existence of a team generates new problems. These problems can be resolved, or at least mitigated, but honesty demands that their existence be explicitly acknowledged.
- In a similar vein, the standard approach does not directly address the situation where system design is at stake, that is, when relationships between a set of products is at issue. The issue is again complexity. As complexity of the problem increases, the likelihood of an individual successfully performing the first step of the typical process decreases.

The common thread in these remarks is complexity of the application being designed. As complexity increases or, as we shall see later, the experience of the designer is not sufficient for the challenge, alternative approaches to the design process must be adopted. As Jones puts it, "... the principle of deciding the form of the whole before the details have been explored outside the mind of the chief designer does not work in novel situations for which the necessary experience cannot be contained within the mind of one person."

The good news is that there are many design strategies to choose from—essentially they are process models applied at the level of design:

- Standard: The linear model described above.
- Cyclic: As problems or infeasible approaches are identified in stages 2 to 4 of the standard model, the process reverts to an earlier stage.
- Parallel: After stage 1 of the standard model, independent alternatives are explored in parallel; at suitable times selection is made between the identified viable alternatives. A related strategy is to parallelize at stage 1: Multiple independent attempts at design are initiated, with no dependencies between them.
- Adaptive ("lay tracks as you go"): The design strategy to follow in the next stage of the design activity is decided at the end of a given stage, based upon insights gained during that stage.
- Incremental: Design at each stage of development is treated as a task of incrementally improving whatever design or previous product exists after a preceding stage.

Sensible management of the design process also yields another "strategy": The process is observed as it proceeds and the methods and approach used are modified as needed to focus on the best strategy and to avoid unpromising developments.

Now that we have identified conditions under which the standard approach to design does not work as well as alternative approaches, we now focus on the critical first step of design: identifying a candidate set of feasible concepts.

4.2 ARCHITECTURAL CONCEPTION

The standard approach to architectural and engineering design says that the first major step consists of identifying a feasible set of “alternative arrangements for the design as a whole,” choosing one of those arrangements, and then proceeding to refine and elaborate it. What the standard approach does not tell, however, is *how* to identify that set of viable arrangements—or even one viable arrangement.

A facile answer to the question is to simply state, “Apply the fundamental design tools of software engineering: abstraction and modularity.” While the utility, even the necessity, of such intellectual tools is unquestionable, they are only tools. As very basic tools—principles, really—one must decide where and how to wield them. One can carve with those knives, but to what design?

An equally facile answer is just to say, “Inspiration.” Creative inspiration is needed, to be sure, but it is not a reliable magic wand to be blithely waved over a complex design challenge. A more effective strategy is to minimize and isolate those parts of a design for which creative inspiration is required, and to apply more prosaic and predictable techniques elsewhere. In other words, focus the creative efforts. Still, however, one has to know where creativity is and is not required.

A common, effective, and appropriate answer to the question of how to identify a feasible set of “alternative arrangements for the design as a whole” is applying experience. This is not a shallow answer to the problem. There is substantial sophistication to the use of experience. It is not foolproof and does not always suffice—but treatment of such issues comes later in the chapter.

We proceed by first discussing the most basic design tools of software engineering, then begin an extended discussion of how the lessons of experience may be used to tackle new design problems.

4.2.1 Fundamental Conceptual Tools

The most basic design tools are separation of concerns, abstraction, modularity, and other “first principles of software engineering” such as anticipation of change and design for generality. As noted above, however, they are hard to apply straight out of the box; fortunately it is usually not necessary to do so. These concepts are familiar, so we provide only a brief discussion.

Abstraction and the Simple Machines

Abstraction is the selection of a set of concepts to represent a more complex whole. One view of this definition sees abstraction as a process that moves upward, from details to summarizing concepts. As it relates to design, however, abstraction is usually employed as a tool to be used when moving downward: A set of concepts is chosen to allow discussion of an idea—an arrangement of abstract parts that (it is hoped) constitutes a solution, though still at a high level. The design then moves further downward as the concepts of the abstraction are reified into more concrete structures—a more complex whole. Ultimately, the reification process ends with the production of source code. While refinement or deduction might be more accurate and appropriate terms for this activity, *abstraction* is often used because the intent from the outset is to create source code for which the abstraction is an accurate and useful characterization.

The question remains, though, what concepts should be chosen at the outset of a design task? One good answer, and one that helps support the use of the term *abstraction* is, "Search for a simple machine that serves as an abstraction of a potential system that will perform the required task."

The simple machines of software design are many. For instance, what kind of simple machine makes a spreadsheet program? It is "just a graph of relationships, and when one node is changed, the relationships are all reevaluated to bring everything back into synch." What makes up a parser? "Essentially, it is just a push-down automaton." How does fax machine software work? "At core, it is basically just a little state machine." How does an avionics system work? "You just read all the sensors, calculate the control laws, write values out to the displays and actuators, and then do it all over again."

While these answers may seem trivial and inadequate—and they are in many ways—the point is that they provide a plausible first conception of how an application might be built. That is, they provide the first clues into designing a system's architecture.

Every application domain has its common simple machines. The commonness of these simple machines reflects, of course, experience in designing applications within those domains. Some examples are shown in Figure 4-1. Choice of a simple machine can enable further exploration of the feasibility of that design concept.

Figure 4-1.
Common simple
machines.

Domain	Simple Machines
Graphics	Pixel arrays Transformation matrices Widgets Abstract depiction graphs
Word processing	Structured documents Layouts
Industrial process control	Finite state machines
Income tax return preparation	Hypertext Spreadsheets Form templates
Web pages	Hypertext Composite documents
Scientific computing	Matrices Mathematical functions
Financial accounting	Spreadsheets Databases Transactions

Choosing the Level and Terms of Discourse

Whether or not the approach of attempting to identify simple machines is adopted, any attempt to use abstraction as a tool requires choice of a field of discourse, and once that is chosen, the selection of the terms used in discourse is needed. Often the two choices are virtually inseparable.

The linear, or standard approach, to architectural concept formation demands that the initial subject of discourse be that of the application as a whole. For example, the old technique of stepwise refinement begins at that level.

Two alternatives exist, however. The first is to choose to work, initially, at a level lower than that of the whole application. That is, the subject of discourse might be something that is optimistically assumed to be a part of a solution to the larger problem. Design is performed at that level in the hope that once several such subproblems are solved they can be composed together to form an overall solution, where the means of composition does not impinge upon their internal structure. Success with this approach need not demand great foresight; sometimes just whittling away at parts of a problem makes later tackling of the overall problem easier. Good foresight certainly helps, of course.

The occasion when this approach is particularly appropriate is when there exists a body of preexisting components that are available for reuse in the development of a new application. (Note this is almost *always* the case!) Thorough understanding of what those components are capable of, and how they may be composed, can provide critical insights into how the whole application should be structured so as to enable the potential reuse.

The second alternative is to choose to work, initially, at a level *above* that of the desired application. In some cases, this may mean solving a more general problem. This has worked well in handling complicated input to programs. Rather than building a custom component for processing an application's simple, restricted user input language, it can be very effective to employ a generic parser tool. Though the parser may end up being far more powerful than what the specific application requires, the maturity of technology development in that domain, coupled with an abundance of reusable packages embodying that technology, will likely result in a smaller, faster, off-the-shelf component being used.

In other cases, this may simply mean understanding the core issues of an application in a more fundamental way. To use a physical example, building a guitar may suggest thinking about choosing strings, wood for the fingerboard, side, top, bridge, and so on. Moving up a level of abstraction, however, would involve thinking about the physics of vibrating strings, wave propagation, and so forth [see, for example (Vogel)]. The benefit is to identify in advance what the key design factors are.

To use a software example, consider the challenge of designing an application to help in the preparation of income tax returns. For any given year, the nation's tax laws determine the information that must be reported, the calculations that must be done, and how the information must be reported on various forms. Publications from the taxing authority provide instructions and explanations. A custom application could, of course, be designed to perform those functions as required. But if the problem is reconsidered at a higher, more abstract level, it can be formulated as a spreadsheet problem coupled with a formatting (text-layout) problem coupled with hypertext. The spreadsheet includes within it the particular

formulas required in a given year. The text layout can be viewed in a similarly generic way, with rules specifying the particular layout required for a particular year. Hypertext associates instructions with the portions of the forms to which the instructions apply. Built in this way (that is, using a generic spreadsheet engine, a generic form layout engine, and a hypertext engine), a tax software application can be created and updated efficiently from year to year, enabling a profitable business.

Separation of Concerns

Separation of concerns is the subdivision of a problem into independent parts. For instance, the visual interface that a customer sees at a bank's automatic teller machine is independent from the logic used to control operation on that customer's accounts. If the concepts at issue are genuinely independent, then using this technique is straightforward. The difficulties arise when the issues are either actually or apparently intertwined.

When such interrelationships are found their causes should be identified. Sometimes interrelationships may be present just because of use, or abuse, of language. For example, a single, common word such as "account" may be used to describe what are, in fact, different entities in different parts of an application. In other cases, concepts may be intertwined because of (past or prospective) solution efficiency. For example, the presence of numeric keys in user records may simply reflect prior database implementation strategies, and not be intrinsic information about the user.

A primary example of separation of concerns at work in software architecture is the separation of a system's structure into components (loci of computation) and connectors (loci of communication). This separation reflects the independence of the issues and is at the heart of the chapter on connectors. The myriad details and concerns addressed in that chapter reveals that, like many applications of the principle, numerous trade-offs and alternatives must be considered.

Separation of concerns frequently involves many trade-offs; total independence of concepts may not be possible. The designer is thus left with the substantial obligation of assessing performance, cost, appearance, or functional trade-offs between competing conceptions.

4.2.2 The Grand Tool: Refined Experience

While the intellectual tools of abstraction and separation of concerns are of undeniable use in the process of designing an architecture, in and of themselves they provide scant guidance to the designer. Experience can be a masterful teacher, however, and can provide the guidance necessary to wield the basic tools in an effective fashion. Raw experience by itself, though, does not provide the mature guidance that enables effective tackling of new problems. Experience must be reflected upon and refined to the point where the designer understands the essential issues and lessons from prior work, enabling judicious choice of context-appropriate techniques.

The lessons from prior work include not only the lessons of successes, but also the lessons arising from failure. In either case, it is only reflection that can indicate the root causes for the success or failure, and hence the critical knowledge that can be carried forward. (This is not to say that every failure has the seeds of valuable technical advice. Regrettably, too many projects fail for reasons independent of technology.) Lessons from

prior work need not only come from one's own experience, of course. One virtue of reading widely in a technical field or attending technical conferences is to benefit from the experiences of others. There is no virtue in reinventing wheels, broken or otherwise.

When valuable technical experience is possessed, however, substantial efficiencies result. Experience can provide that initial feasible set of "alternative arrangements for the design as a whole" called for at the beginning of Section 4.1. One need not devote valuable design time to topics for which one already has a validated solution in hand. Similarly, communication between development team members is enhanced: when others have experience with the same or similar preexisting systems, it is easier to describe use of or modifications to an established approach, as compared to conveying a completely new concept.

Applying established approaches is not an entirely risk-free strategy however. Once one has a favorite hammer, everything starts to look like a nail. Pursuing such a strategy may yield a working application, but can easily yield one that is seriously deficient in some respect or is at least suboptimal. Not only are the benefits of previous systems imported by reusing an approach, but so are the limitations. Innovation is certainly inhibited by mindlessly following one's standard approach. That said, it is nonetheless clear that refined experience is indeed the Grand Tool for designers to apply in creating new applications.

Software engineering literature is replete with surefire approaches to design—tools and techniques that are assured to always work and provide that needed solution. Some of these mistaken approaches to design confuse a notation or language with a design method. UML, for instance, is a notation, not an approach. So, too, is Java. Other approaches confuse a tool or technology with a design method. Thus some have said that CORBA provides all the needed guidance for building robotic systems; others have assured us that rule-based expert systems are all we need. Most assuredly, there is no magic potion, and an adequate approach to design problems demands care, thought, creativity, discipline, experience, and method.

Silver
Bullets and
Other
Misfires

The software designer of the twenty-first century has a wide basis of experience from which to draw. Reflection on the development of thousands of systems has yielded a wide variety of approaches and lessons, ranging from broadly encompassing domain-specific architectures to simple programming language-level design patterns. The following section presents key lessons from this experience base in considerable detail.

4.3 REFINED EXPERIENCE IN ACTION: STYLES AND ARCHITECTURAL PATTERNS

The distilled architectural wisdom that has arisen from a vast body of experience comes in many shapes and flavors. As a whole, we are treating them under the rubric of styles and patterns. As introduced in Chapter 1, the notion of architectural styles is an ancient one, dating back at least to Vitruvius. Styles are designed to capture knowledge of effective designs for achieving specified goals within a particular application context. For building architecture, the combination of goals and context might be providing a house for a single family in a Mediterranean climate using stone and tile as principal building materials. The

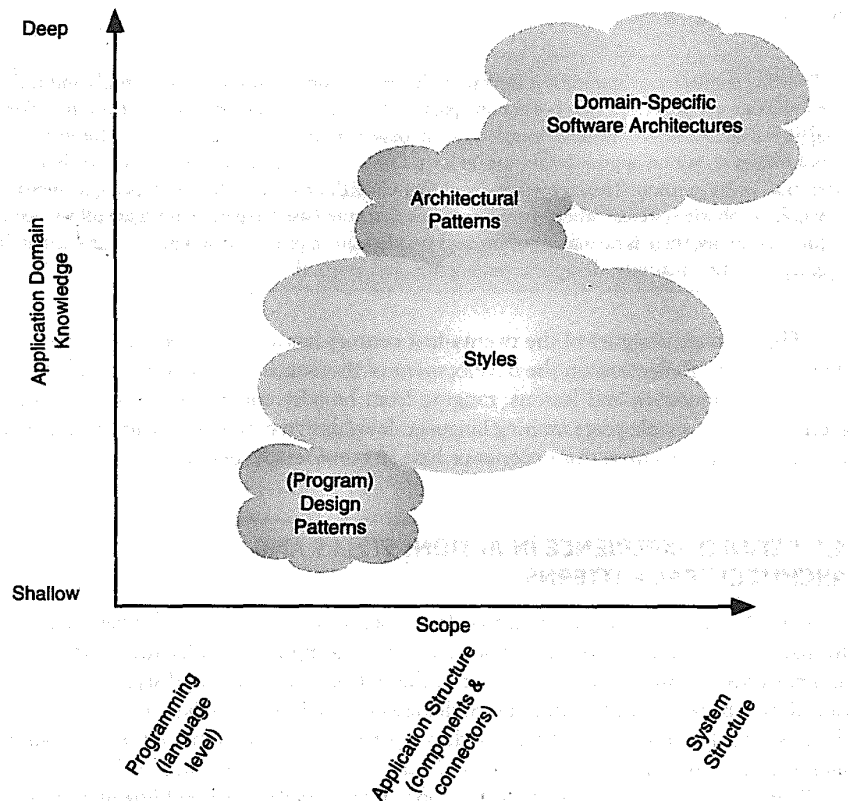
correspondingly appropriate style might be a single-story villa. For software, the goals and context might be providing a secure instant messaging system to operate between remote sites of a global company; the appropriate style might be client-server.

Patterns and styles can be characterized for large problems as well as small. Again, with regard to buildings, one can characterize a pattern for determining the appropriate overhang of a tiled roof over an entryway; similarly, one can characterize a (much more complicated) pattern for provision of elevator service in a ninety-story skyscraper. So it is with software architecture; patterns and styles have been developed that are appropriate for small as well as large and complex applications.

Not surprisingly, the lessons of experience in software architecture have arisen from several different software development communities, with each having its own contribution to make. Unfortunately, this has resulted in a variety of terms for characterizing this experience, not all of which describe distinct concepts. As an aid to teasing out the various issues and as a means to provide structure for the rest of this section, see Figure 4-2 for a summary of the main types of architectural styles and patterns.

The horizontal axis of Figure 4-2 refers to the scope of applicability of the body of knowledge—the domain of discourse. For instance, design patterns exemplified by those

Figure 4-2.
Domain
knowledge versus
scope in showing
the relationship
between styles
and patterns.
Beware! The
boundaries
between the
concepts are not
precise.



presented in the book by Gamma (Gamma et al. 1995), are focused on the programming of object-oriented solutions to program design problems. Such patterns are inapplicable to enterprise-level system design, which is at the other end of the horizontal axis.

The vertical axis reflects the amount of domain knowledge represented by (or encoded within) the body of knowledge. Program design patterns are shown at the shallow end of this scale, reflecting that they are very generally applicable and are not restricted to use within a particular application domain. At the other end of the scale are domain-specific software architectures. Such styles are applicable to the design of very large applications and concomitantly encode substantial knowledge about the design of applications within a domain. Thus, for instance, a domain-specific software architecture for retail banking might specify all major parts of the application; it is unlikely, however, that such a structure would be appropriate for any other kind of application apart from retail banking.

The diagram has fuzzy boundaries, however, and should only be taken as a rough guide. Programming languages exist at many levels of abstraction; applications range from very simple to extraordinarily complex. The scope axis should not, therefore, be taken as genuinely linear or fully ordered. The concept clouds in the diagram similarly have fuzzy boundaries; what one architect may term an architectural pattern might be called a style by another. The following sections will bring a little more precision to the discussion. While the topic of domain-specific architectures was introduced in Chapter 2 and will be dealt with at substantial length in Chapter 15, we begin the following exposition by a brief consideration of such architectures, as they are a powerful way of exploiting experience in the task of developing the architecture for a new system. We then consider architectural patterns and styles at length. These topics were introduced and defined in the preceding chapters; here, a wide variety of specific instances are presented. This chapter does not include any further discussion of program design patterns. As Figure 4-2 indicates, such patterns deal with concepts at the level of programming languages, and most often only for object-oriented languages. As such, they offer little guidance to the architect seeking to identify a feasible set of “alternative arrangements for the design as a whole.” (They do, of course, offer significant guidance to a programmer charged with implementing a chosen architecture in an object-oriented language, but that is not the topic of this chapter.)

4.3.1 Domain-Specific Software Architectures

Domain-specific software architectures (DSSAs) encode substantial knowledge, acquired through extensive experience, about how to structure complete applications within a particular domain. A useful operational definition of DSSAs is the combination of (1) a *reference architecture* for an application domain (as defined in Chapter 3), (2) a library of software components for that architecture containing reusable chunks of domain expertise, and (3) a method of choosing and configuring components to work within an instance of the reference architecture (Hayes-Roth et al. 1995). As such, a DSSA represents the most valuable type of experience useful in identifying a feasible set of “alternative arrangements for the design as a whole.”

In terms of our familiar building analogy, DSSAs are akin to the generic design of tract houses, that is, house designs that are instantiated dozens or hundreds of times to yield individual homes in a housing development. Such homes are not identical to one

another; they typically vary in terms of cabinetry, banister styles, colors, carpet, roofing material, and such. They will be same, though, with respect to the floor plan, the location of the windows, and with regard to all major structural elements. The contractor for an individual house knows how to instantiate the generic plan in light of the consumer's specific preferences and choices to yield a very cost-effective structure that nonetheless is somewhat personalized.

The extended example of software for consumer electronics presented in Chapter 1 serves as an illustration of a DSSA. (While this example was presented there as an illustration of product families, the two concepts are closely related; this relationship is carefully explored in Chapter 15.) Different consumer electronics applications can be created by parameterizing or specializing the core architecture. These specializations may represent adding new, unprecedented components into a television's architecture, for instance, but where those additions are performed on parts of the architecture are identified in advance as places for such extensions to occur.

The difficulty with DSSAs, of course, is that they are specialized for a particular domain and so are only of value if one exists for the domain wherein the engineer is charged with building a new application. While this may make it sound as though DSSAs are rare, the contrary is true. Companies tend to develop multiple applications within a particular area of expertise. This expertise manifests itself in strong similarities in structure between two products from the company and another. These similarities, and the rationale for them, is the basis of a DSSA. What is unfortunately true of such situations, however, is that frequently the common structure—the DSSA—is never written down, but remains latent in the minds of the company's engineers.

DSSAs and product families are explored in Chapter 15 in substantial detail, thus further discussion of these concepts is deferred. The reader should remember, however, that DSSAs are the preeminent means for maximal reuse of knowledge and prior development and are thus critical for developing a new product's design in an established domain.

4.3.2 Architectural Patterns

Architectural patterns are akin to DSSAs, but applied within a much narrower scope. They represent significant reuse of experience, but help you less fully. We repeat here the definition from Chapter 3:

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.

In the sections below, we sketch three very simple examples: state-logic-display, model-view-controller, and sense-compute-control.

State-Logic-Display (aka. Three-Tier)

The example used in Chapter 3 to illustrate this definition is the familiar three-tier architecture, which is commonly employed in business applications where there is a data store behind a set of business logic rules, and that business logic is accessed by user interface components. This pattern, also known as state-logic-display, has been popular since it maps nicely to a distributed implementation in which communication between the components

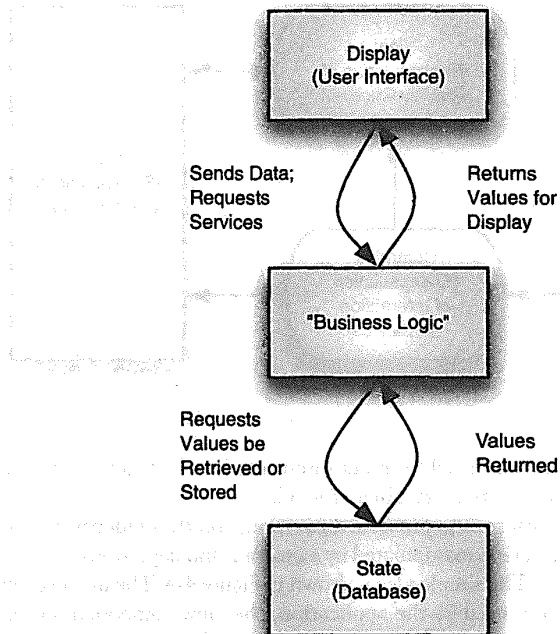


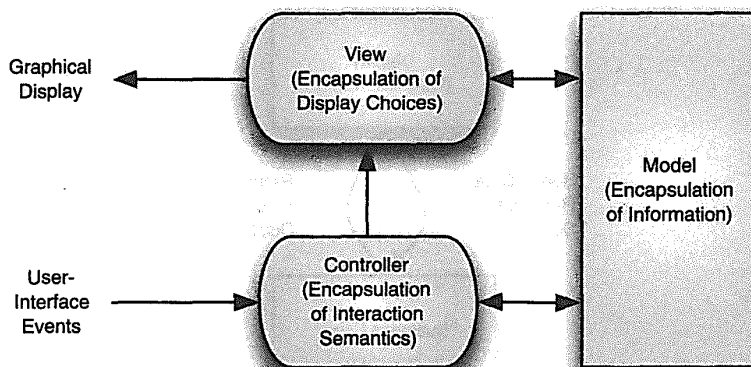
Figure 4-3.
Canonical form
of state-logic-
display
architectures.

is by remote procedure call. Figure 4-3 shows the pattern, redrawn in an orientation different from that used in Chapter 3. In business applications, the data store is usually a large database server; the business logic component may implement, for instance, some banking transaction rules; and the display component be a relatively simple component for managing interaction with a user on a desktop PC. Multiplayer games also map nicely to this architecture, wherein the business logic component implements the game rules and the data store maintains the game state. Multiple players can simultaneously play the game wherein each user has his or her own display component. Many Web-based applications can be approximately characterized as three-tier architectures, where the display component is the user's Web browser, the business logic component is the software on the Web site (Web server plus application-specific logic), and the data store is a database accessed by the Web server on a local host. This characterization of Web applications is not entirely fair, however, as communication between the display component (browser) and the business logic component (Web server) is not by means of a remote procedure call, but rather by the HTTP protocol, which does not necessarily maintain a connection between the two components in between interactions.

Model-View-Controller (for Graphical User Interfaces)

Since its invention in the 1980s, the model-view-controller (MVC) pattern has been a dominant influence in the design of graphical user interfaces. While MVC certainly is applied at the low levels of program design and thus could be classified in the design

Figure 4-4.
Notional model-
view-controller
pattern.



patterns category (and hence omitted from our discussion here), it can also be viewed as providing a structure for applications at a higher level.

The objective of the pattern is to promote separation, and thus independent development paths, between information manipulated by a program and depictions of user interactions with that information. The simple idea is shown in Figure 4-4. The model component encapsulates the information used by the application; the view component encapsulates the information chosen and necessary for graphical depiction of that information; the controller component encapsulates the logic necessary to maintain consistency between the model and the view, and to handle inputs from the user as they relate to the depiction.

The notional interactions between these components are as follows (variations exist in the practice). When the application changes a value in the model object, notification of that change is sent to the view so that any affected parts of the depiction can be updated and redrawn. Notification also typically goes to the controller as well, so that the controller can modify the view if its logic so requires. The view may query the model for additional data needed for the display. When handling input from the user (such as a mouse click on part of the view), the windowing system sends the user event to the controller; the controller may query the view for information to assist in determining what action to take. The controller then updates the model object in keeping with the desired semantics. Then, of course, if the model object changes values it must notify the view and controller so that the user interface can be updated, and so the cycle of interactions continues.

While Figure 4-4 shows three distinct components, from the description above of their interactions it is clear that there is often a very close coupling between the actions of the view component and the controller component. In many cases, therefore, the view and controller components are merged; such is the case with the architecture of Java's Swing framework.

At a much higher level of abstraction than programming, the MVC paradigm can be seen at work in the World Wide Web. Web resources correspond to the model objects; the HTML rendering agent within a browser corresponds to the viewer; the controller in the simplest case corresponds to the code that is part of the browser that responds to user input and which causes either interactions with a Web server or modifies the browser's display in

some manner. In the more complicated situation, the controller would also encompass code uploaded from a server that governs user interaction with the display, such as JavaScript uploaded as part of a Web page or code on the server that determines the representation of the resource to transmit to the browser.

MVC has been the subject of much work, and a variety of references are available discussing the details of several different implementation approaches. MVC has also inspired the development of other related architectural patterns, such as presentation-abstraction-controller (PAC). Detailed citations are provided at the end of the chapter.

Sense-Compute-Control (aka. Sensor-Controller-Actuator)

The sense-compute-control (SCC) pattern is typically used in structuring embedded control applications. This may range from simple devices such as those found in kitchen appliances to sophisticated systems used in automotive applications or robotic control. The basic idea is illustrated in Figure 4-5. A computer is embedded in some application; sensors from various devices are connected to the computer and may be sampled to determine their value. Also attached to the computer are hardware actuators. By sending a signal to such devices the computer may, for example, open a valve, lower flaps, or turn off a light, and hence control the system.

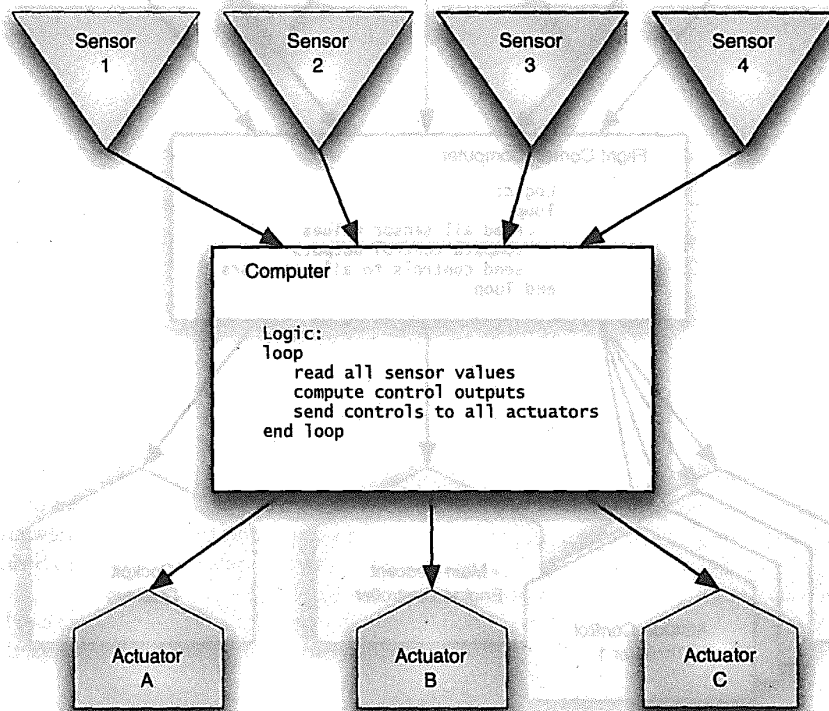
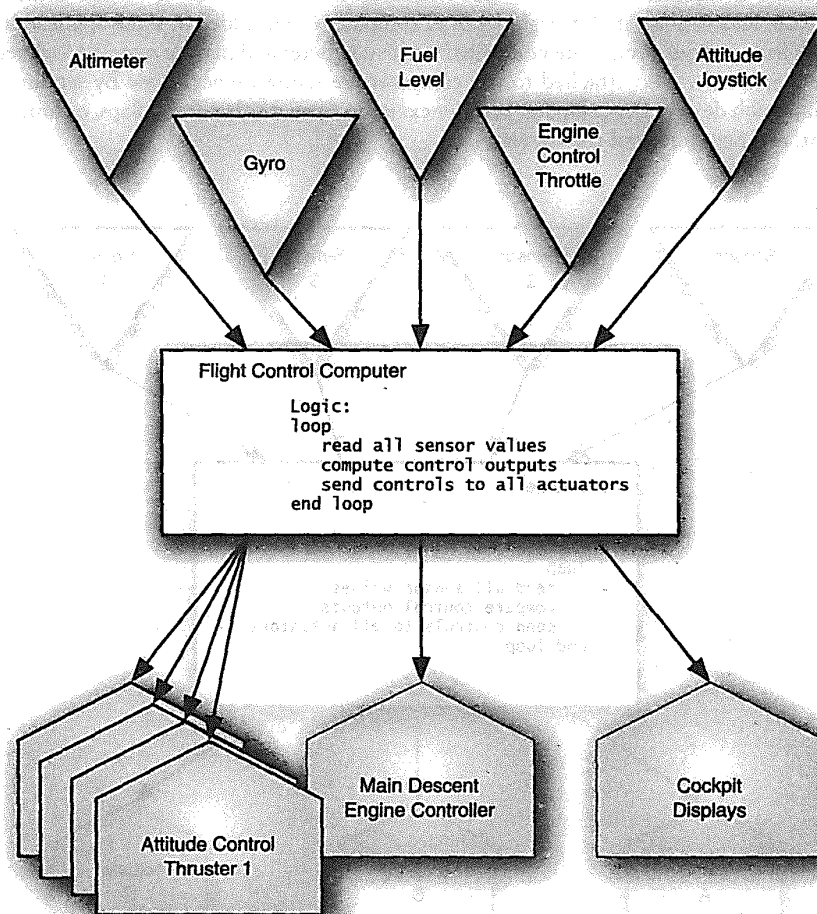


Figure 4-5.
Sense-compute-control: Different-shaped boxes are used to indicate the different types of devices present in the system.

The architectural pattern is simply one of cycling through the steps of reading all the sensor values, executing a set of control laws or functions, and then sending outputs to the various actuators. Typically, such a cycle is keyed to a clock, where the frequency of the clock ticks may be keyed to the maximum rate at which the sensors' values may change or to the sensitivity of the actuators to varying inputs on their input control lines. *Note that there is implicit feedback in such applications via the external environment.* That is, the typical situation is that by changing one of the actuators, something will change in the external environment that one of the sensors will ultimately detect, and hence cause a change in its value.

A simple flight control application can serve to make these ideas concrete. We will use the example of software controlling a notional lunar excursion module, or "lunar lander," to the surface of the moon, illustrated in Figure 4-6.

Figure 4-6.
Sense-compute-
control applied to
a lunar lander.



In this example there are five sensors on board the spacecraft: The altimeter (presumably radar) senses the altitude above the surface of the moon; the gyro provides information on the x-y-z axis orientation of the craft; the fuel level indicates the remaining fuel on board; the engine control throttle indicates to the computer the pilot's desired setting of the descent engine's thrust (for instance, 75 percent throttle); and the joystick indicates the pilot's inputs for orienting the spacecraft. (Keep in mind that this is an entirely fly-by-wire application, so there is no physical connection between the joystick, for instance, and any control surfaces. The computer is responsible for sensing all control device inputs.) The diagram shows six actuators: four attitude control thrusters, the main descent engine controller, and the cockpit displays.

As the craft approaches the surface, the sensors will provide to the computer information on the Lunar Lander's altitude, attitude, fuel remaining, and the pilot's control inputs. The flight control computer will process its control laws, computing values to be sent to the various actuators to control the descent engine, the attitude control thrusters, and update the cockpit displays. From changes in the altitude the computer could determine, for example, the descent rate, and display that to the pilot. Based upon the pilot's judgment, he or she could increase the throttle to slow the descent or decrease the throttle and allow the Lunar gravity to increase the descent rate.

Depending on the system designer's choices, a variety of control strategies could be programmed into the flight control computer. The scenario sketched above reflects the strategy applied in the Apollo landings. A fully automated landing sequence could also be programmed, in which the software would determine the optimal descent engine and attitude control thruster settings, based upon the altitude and attitude of the craft. From the standpoint of the software's architecture, these different strategies are all the same; at each cycle of the loop the software reads the current sensor values, computes the desired settings for the actuators, and sends those values to the actuators.

4.3.3 Introduction to Styles

Architectural styles are a primary way of characterizing lessons from experience in software system design. As such, they can be a key element in developing initial or detailed conceptions of a software system's architecture. As Figure 4-2 indicates, architectural styles are broadly applicable; they can be useful in determining everything from subroutine structure to top-level application structure. Styles, as used here and as seen in contrast with the architectural patterns of the preceding section, reflect less domain knowledge than architectural patterns, and hence are more broadly applicable. Keep in mind, however, that the boundaries between these concepts are not precise.

Chapter 3 introduced architectural styles in some detail, and presented the following definition:

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.

Accordingly, the discussion of styles will include the decisions and constraints comprising the style, as well as the beneficial qualities induced by these choices.

**Styling with
Perry and
Wolf**

Dewayne Perry and Alexander Wolf wrote early and persuasively on the importance of styles in software architecture. The influence of their work—widely cited in the literature—motivates inclusion of the following extended quote.

The notion of architectural style is particularly useful from both descriptive and prescriptive points of view. Descriptively, architectural style defines a particular codification of design elements and formal arrangements. Prescriptively, style limits the kinds of design elements and their formal arrangements. That is, an architectural style constrains both the design elements and the formal relationships among the design elements. Analogously, we shall find this a most useful concept in software architecture.

Of extreme importance is the relationship between engineering principles and architectural style (and, of course, architecture itself). For example, one does not get the light, airy feel of the perpendicular style as exemplified in the chapel at King's College, Cambridge, from Romanesque engineering. Different engineering principles are needed to move from the massiveness of the Romanesque to lightness of the perpendicular. It is not just a matter of aesthetics. This relationship between engineering principles and software architecture is also of fundamental importance.

Finally, the relationship between architectural style and materials is of critical importance. The materials have certain properties that are exploited in providing a particular style. One may combine structural with aesthetic uses of materials, such as that found in the post and beam construction of Tudor-style houses. However, one does not build a skyscraper with wooden posts and beams. The material aspects of the design elements provide both aesthetic and engineering bases for an architecture. Again, this relationship is of critical importance in software architecture.

Thus, we find in building architecture some fundamental insights about software architecture: multiple views are needed to emphasize and to understand different aspects of the architecture; styles are a cogent and important form of codification that can be used both descriptively and prescriptively; and, engineering principles and material properties are of fundamental importance in the development and support of a particular architecture and architectural style. . . .

The important thing about an architectural style is that it encapsulates important decisions about the architectural elements and emphasizes important constraints on the elements and their relationships. The useful thing about style is that we can use it both to constrain the architecture and to coordinate cooperating architects. Moreover, style embodies those decisions that suffer erosion and drift. An emphasis on style as a constraint on the architecture provides a visibility to certain aspects of the architecture so that violations of those aspects and insensitivity to them will be more obvious.

(Perry and Wolf 1992)

The architectural styles chosen for presentation here were selected on the basis of reflecting a diversity of approaches, common use, and being a part of the common vocabulary of software architects. Sufficient detail is presented so that the reader can appreciate the essence of each style and the situations for which it is appropriate. While this chapter does not present a comprehensive catalog of styles, many of the most popular and well known are discussed, showing the designer the kind of role that styles can play in solving design problems. (Chapter 11 will explore a few more advanced styles, such as REST, in some detail.)

There are a variety of potential ways of classifying and organizing styles. We have chosen to present the simple styles according to the following outline:

Traditional Language-Influenced Styles

- Main program and subroutines
- Object-oriented

Layered

- Virtual machines
- Client-server

Dataflow Styles

- Batch-sequential
- Pipe-and-filter

Shared Memory

- Blackboard
- Rule-based

Interpreter

- Interpreter
- Mobile code

Implicit Invocation

- Publish-subscribe
- Event-based

Peer-to-Peer

Later in the chapter, two slightly more complex styles are discussed: C2 and distributed objects, with specific consideration of CORBA.

Other organizations would highlight other similarities and relationships among the presented styles. The above categorization should thus only be taken as one coarse-grained perspective, useful primarily in the initial introduction of the styles. A summary/comparison of all the styles is included in Table 4-1 toward the end of the chapter.

Most of the styles presented will be illustrated by reference to a type of application for which the style is well suited and by sketching a game version of the lunar lander program that was used in illustrating the sense-compute-control (SCC) pattern. By using the same, or a similar, application with each style some sharp contrasts between the styles will be evident. The game version of lunar lander differs from the SCC version in one critical respect: The environment needs to be simulated. In a real flight control application, the actions of the physical universe affect the spacecraft. For instance, as the lander approaches the moon, the moon's gravitational field draws the spacecraft down. Each time the sensors are read different values will appear, not due to any action of the software, but simply

because of the action of gravity. In the game version, the physics of the universe must be simulated. For example, upon each tick of a game clock the values of the sensors could be computed by an environment simulator, rather than read from genuine sensor devices. As you compare the various examples, please be aware that the various lunar lander games are not equivalent. The purpose of the examples is to help develop understanding of the various styles through reference to a common application context—not a common, precise, and complete requirements specification.

Some of the styles discussed have complex details associated with their use that have been omitted here. Our primary objective is to illustrate a wide range of diverse styles that represent the capture of extensive experience. The material presented here can serve as an introduction to the styles; detailed instructions for their use must be sought elsewhere.

4.3.4 Simple Styles

Traditional Language-Influenced Styles

The two styles considered here reflect the program styles that result from the traditional use of programming languages such as C, C++, Java, and Pascal. These languages, of course, can be used to implement architectures in very different styles; discussed here are architectures that primarily reflect the basic organizational and control-flow relationships between components provided by these imperative languages.

Main Program and Subroutines. The main program and subroutine style should be instantly familiar to anyone who has programmed in a language such as C, Fortran, Pascal, or Basic. The components are the main program and the various subroutines (aka. procedures); the connectors between the components are procedure calls. For purposes of this discussion we specifically exclude programs that use shared memory (such as global variables) as a means of communication from being a part of this style.

Style: Main Program and Subroutines (no shared memory)

Summary: Decomposition based upon separation of functional processing steps.

Components: Main program and subroutines.

Connectors: Function/procedure calls.

Data elements: Values passed in/out of subroutines.

Topology: Static organization of components is hierarchical; full structure is a directed graph.

Additional constraints imposed: None.

Qualities yielded: Modularity: Subroutines may be replaced with different implementations as long as interface semantics are unaffected.

Typical uses: Small programs; pedagogical purposes.

Cautions: Typically fails to scale to large applications; inadequate attention to data structures. Unpredictable effort required to accommodate new requirements.

Relations to programming languages or environments: Traditional imperative programming languages, such as BASIC, Pascal, or C.

A simple version of the lunar lander game program is shown in Figure 4-7. Its design is based on the functional decomposition of the processing into repeated user interface steps

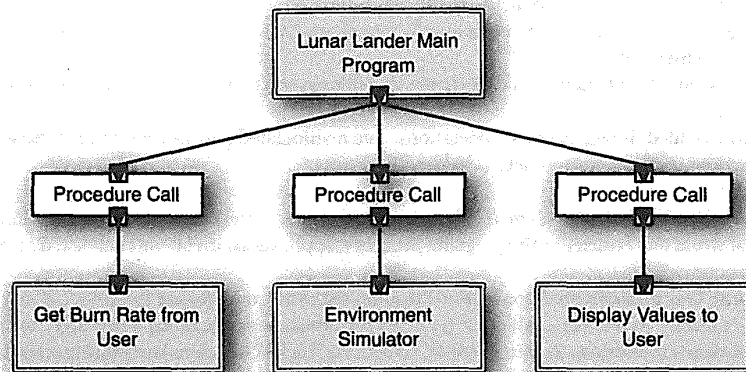
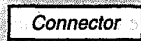


Figure 4-7.
Lunar Lander:
Main program
and subroutines.

Legend



— Connects a *requires* interface to a *provided* interface

(input and output) and all the processing that corresponds to simulating the environment and the actions of the spacecraft. Specifically, the main program displays greetings and instructions, then enters a loop wherein it calls the three subroutine components in turn. The first obtains the pilot's throttle-setting input. The second component primarily serves as the environment simulator, determining how much fuel is left and what the altitude and descent rate are. The only flight control law in this simple game is the translation of a pilot-specified throttle percentage into a volume/second fuel-flow setting (the burn rate) to control the descent engine. The third component displays the updated state. In this game there is no clock, so one cycle through the functional processing corresponds to one clock tick.

Object-Oriented. The object-oriented style should be similarly familiar. The only structuring provided is that of a world of objects whose lifetimes vary according to use. Comprehending a program structured in this manner requires understanding the numerous static and dynamic relationships among the objects.

Summary: State strongly encapsulated with functions that operate on that state as objects.

Objects must be instantiated before the objects' methods can be called.

Components: Objects (aka. instance of a class).

Connector: Method invocation (procedure calls to manipulate state).

Style:
Object-
Oriented

Data elements: Arguments to methods.

Topology: Can vary arbitrarily; components may share data and interface functions through inheritance hierarchies.

Additional constraints imposed: Commonly: shared memory (to support use of pointers), single-threaded.

Qualities yielded: Integrity of data operations: data manipulated only by appropriate functions. Abstraction: implementation details hidden.

Typical uses: Applications where the designer wants a close correlation between entities in the physical world and entities in the program; pedagogy; applications involving complex, dynamic data structures.

Cautions: Use in distributed applications requires extensive middleware to provide access to remote objects. Relatively inefficient for high-performance applications with large, regular numeric data structures, such as in scientific computing. Lack of additional structuring principles can result in highly complex applications.

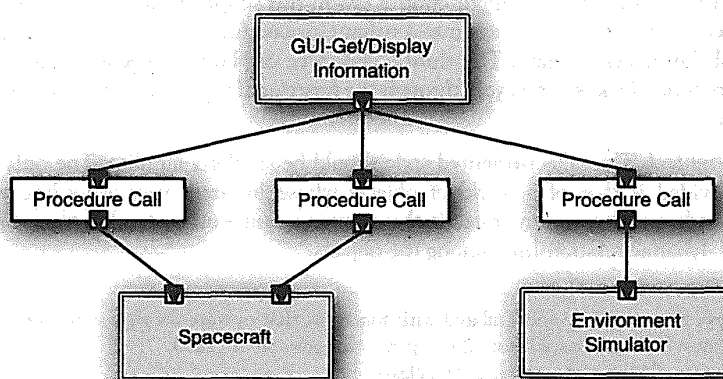
Relations to programming languages or environments: Java, C++.

The object-oriented design of the lunar lander game shown in Figure 4-8 has three encapsulations: the spacecraft, the user's interface, and the environment (essentially a physics model that allows calculation of the descent rate). Note a contrast with the main program and subroutine's (MPS) design: Interactions here with the user are handled by a single object, which performs both input and output functions. The functional decomposition of the MPS design resulted in those functions being performed by separate subroutines. Some details of the object-oriented design are apparent in Figure 4-9, which is a Unified Modeling Language characterization of the objects.

Layered

Layered styles are as simple and familiar as those above that reflect the traditional use of programming languages such as C or Java. The essence of a layered style is that an

Figure 4-8.
Lunar Lander in
the object-
oriented style.



architecture is separated into ordered layers, wherein a program within one layer may obtain services from a layer below it. The virtual machines style is familiar to anyone who has studied computer and operating systems architectures; the client-server layered style is ubiquitous in business applications. Both are discussed below.

Virtual Machines. In the virtual machines style, a *layer* offers a set of services (“a machine with a bunch of buttons and knobs”) that may be accessed by, at least, programs residing within the layer above it. The services that a layer offers can be termed the “provides interface” of the layer. The services may be implemented by various programs within the layer, but, when designing an architecture, from the perspective of the entities that use the layer’s services such distinctions are not apparent. In Figure 4-10, for instance, program A in layer 1 can access the services provided by layer 2; those services may in fact be implemented by programs B and C, but such details are not of concern to program A.

In a strict virtual machines style, programs at a given level may only access the services provided by the layer immediately below it. Thus, again referring to Figure 4-10, program A can only access services provided by layer 2, and not layer 3. A nonstrict virtual machines style would allow A to access the services of layer 3 as well.

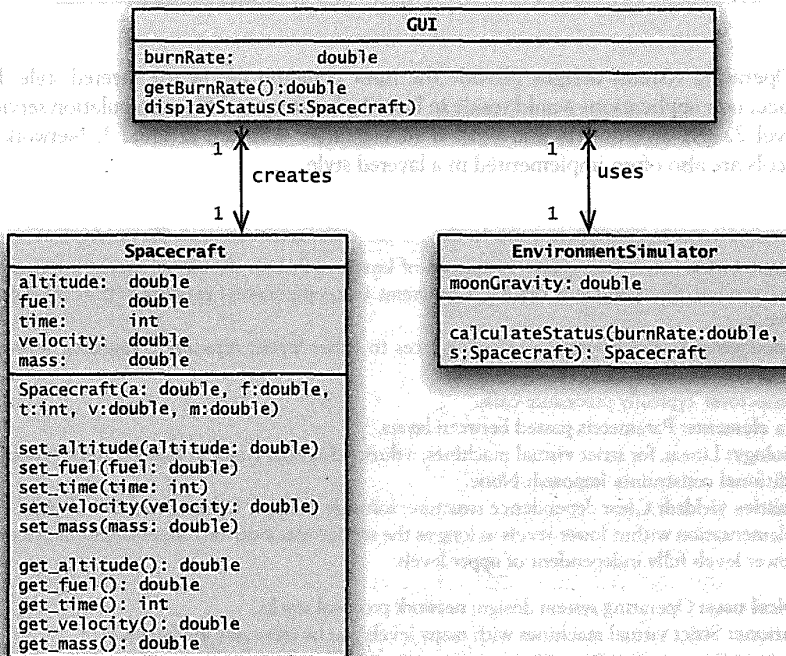
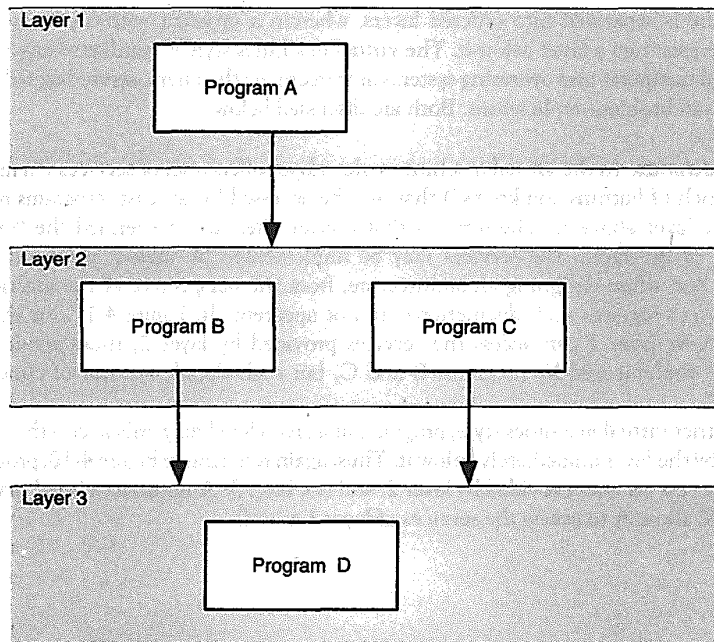


Figure 4-9.
JML
representation of
the classes used in
Figure 4-8.

Figure 4-10.
Notional virtual
machines
architecture.



Operating systems designs provide the most common use of the layered style. For instance, user applications would reside in layer 1, directory and file manipulation services on level 2, and disk drivers and volume management software at level 3. Networking protocols are also often implemented in a layered style.

Style: Virtual Machines

Summary: Consists of an ordered sequence of layers; each *layer*, or virtual machine, offers a set of services that may be accessed by programs (subcomponents) residing within the layer above it.

Components: Layers offering a set of services to other layers, typically comprising several programs (subcomponents).

Connectors: Typically procedure calls.

Data elements: Parameters passed between layers.

Topology: Linear, for strict virtual machines; a directed acyclic graph in looser interpretations.

Additional constraints imposed: None.

Qualities yielded: Clear dependence structure; software at upper levels immune to changes of implementation within lower levels as long as the service specifications are invariant. Software at lower levels fully independent of upper levels.

Typical uses: Operating system design; network protocol stacks.

Cautions: Strict virtual machines with many levels can be relatively inefficient.

A five-level version of the lunar lander game is shown in Figure 4-11. The top layer handles inputs received from the user through the keyboard; it calls the second layer to service those inputs. The second layer includes all the details of the lunar lander that pertain to the game logic and the environment simulator. It calls the third layer to begin the process of displaying the updated Lander state to the user. The third layer is a generic, two-dimensional game engine. Such a layer is capable of supporting any of a wide variety of games that only require two-dimensional graphics. The fourth layer is the operating system, which provides, among other things, platform-specific user interface (UI) support, such as window management. The bottom layer is provided by firmware or hardware, and

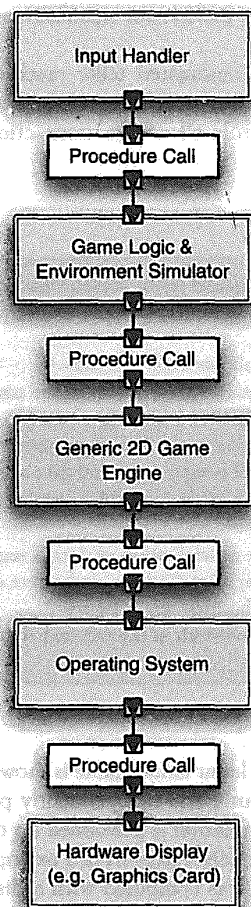


Figure 4-11.
Lunar Lander in
the virtual
machines style.

drives the physical user interface display. Each layer is independent of the layers above it, and only requires access to the services of the layer immediately below it. Note that the focus of this architecture is different from the preceding examples. The explicit inclusion here of a game engine, the operating system, and hardware is intended to help illustrate the layered style.

Client-Server. The client-server style can be simply understood as a two-layer virtual machine with network connections. That is, the server is the virtual machine below the clients, each of which accesses the virtual machine's interfaces via remote procedure calls or equivalent network access methods. Typically, there are multiple clients that access the same server; the clients are mutually independent. The obligation of the server is to provide the specific services requested by each client. Clients are sometimes referred to as "thin" or "thick," reflecting whether they include any significant processing beyond user interface functions.

Grossly simplified, a bank's automated teller machine (ATM) system is a client-server application. The clients are the ATMs and the server is the bank, which keeps track of all accounts and controls all of the transactions requested by users at the ATMs.

Style: Client-Server

Summary: Clients send service requests to the server, which performs the required functions and replies as needed with the requested information. Communication is initiated by the clients.

Components: Clients and server.

Connectors: Remote procedure call, network protocols.

Data elements: Parameters and return values as sent by the connectors.

Topology: Two-level, with multiple clients making requests to the server.

Additional constraints imposed: Client-to-client communication prohibited.

Qualities yielded: Centralization of computation and data at the server, with the information made available to remote clients. A single powerful server can service many clients.

Typical uses: Applications where centralization of data is required, or where processing and data storage benefit from a high-capacity machine, and where clients primarily perform simple user interface tasks, such as many business applications.

Cautions: When the network bandwidth is limited and there are a large number of client requests.

A multiplayer version of the lunar lander game is shown in Figure 4-12. In the system shown, three players simultaneously and independently play the game. All game state, game logic, and environment simulation is performed on the server. The clients perform the user interface functions. The connectors in this example are remote procedure calls, which are procedure call connectors combined with a distributor. A distributor identifies network interaction paths and subsequently routes communication along those paths. (The intricacies of connector types and their substructures will be explained in detail in Chapter 5.)

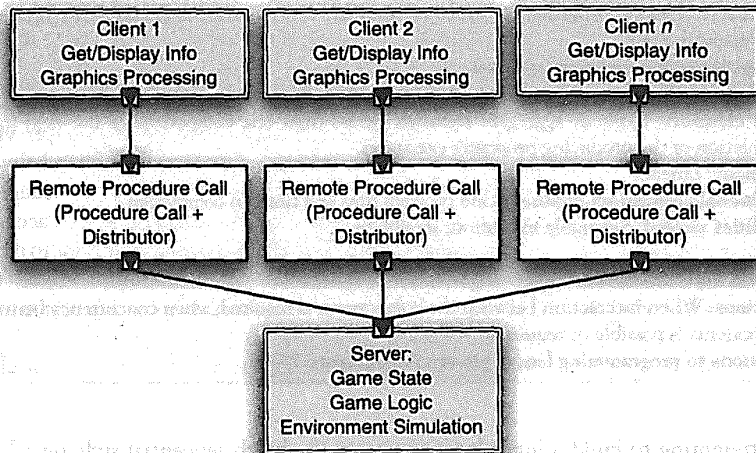


Figure 4-12.
A multiplayer
version of Lunar
Lander, in a
client-server
style.

Dataflow Styles

The two styles presented here can be characterized as dataflow styles. Their essence concerns the movement of data between independent processing elements.

Batch-sequential. The batch-sequential style is one of the oldest in the design of computer systems. It arose in the early days of data processing when the limitations of computing equipment required that large problems be subdivided into severable components that could communicate by the transfer of magnetic tapes.

A classic example of this style is shown in Figure 4-13. The application is one of updating a bank's record of all its accounts, based upon the overnight processing of the day's transactions (such as deposits or withdrawals from an account). The process starts with a master tape that has all of the bank's accounts on it, sorted by account number, and a daily tape that has the day's transactions on it.

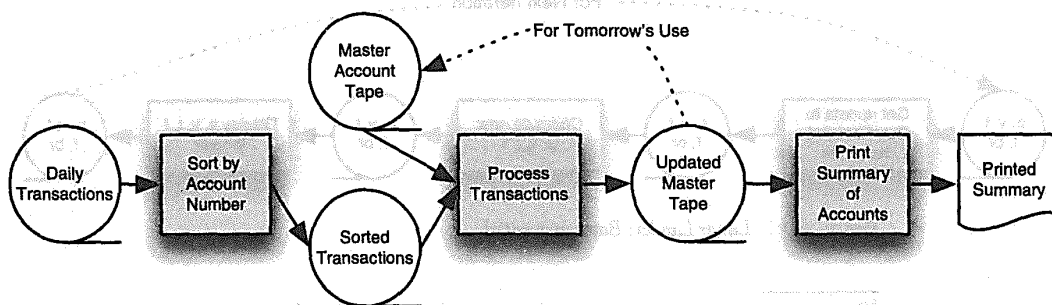


Figure 4-13. Financial records processed in batch-sequential style.

Style: Batch-Sequential.

Summary: Separate programs are executed in order; data is passed as an aggregate from one program to the next.

Components: Independent programs.

Connectors: The human hand carrying tapes between the programs, aka “sneaker-net.”⁵

Data elements: Explicit, aggregate elements passed from one component to the next upon completion of the producing program’s execution.

Topology: Linear.

Additional constraints imposed: One program runs at a time, to completion.

Qualities yielded: Severable execution; simplicity.

Typical uses: Transaction processing in financial systems.

Cautions: When interaction between the components is required; when concurrency between components is possible or required.

Relations to programming languages or environments: None.

Attempting to build a lunar lander game in the batch sequential style reveals how mismatched the style is to the application. The outline of such a (bizarre) design is presented in Figure 4-14. Here the game state is maintained on magnetic tapes.

Each functional processing step of the game is performed by a separate program. Upon performing the step’s function, the program produces an updated version of the game state, which is then handed off to the next program in the process. After the final step of displaying the game state is performed, the produced tape is carried back to the first program for another pass. This is obviously not a highly interactive, real-time game!

Pipe-and-Filter. Though radically different in detail from batch-sequential, the pipe-and-filter style introduced in Chapter 1 is also a dataflow style. Recall that the essence of the style is that streams of character data are passed from one filter program to another. The filters can operate concurrently, with no requirement for a producing component to finish before a component that consumes the producer’s output begins. Chapter 1 included a natural application of the pipe-and-filter style to the problem of producing a sorted list of selected file names.

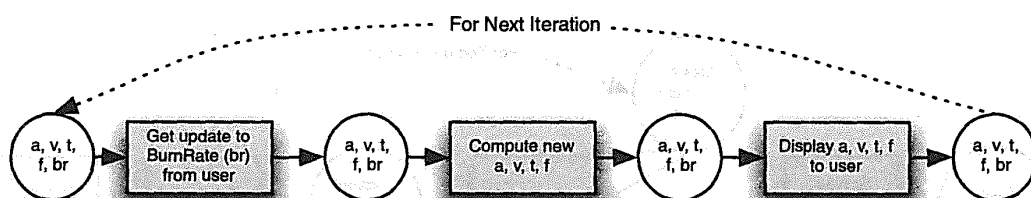


Figure 4-14. Lunar Lander: Batch-sequential.

⁵Savvy computer operators often wear athletic shoes, also known as “sneakers.”

Summary: Separate programs are executed, potentially concurrently; data is passed as a stream from one program to the next.

Components: Independent programs, known as filters.

Connectors: Explicit routers of data streams; service provided by operating system.

Data elements: Not explicit; must be (linear) data streams. In the typical Unix/Linux/DOS implementation the streams must be text.

Topology: Pipeline, though T fittings are possible.

Qualities yielded: Filters are mutually independent. Simple structure of incoming and outgoing data streams facilitates novel combinations of filters for new, composed applications.

Typical uses: Ubiquitous in operating system application programming.

Cautions: When complex data structures must be exchanged between filters; when interactivity between the programs is required.

Relations to programming languages or environments: Prevalent in Unix shells.

Style:
Uniform
Pipe-and-
Filter.

A design for the lunar lander game in the pipe-and-filter style appears in Figure 4-15. Get Burn Rate runs looping on its own, always polling the user for a value. When a value is generated it is sent off through the stream connector to the second filter. The second filter runs looping on its own as well, updating time as it chooses. It serves as the environment simulator and calculates the descent rate of the spacecraft. The third filter likewise loops, updating the display whenever new values become available on its input port. This design follows from a rather functional understanding of the game, similar to the main program and subroutines design. Functioning of a stream connector, and pipe-and-filter in general, is explored in much more depth in Chapter 5.

Shared State

The essence of shared state styles (sometimes colloquially referred to as “shared memory” styles) is that multiple components have access to the same data store, and communicate through that data store. This corresponds roughly to the ill-advised practice of using global data in C or Pascal programming. The difference is that with shared state styles the center of design attention is explicitly on these structured, shared repositories, and that consequently they are well-ordered and carefully managed. The use of global data in programming is usually only one of several means of communication between subprograms, and is done as an expedience.



Figure 4-15.
Lunar Lander in
pipe-and-filter
style.

Blackboard. The blackboard style arose in artificial intelligence applications. The intuitive sense of the style is one of many diverse experts sitting around a blackboard, all attempting to cooperate in the solution of a large, complex problem. As any given expert recognizes some part of the problem on the blackboard for which he feels competent to solve, he grabs that subproblem, goes away and works on it, and when finished, returns and posts the solution on the blackboard. Posting that solution may enable another expert to identify a problem which he can solve, and so the process continues until the whole problem is solved. Thus the state of information on the blackboard determines the order of execution of the various "expert" programs.

Style:
Blackboard

Summary: Independent programs access and communicate exclusively through a global data repository, known as a blackboard.

Components: Independent programs, sometimes referred to as "knowledge sources," blackboard.

Connectors: Access to the blackboard may be by direct memory reference, or can be through a procedure call or a database query.

Data elements: Data stored in the blackboard.

Topology: Star topology, with the blackboard at the center.

Variants: In one version of the style, programs poll the blackboard to determine if any values of interest have changed; in another version, a blackboard manager notifies interested components of an update to the blackboard.

Qualities yielded: Complete solution strategies to complex problems do not have to be preplanned. Evolving views of the data/problem determine the strategies that are adopted.

Typical uses: Heuristic problem solving in artificial intelligence applications.

Cautions: When a well-structured solution strategy is available; when interactions between the independent programs require complex regulation; when representation of the data on the blackboard is subject to frequent change (requiring propagating changes to all the participating components).

Relations to programming languages or environments: Versions of the blackboard style that allow concurrency between the constituent programs require concurrency primitives for managing the shared blackboard.

Figure 4-16 shows the lunar lander program solved in a blackboard style. Here, a single connector regulates access from the various experts in manipulating the information on the blackboard. The blackboard maintains the game state; the experts perform the independent tasks of (1) updating the descent engine burn rate based upon input from the user, (2) displaying to the user the current state of the spacecraft and any other aspect of the game state, and (3) updating the game state based upon a time and physics model. That is, the third component determines how fast time passes and updates the game state in accordance with the passage of time and the physics model of the moon and the Lander.

Rule-Based/Expert System. A rule-based architecture is a highly specialized type of shared memory architecture. The shared memory is a so-called knowledge base, that is, a database that contains facts (statements of values of variables) and production rules which consist

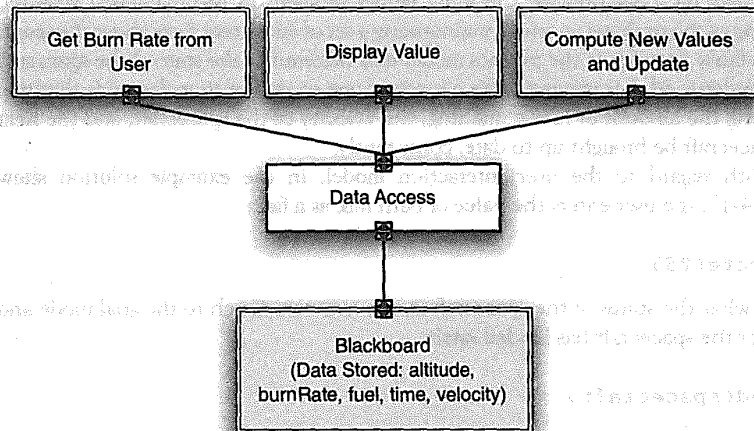


Figure 4-16.
Lunar Lander in
blackboard style.

of “if . . . then” clauses over the set of variables. A user interface component provides two modes: one for entering facts and production rules and another for entering queries, known as goals. Operating on the knowledge base, in response to user input, is an inference engine. Facts and production rules are added to the knowledge base; goals are compared against existing facts in the database. If an exact match is found, it returns *true* to the user interface. Otherwise, it evaluates the rules as necessary to determine the validity of the goal.

Summary: Inference engine parses user input and determines whether it is a fact/rule or a query. If it is a fact/rule, it adds this entry to the knowledge base. Otherwise, it queries the knowledge base for applicable rules and attempts to resolve the query.

Components: User interface, inference engine, knowledge base.

Connectors: Components are tightly interconnected, with direct procedure calls and/or shared data access.

Data Elements: Facts and queries.

Topology: Tightly coupled three-tier (direct connection of user interface, inference engine, and knowledge base).

Qualities yielded: Behavior of the application can be easily modified through dynamic addition or deletion of rules from the knowledge base. Small systems can be quickly prototyped. Thus useful for iteratively exploring problems whose general solution approach is unclear.

Typical uses: When the problem can be understood as matter of repeatedly resolving a set of predicates.

Cautions: When a large number of rules are involved, understanding the interactions between multiple rules affected by the same facts can become very difficult. Understanding the logical basis for a computed result can be as important as the result itself.

Relations to programming languages or environments: Prolog is a common language for building rule-based systems.

Style: Rule-Based/Expert System

Designing a solution for the lunar lander problem in the rule-based style requires thinking of the problem as one of maintaining a set of consistent facts about the spacecraft. This is fairly natural, as the physics model that determines the state of the spacecraft can be so understood. For instance, the passage of one second of time demands that the facts regarding the amount of fuel remaining, the velocity of the spacecraft, and the height of the spacecraft be brought up to date, consistently.

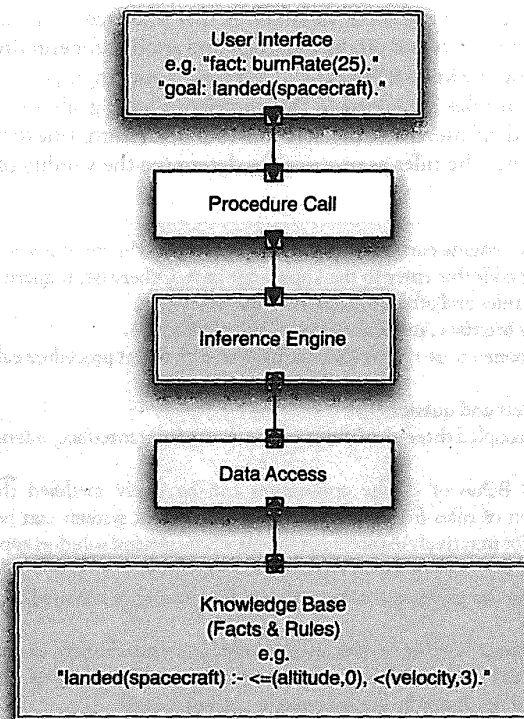
With regard to the user interaction model, in the example solution shown in Figure 4-17, the user enters the value of burn rate as a fact:

burnrate(25)

To see what the status of the spacecraft is, the user can switch to the goal mode and asks whether the spacecraft has landed safely:

landed(spacecraft)

Figure 4-17.
Lunar Lander in
a rule-based
style.



To handle this query, the inference engine queries the database. If the existing facts and other production rules satisfy the conditions “altitude ≤ 0” and “velocity < 3 ft/s,” then the engine returns true to the user interface. Otherwise, it returns false.⁶

Interpreter

The distinctive characteristic of interpreter styles is dynamic, on-the-fly interpretation of commands. Commands are explicit statements, possibly created moments before they are executed, possibly encoded in human-readable and editable text. Commands are phrased in terms of predefined primitive commands. Interpretation proceeds by starting with an initial execution state, obtaining the first command to execute (possibly by reading some user input), executing the command over the current execution state, thereby modifying that state, then proceeding to execute the next command. Identification of the next command may be affected by the result of executing the previous command; indeed, in the general case the next command may be the product of the execution of the previous command.

The *basic interpreter* architectural style involves the execution of commands one at a time. Similarly, though typically at a larger granularity, the *mobile code* style involves the execution of one chunk of code at a time. The difference with mobile code is that the place where the commands are executed may vary over time.

Basic Interpreter. Execution of commands in the basic interpreter style is similar to the rule-based style described above, where the inference engine parses a command input and performs variable resolution based on the knowledge repository. In the case of the basic interpreter style, the command interpreter is more general (capable of more generic operations than performing inferences over a set of rules). The interpretation of a single command may involve numerous primitive operations. The knowledge repository is similarly more general since arbitrary data structures may be involved.

While perhaps sounding complicated, the interpreter style is quite familiar to what might be the world's largest cohort of programmers: the users of Microsoft Excel. Excel's formulas are in fact commands interpreted by the Excel execution engine—the interpreter. Similarly Excel's macros are interpreted by the Visual Basic interpreter. Many graphical editing programs are similarly interpreter based: a structure of commands describing the drawing is maintained by the drawing program. The drawing on the screen seen by the user is created by an internal interpreter running over the text, interpreting it, issuing drawing commands to the graphics engine.

⁶For a realistic model, the set of rules in the database will be extensive to perform the computation of a , v , t , f . A simple one-dimensional physics equation calculates altitude as follows:

$$\frac{1}{2} * (\text{accelerationThrust} - \text{accelerationGravity}) * t^2 + v * t + \text{altitudeOld}$$

However, a more realistic calculation of thrust is quite complicated (see notes at the end of the chapter for references).

**Style:
Interpreter**

Summary: Interpreter parses and executes input commands, updating the state maintained by the interpreter.

Components: Command interpreter, program/interpreter state, user interface.

Connectors: Typically the command interpreter, user interface, and state are very closely bound with direct procedure calls and shared state.

Data elements: Commands.

Topology: Tightly coupled three-tier; state can be separated from the interpreter.

Qualities yielded: Highly dynamic behavior possible, where the set of commands is dynamically modified. System architecture may remain constant while new capabilities are created based upon existing primitives.

Typical uses: Superb for end-user programmability; supports dynamically changing set of capabilities.

Cautions: When fast processing is needed (it takes longer to execute interpreted code than executable code); memory management may be an issue, especially when multiple interpreters are invoked simultaneously.

Relations to programming languages or environments: Lisp and Scheme are interpretive languages, and sometimes used when building other interpreters; Word/Excel macros.

In the version of the lunar lander game shown in Figure 4-18, for each command entered, the interpreter engine processes the code and updates the interpreter state as necessary. The commands in this case are specific directives regarding how the spacecraft should be manipulated—most importantly, how the descent engine should be throttled. The interpreter, thus, has been programmed specifically to interpret such commands. It may return values to the user, depending on the command. Thus, when the user enters “BurnRate(50),” the interpreter takes BurnRate as the command and the parameter as the amount of fuel to burn. It then calculates the necessary updates to the altitude, fuel level, and velocity. Time is simulated by having t incremented each occasion the user enters a BurnRate command. When the user enters the CheckStatus command, the user receives the current state of altitude, fuel, time, and velocity.

Mobile Code. As the name suggests, mobile code styles enable code to be transmitted to a remote host for interpretation. This may be due to a lack of local computing power, lack of resources, or due to large data sets remotely located. Mobile code may be classified as code on demand, remote evaluation, or mobile agent depending on where the code is being transmitted, who requested the transmission, and where the program state resides. Code on demand is when the initiator has resources and state but downloads code from another site to be executed locally. Remote evaluation is when the initiator has the code but lacks the resources (such as the interpreter) to execute the code. Thus, it transmits code to be processed at a remote host, such as in grid computing. Results are returned to the initiator. Mobile agent is when the initiator has the code and the state but some of resources are located elsewhere. Thus the initiator moves to a remote host with the code, the state, and some of the resources. Processing need not return to the initiator.

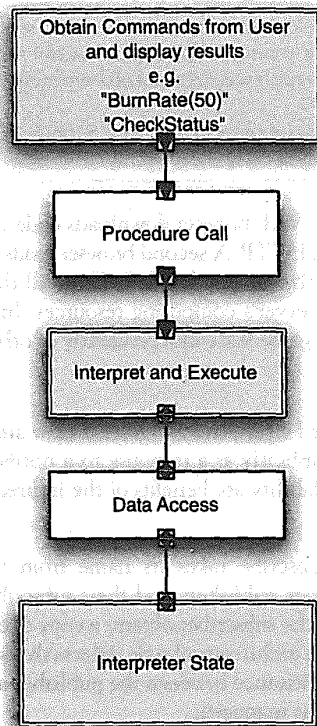


Figure 4-18.
Lunar Lander in
interpreter style.

In all of the mobile code styles, a data element (some representation of a program) is dynamically transformed into a data processing component (when those commands are interpreted or otherwise executed).

Summary: Code moves to be interpreted on another host; depending on the variant, state does also.

Components: Execution dock, which handles receipt and deployment of code and state; code compiler/interpreter.

Connectors: Network protocols and elements for packaging code and data for transmission.

Data elements: Representations of code as data; program state; data.

Topology: Network.

Variants: Code-on-demand, remote evaluation, and mobile agent.

Qualities yielded: Dynamic adaptability. Takes advantage of the aggregate computing power of available hosts; increased dependability through provision of migration to new hosts.

Typical uses: When processing large data sets in distributed locations, it becomes more efficient to have the code move to the location of these large data sets; when it is desirable to dynamically customize a local processing node through inclusion of external code.

Style:
Mobile Code

Cautions: Security issues—execution of imported code may expose the host machine to malware. Other cautions—when it is necessary to tightly control the different software versions deployed; when costs of transmission exceed the cost of computation; when network connections are unreliable.

Relations to programming languages or environments: Scripting languages (such as JavaScript, VBScript), ActiveX controls. Grid computing.

In Figure 4-19, one client Web browser downloads code-on-demand in the form of a Lunar Lander game applet via HTTP. A second browser loads a JavaScript Lunar Lander; a third uses some other form that is not detailed. Thus, all the game logic moves to the client machines, freeing the server's computing resources. In the figure as shown, each client machine maintains the game state independently of other clients.

Implicit Invocation

Unlike the previously discussed styles, the two styles below are characterized by calls that are invoked indirectly and implicitly as a response to a notification or an event. Ease of adaptation and enhanced scalability are benefits of the indirect interaction between very loosely coupled components.

Publish-Subscribe. Publish-subscribe takes its name from the analogous relationship between magazine or newspaper publishers and their subscribers. The publisher periodically creates information and the subscriber obtains a copy of the information or at least is informed of its availability. The architectural style reflects this kind of relationship. Several variants exist, differing in the distance between the publisher and the subscribers, and the way in which the relationship is managed.

In a simple publish-subscribe style the publisher maintains a list of subscribers to each of which a procedure call is issued when new information is available. Subscribers register their interest with publishers, providing them with the procedure interface to be used (the call-back) when information is published. Subscribers can also deregister their interest, having the call-back removed from the publisher's subscription list.

The simple publish-subscribe style works well within the context of a simple program. For large-scale, network-based applications, a more sophisticated set of services is needed. Publishers need to advertise the existence of information resources to which others may subscribe, either at system start-up, periodically, or on demand. Subscriptions can no longer take the form of call-back procedures, but involve the use of network protocols. Efficiency concerns argue against point-to-point relationships between publishers and subscribers; intermediate proxies or caches can help performance—just as with physical newspapers, where a paper carrier serves as proxy for the newspaper company, being responsible for delivery of the papers within a small geographic region.

The publish-subscribe style caters to applications where there is a clear delineation between producers and consumers of information. A familiar network-based application of the publish-subscribe style is an online job posting service. Hiring managers post, or publish, the job openings. Meanwhile, job seekers subscribe to the online service to receive notifications of new job postings. While this application works with perhaps only a few notifications per day, other network publish-subscribe applications operate with many notifications per second.

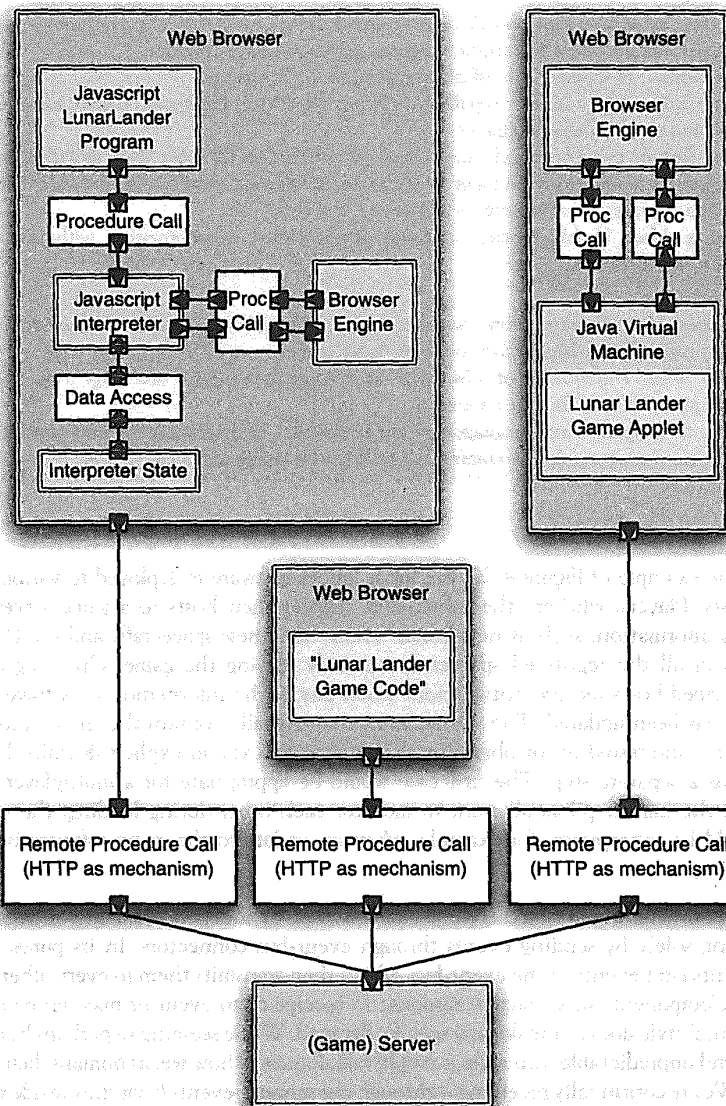


Figure 4-19.
Mobile code:
Lunar Lander as
code-on-demand.

Summary: Subscribers register/deregister to receive specific messages or specific content. Publishers maintain a subscription list and broadcast messages to subscribers either synchronously or asynchronously.
Components: Publishers, subscribers, proxies for managing distribution.

Style:
Publish-Subscribe

Connectors: Procedure calls may be used within programs, more typically a network protocol is required. Content-based subscription requires sophisticated connectors.

Data elements: Subscriptions, notifications, published information.

Topology: Subscribers connect to publishers either directly or may receive notifications via a network protocol from intermediaries.

Variants: Specific uses of the style may require particular steps for subscribing and unsubscribing. Support for complex matching of subscription interests and available information may be provided and be performed by intermediaries.

Qualities yielded: Highly efficient one-way dissemination of information with very low coupling of components.

Typical uses: News dissemination—whether in the real world or online events. Graphical user interface programming. Multiplayer-network-based games.

Cautions: When the number of subscribers for a single data item is very large a specialized broadcast protocol will likely be necessary.

Relations to programming languages or environments: In large-scale systems support for publish-subscribe is provided by commercial middleware technology.

In the example of Figure 4-20, the lunar lander software is deployed to various network hosts. Players, who are the subscribers, register their hosts to a game server that publishes information, such as new Lunar terrain data, new spacecraft, and the (Lunar) locations of all the registered spacecraft currently playing the game. Once registered, the subscribed hosts receive notifications when any of the information they have registered for has been updated. The notifications may actually contain the information the subscriber is interested in, or obtaining the information via an explicit download could proceed as a separate step. The first case would be appropriate for a multiplayer game in which the Lander pilots all want to monitor each other during landing; the second case would be appropriate, for example, when an update to the game software is made available.

Event-Based. The event-based style is characterized by independent components communicating solely by sending events through event-bus connectors. In its purest form, components emit events to the event-bus, which then transmits them to every other component. Components may react in response to receipt of an event or may ignore it; the architectural style does not make any specific demand. While seeming to perhaps be rather chaotic and unpredictable, this style is roughly analogous to how we, as humans, behave in society. We are continually receiving—through our senses—events from the outside world: Some we react to and some we ignore. Similarly, we are continually emitting events into the world, such as by speaking. Again, sometimes they elicit a response from others and sometimes not—such as when you are talking to your teenage son.

For efficiency reasons, the pure form of event-based architectures is seldom used; it is more efficient to only distribute events to those components that express an interest in them. With this optimization, the style becomes similar to publish-subscribe. The distinctive character of the event-based style, however, is that there is no classification of components into publishers and subscribers; all components potentially both emit and

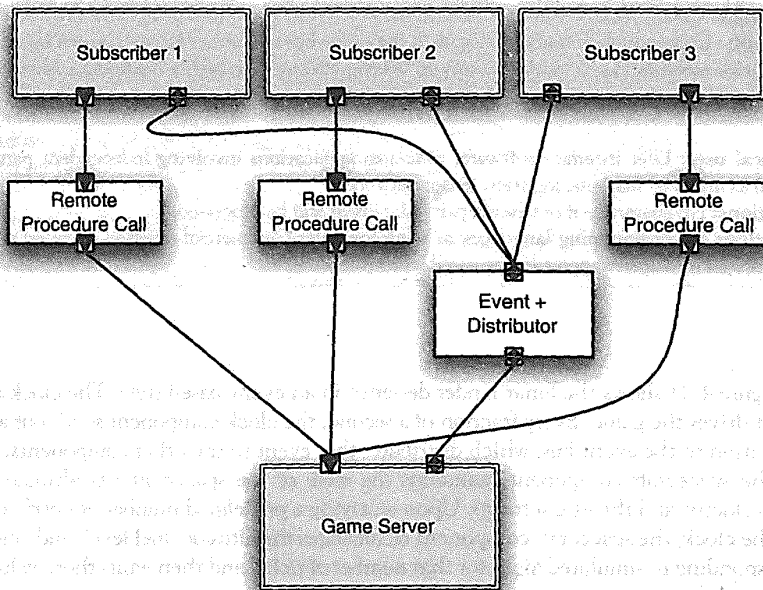


Figure 4-20.
Lunar Lander in
publish-subscribe.

receive events. With the event-based style, the optimization of event distribution is the responsibility of the connectors; registration of interest in particular events is only handled by the connectors. Similarly, replication of events for delivery to multiple recipients may take place late in the transmission process and would be transparent to both the event emitter and the event receiver. Finally, event distribution may be handled either on a push or pull basis. In the case of pull, or polling, event recipients can query the connector to see if a new event is available. Such a query could either be blocking (the component waits until an event is available) or nonblocking (the component returns immediately, either with a new value—if available—or not).

The event-based style is highly suited to strongly decoupled concurrent components, where at any given moment a component either may be creating information of potential interest to others or may be consuming information. A classic example usage is in financial markets: Independent companies (traders) may want to know the latest price of a commodity being sold in London; similarly, when that trader completes a deal, the results of that trade represent information of interest to others around the world.

Summary: Independent components asynchronously emit and receive events communicated over event buses.

Components: Independent, concurrent event generators and/or consumers.

Connectors: Event bus. In variants, more than one may be used.

Data elements: Events—data sent as a first-class entity over the event bus.

Style:
Event-Based

Topology: Components communicate with the event-buses, not directly to each other.
Variants: Component communication with the event-bus may either be push or pull based.
Qualities yielded: Highly scalable, easy to evolve, effective for highly distributed, heterogeneous applications.
Typical uses: User interface software, wide-area applications involving independent parties (such as financial markets, logistics, sensor networks).
Cautions: No guarantee if or when a particular event will be processed.
Relations to programming languages or environments: Commercial message-oriented middleware technologies support event-based architectures.

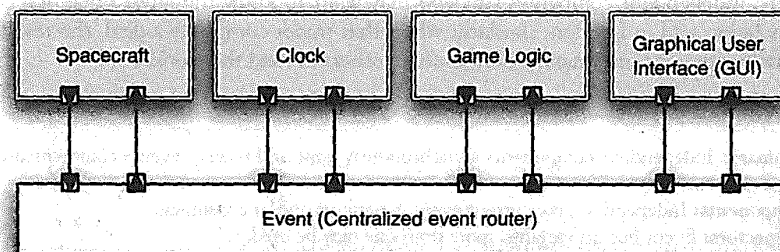
Figure 4-21 shows the lunar lander designed in an event-based style. The clock component drives the game. Every fraction of a second, the clock component sends out a tick notification to the event bus, which distributes that event to the other components.

The spacecraft component maintains the state of the spacecraft (its altitude, fuel level, velocity, and throttle setting). Upon receiving a predefined number of notifications from the clock, the spacecraft component recalculates the altitude, fuel level, and velocity (corresponding to simulated flight for that number of ticks) and then emits those values to the event bus.

The graphical user interface (GUI) component drives the game player's display. Receipt of events providing the spacecraft's altitude, fuel, and velocity causes the GUI component to update its display based on those values. The GUI component also obtains new burn rate settings from the user; when this happens the component emits a notification of this new value onto the event bus. Upon receipt of this notification the Spacecraft component updates its internal model of the spacecraft and, as always, emits the updated state back to the bus.

The game logic component, receiving both information about the state of the spacecraft and the amount of time that has passed (by counting clock ticks), determines whether the game is over, and if so, what the final condition of the spacecraft was when the game ended and hence the player's score.

Figure 4-21.
Lunar Lander in
the event-based
style.



"Implicit invocation" describes the execution model of both the publish-subscribe and event-based styles. An action, such as invocation of a procedure, may be taken as the result of some activity occurring elsewhere in the system, but the two activities are distant from one another. Though there is a causal relationship, the invocation is indirect, since there is no direct coupling between the components involved. The invocation is implicit, since the causing component has no awareness that its action ultimately causes an invocation elsewhere in the system.

**Implicit
Invocation**

Peer-to-Peer

The peer-to-peer (P2P) architectural style consists of a network of loosely coupled autonomous peers, each peer acting both as a client and a server. Peers communicate using a network protocol, sometimes specialized for P2P communication—such was the case for the original Napster and Gnutella file-sharing applications. Unlike the client-server style where state and logic are centralized on the server, P2P decentralizes both information and control.

Absence of centralization makes resource discovery an important issue for P2P applications. In a pure P2P application, queries for information are issued to the network of peers at large; requests propagate until the information is discovered or some propagation threshold is passed. If the desired information is located, the peer obtains the direct address of that peer and contacts it directly. This style is limited by the network algorithm used to query the system and the network bandwidth available. Hybrid applications optimize this process by having certain peers play special roles, either for locating other peers or for providing directories locating information. The original Napster, for example, was not a true *peer-to-peer* application because it used a centralized server for indexing music and locating other peers. Although the term peer-to-peer has been popularized by file-sharing applications, it enables a host of applications such as business to business commerce, chat, remote collaboration, and sensor networks.

Summary: State and behavior are distributed among peers that can act as either clients or servers.

Components: Peers—-independent components, having their own state and control thread.

Connectors: Network protocols, often custom.

Data elements: Network messages.

Topology: Network (may have redundant connections between peers); can vary arbitrarily and dynamically.

Qualities yielded: Decentralized computing with flow of control and resources distributed among peers. Highly robust in the face of failure of any given node. Scalable in terms of access to resources and computing power.

Typical uses: Where sources of information and operations are distributed and network is ad hoc.

Cautions: When information retrieval is time critical and cannot afford the latency imposed by the protocol. Security—P2P networks must make provision for detecting malicious peers and managing trust in an open environment.

**Style:
Peer-to-Peer**

In the example of Figure 4-22, a group of lunar lander spacecraft are on their way to land in different parts of the moon. Multiple Landers are used in this example to highlight that the P2P architectural style is designed to support interaction between highly autonomous components, and especially application contexts where the number of participating components may vary over time. Lunar Lander 1 (LL1) wants to find out if another spacecraft has already landed in a specific area, so that collisions can be avoided. Each of the spacecraft is configured to communicate with the others using a P2P protocol over a network. A spacecraft will only be able to communicate with others that are within a specified, limited distance.

Lunar Lander 1 will go through the following steps to obtain the information:

1. At time T_0 , LL1 queries for available spacecraft, that is, spacecraft within communication range.
2. Only LL2 responds.
3. LL1 queries if LL2 has the requisite information.
4. Since LL2 does not have the information, it passes the query on to its adjacent communicating node, LL3 (that is, assume that LL2 and LL3 already have been communicating and do not have to discover each other).
5. Since LL3 does not have the information, it passes the query on to its communicating nodes, LL4, LL6, and LL7.
6. LL7 responds to LL3 that it has the desired information and passes the information requested to LL3.
7. LL3 passes the response back to LL2 and subsequently to LL1.

If at some later time T_n , LL7 comes within communication range, LL1 could directly contact it and query it for additional information, as shown at the bottom of the figure.

4.3.5 More Complex Styles

More complex styles than those discussed above have arisen from efforts to exploit design knowledge gained from substantial experience. The complexity reflects (a) provision of greater benefits, and (b) specialization to certain application contexts. As examples, the C2 and distributed objects styles discussed below combine elements of several of the preceding styles to yield powerful design tools capable of handling challenging problems. They both have resulted from an extended period of engineering experimentation, in which numerous design trade-offs were made.

The C2 (components and connectors) style grew out of a desire to obtain the benefits of the model-view-controller pattern in a distributed, heterogeneous platform setting. Concepts from layered and event-based architectures were added. The resulting style, though originally developed to support graphical user interface applications, was found to be beneficial in a wide variety of applications—indeed more so outside the domain of GUIs than within. C2's primary role in this presentation is showing how elements of many styles may be judiciously combined to meet a variety of needs.

The distributed objects style has its roots in the object-oriented style. As with C2, its developers sought to extend the benefits of a core style to a distributed heterogeneous world.

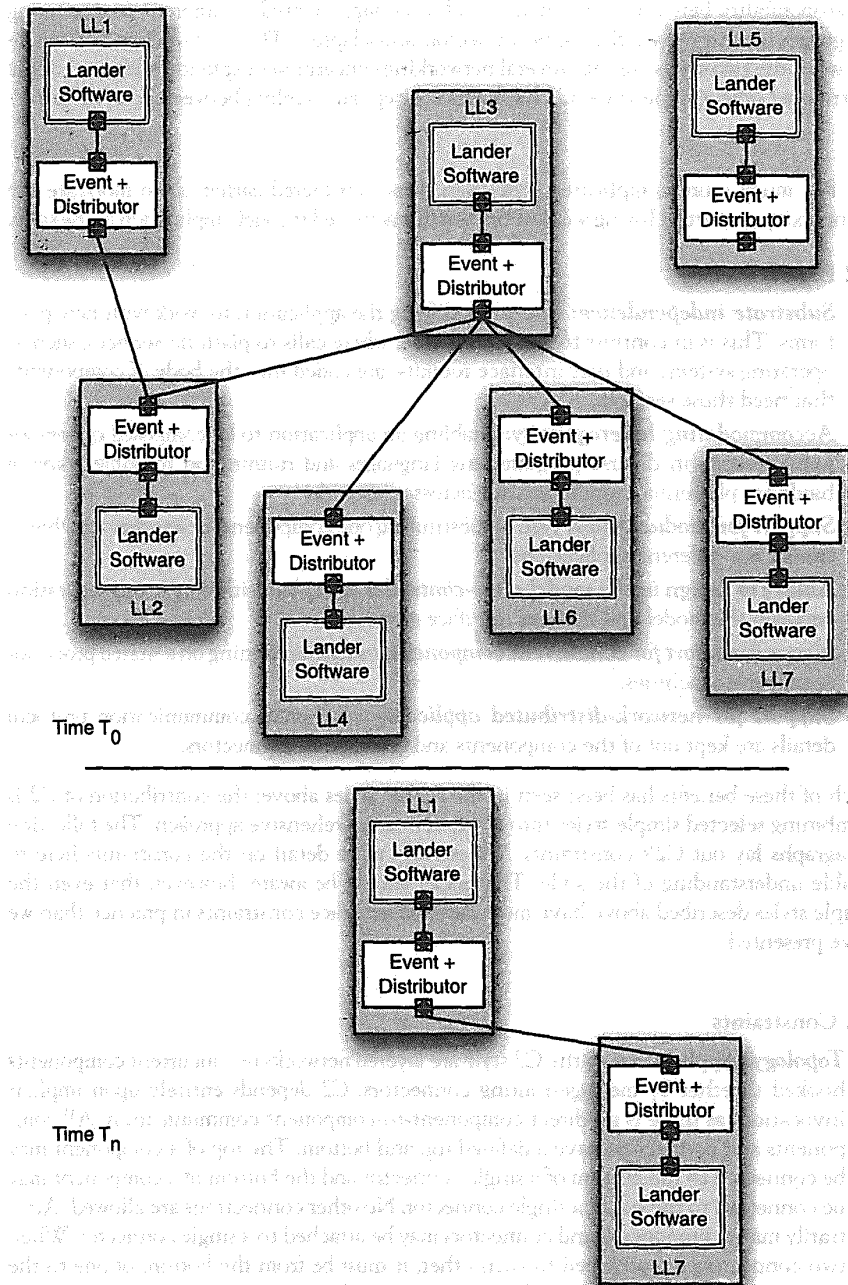


Figure 4-22.
Lunar Lander as
a P2P
application.

Interoperability between components or objects implemented in different programming languages is achieved via the use of connectors and adapters. The most obvious differences between the two styles are that several networking concerns are explicitly addressed in the distributed objects style while C2 has lower conceptual coupling between components.

C2

C2 is a much more complicated style than those considered earlier, so to motivate the complexity we start by listing some of the benefits achieved through application of the style.

C2 Benefits

- **Substrate independence:** ease in modifying the application to work with new platforms. This is in contrast to the typical case, where calls to platform services, such as operating systems and user interface toolkits, are coded into the body of components that need those services.
- **Accommodating heterogeneity:** enabling an application to be composed of components written in diverse programming languages and running on multiple, varying hardware platforms, communicating across a network.
- **Support for product lines:** ease of substituting one component for another to achieve similar but different applications.
- **Ability to design in the model-view-controller style:** but with very strong separation between the model and the user interface elements.
- **Natural support for concurrent components:** whether running on a shared processor or multiple machines.
- **Support for network-distributed applications:** wherein communication protocol details are kept out of the components and confined to connectors.

Each of these benefits has been seen in the simple styles above; the contribution of C2 is combining selected simple styles into a coherent comprehensive approach. The following paragraphs lay out C2's constraints. We provide more detail on the constraints here to enable understanding of the style. The reader should be aware, however, that even the simple styles described above have more detailed interface constraints in practice than we have presented.

C2 Constraints

Topology: Applications in the C2 style are layered networks of concurrent components hooked together by message-routing connectors. C2 depends entirely upon implicit invocation, as there is no direct component-to-component communication. All components and connectors have a defined top and bottom. The top of a component may be connected to the bottom of a single connector and the bottom of a component may be connected to the top of a single connector. No other connections are allowed. Arbitrarily many components and connectors may be attached to a single connector. When two connectors are attached to each other, it must be from the bottom of one to the top of the other. These rules induce layering, which promotes substrate independence and component substitutability.

Message-based communication: All communication between components is achieved by exchanging messages. Messages are classified either as requests (a specific request for a service to be performed) or notifications (statements of facts about the system, such as the change in value of some object). This requirement is suggested by the asynchronous nature of applications wherein users and the application perform actions concurrently and at arbitrary times, and where various components in the architecture must be notified of those actions. This constraint also promotes mutual independence of the components.

Message flow and substrate independence: Requests may only flow upward in an architecture; notifications flow downward. This rule induces limited visibility or “substrate independence”—a component within the hierarchy can only be aware of components above it and is completely unaware of components that reside beneath it. Since downward communication is limited to the emitting of notifications, a component cannot know which components, if any, below it will react to receipt of a notification.

Interfaces: Each component has a top and bottom domain. The top domain specifies the set of notifications to which a component may react, and the set of requests that the component emits up an architecture. The bottom domain specifies the set of notifications that this component emits down an architecture, and the set of requests to which it responds.

One important note concerns the apparent dependence of a given component on its superstrate, that is, the components above it, since a component can issue a specific request upward in an architecture. If each component is built so that its top domain closely corresponds to the bottom domains of those components with which it is specifically intended to interact in the given architecture, its reusability value is greatly diminished; it can only be substituted by components with similarly constrained top domains. To avoid this, C2 introduces the notion of domain translation. Domain translation is a transformation of the requests issued by a component into the specific form understood by the recipient of the request, as well as the transformation of notifications received by a component into a form it understands.

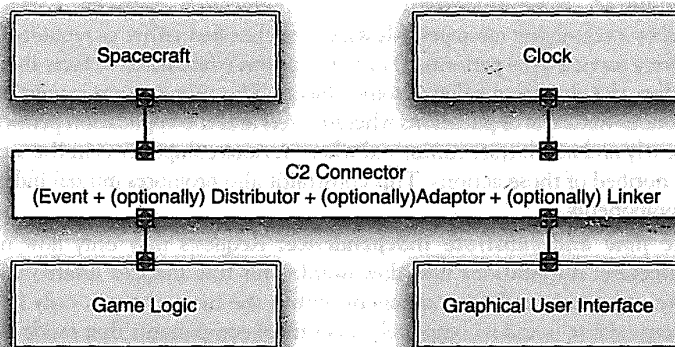
C2 supports compositionality, or hierarchical composition, whereby an entire architecture becomes a single component in another, larger architecture.

Each component may have its own thread(s) of control. This simplifies modeling and programming of multicomponent, multiuser, and concurrent applications, and enables exploitation of distributed platforms.

Finally, there can be no assumption of a shared address space among components. Any premise of a shared address space would be unreasonable in an architectural style that allows composition of heterogeneous, highly distributed components, developed in different languages, with their own threads of control, internal structures, and domains of discourse.

A simple C2 architecture for the lunar lander application is shown in Figure 4-23. With respect to the use of events, it operates largely like the solution shown in Figure 4-21, the event-based style. Here, however, both the spacecraft and clock components are *guaranteed by the architecture* to be completely unaware of the presence of the game logic and GUI components.

Figure 4-23.
Lunar Lander in
the C2 style.



Style: C2

Summary: An indirect invocation style in which independent components communicate exclusively through message routing connectors. Strict rules on connections between components and connectors induce layering.

Components: Independent, potentially concurrent message generators and/or consumers.

Connectors: Message routers that may filter, translate, and broadcast messages of two kinds—notifications and requests.

Data elements: Messages—data sent as first-class entities over the connectors. Notification messages announce changes of state. Request messages request performance of an action.

Topology: Layers of components and connectors, with a defined top and bottom, wherein notifications flow downward and requests upward.

Additional constraints imposed:

- All components and connectors have a defined top and bottom. The top of a component may be attached to the bottom of a single connector and the bottom of a component may be attached to the top of a single connector. No direct component-to-component links are allowed; there is, however, no bound on the number of components or connectors that may be attached to a single connector. When two connectors are attached to each other, it must be from the bottom of one to the top of the other.
- Each component has a top and bottom *domain*. The top domain specifies the set of notifications to which a component may react and the set of requests that the component emits up an architecture. The bottom domain specifies the set of notifications that this component emits down an architecture and the set of requests to which it responds.
- Components may be hierarchically composed, where an entire architecture becomes a single component in another, larger architecture.
- Each component may have its own thread(s) of control.
- There can be no assumption of a shared address space among components.

Qualities yielded:

- Substrate independence: Ease in moving the application to new platforms.
- Applications composable from heterogeneous components running on diverse platforms.

- Support for product lines.
- Ability to program in the model-view-controller style, but with very strong separation between the model and the user interface elements.
- Support for concurrent components.
- Support for network-distributed applications.

Typical uses: Reactive, heterogeneous applications. Applications demanding low-cost adaptability.

Cautions: Event-routing across multiple layers can be inefficient. Overhead high for some simple kinds of component interaction.

Relations to programming languages or environments: Programming frameworks are used to facilitate creation of implementations faithful to architectures in the style. Support for Java, C, Ada.

A more instructive application of C2 is presented in Figure 4-25, in which a KLAX-like game⁷ is supported. A screenshot from the game is shown in Figure 4-24. The essence of the game is that colored tiles fall from the chutes at the top of the user's screen. A palette is used to move horizontally across the screen; it can catch the tiles as they fall from the chutes. By inverting the palette, the tiles can be dropped into the wells below. By matching three or more tiles of the same color, horizontally, vertically, or diagonally across the wells, points are scored and the tiles subsequently removed from the game board. If tiles are not caught by the palette, lives are lost. Lives are also lost if the wells are allowed to overflow.

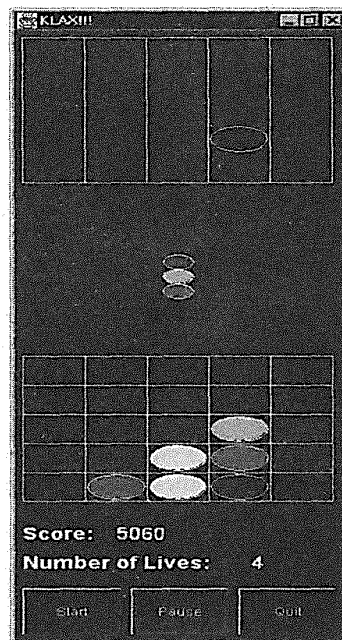
The components that make up the KLAX-like game can be divided into three logical groups. At the top of the architecture are the components that encapsulate the game's state plus the clock. These components are placed at the top because the game state is vital for the functioning of the other two groups of components. The game state components receive no notifications, but respond to requests and emit notifications of internal state changes. In a sense, these components play the role of servers, or perhaps of blackboards. Notifications are directed to the next level, where they are received by both the game logic components and the artist components.

The game logic components request changes of game state in accordance with game rules, and interpret game state change notifications to determine the state of the game in progress. For example, if a tile is dropped from a chute, Relative Position Logic determines if the palette is in a position to catch the tile. If so, a request is sent to Palette, adding the tile to the palette. Otherwise, a notification is sent that a tile has been dropped. This notification is detected by Status Logic, causing the number of lives to be decremented. The logic components perform their functions in keeping with receipt of a particular number of notifications from the clock.

The artist components receive notifications of game-state changes, causing them to update their depictions. Each artist maintains the state of a set of abstract graphical

⁷KLAX was a video game developed by Atari Games Corporation in 1990. KLAX is now a trademark of Midway Games West Inc.

Figure 4-24.
Screenshot of
example
KLAX-like
game.



objects that, when modified, send state change notifications in the hope that a lower-level graphics component will render them on the screen. Tile Artist, for instance, provides a flexible presentation for tiles. Tile Artist intercepts any notifications about tile objects and issues notifications about more concrete drawable objects. For example, a Tile Created-notification received might result in issuance of a Rectangle Created-notification. The Layout Manager component receives all notifications from the artists and determines coordinates to ensure that the game elements are drawn in the correct two-dimensional juxtaposition. The Graphics Binding component receives all notifications about the state of the artists' graphical objects and translates them into calls to a window system. User events, such as a key press, are translated into requests that are sent upward to the artist components.

The application of the C2 style to this design yields several benefits. One is the easy creation of related, but different, games. By substitution of three components the game can be transformed into one where letters instead of tiles fall down the chutes, and the game logic is based upon spelling words in the wells. The components involved are the tile artist, tile match logic, and next tile placing logic, which become, respectively, the letter artist, word match logic, and next letter placing logic components. Similarly confined and easy substitutions or additions of components result in network-based multiplayer games, a high-score version, and so on. The application can run over network boundaries, the user interface toolkit can be swapped out without modification to the various artists, and components can execute concurrently.

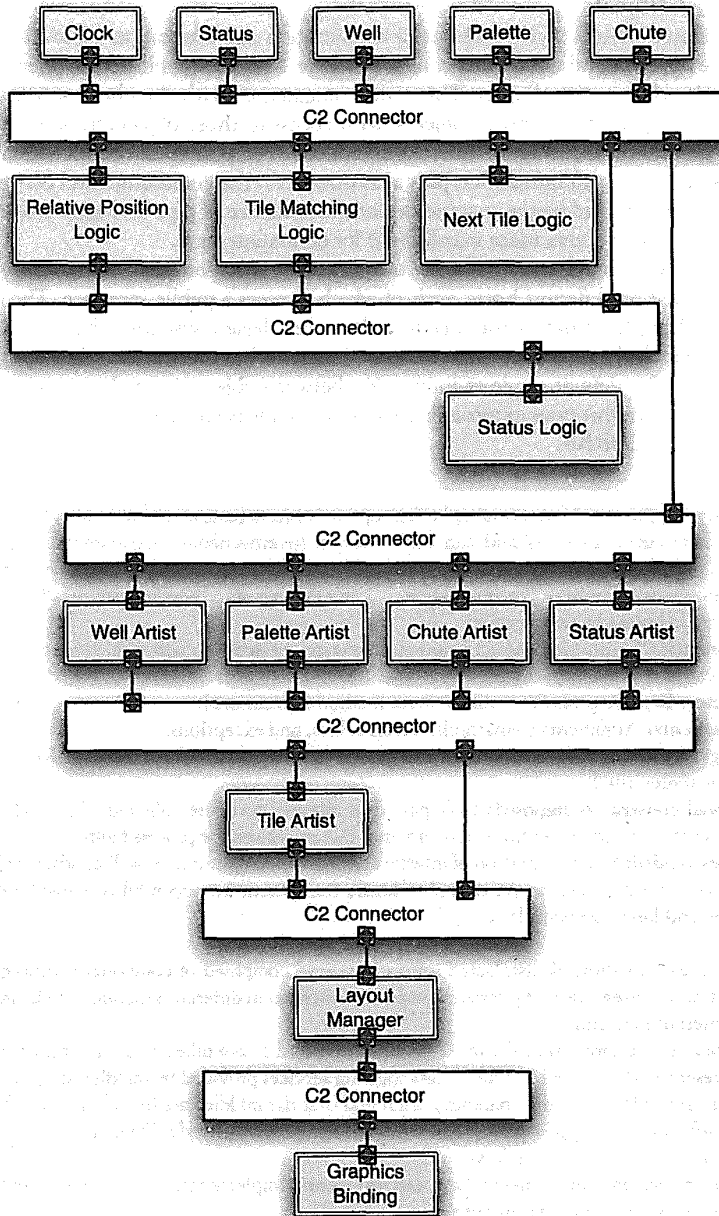


Figure 4-25.
A KLAX-like
video game in the
C2 style.

Distributed Objects

The distributed objects style represents a combination and adaptation of several simpler styles. The fundamental vocabulary comes, of course, from the simple object-oriented style considered earlier in the chapter. This style is augmented with the client-server style to provide the notion of distributed objects, with access to those objects from, potentially, different processes executing on different computers. The posited need for cross-machine and cross-language communication puts a constraint on the communication between the processes, however, and hence to some extent the influence of pipe-and-filter is seen, with serialization of parameters (data marshaling) for communication.

In this style, therefore, objects, in the standard object-oriented terminology sense, are instantiated on different hosts, each of which exposes a public interface. The objects can be anything from data structures to million-line legacy systems. The interfaces are special in that all the parameters and return values must be serializable so they can go over the network. The default mode of interaction between objects is synchronous procedure call, although asynchronous extensions are found in some particular versions of distributed objects, such as CORBA.

Style: Distributed Objects

Summary: Application functionality broken up into objects (coarse- or fine-grained) that can run on heterogeneous hosts and can be written in heterogeneous programming languages. Objects provide services to other objects through well-defined provided interfaces. Objects invoke methods across host, process, and language boundaries via remote procedure calls (RPCs), generally facilitated by middleware.

Components: Objects (software components exposing services through well-defined provided interfaces).

Connector: Remote procedure calls (remote method invocations).

Data elements: Arguments to methods, return values, and exceptions.

Topology: General graph of objects from callers to callees; in general, required services are not explicitly represented.

Additional constraints imposed: Data passed in remote procedure calls must be serializable. Callers must deal with exceptions that can arise due to network or process faults.

Qualities yielded: Strict separation of interfaces from implementations as well as other qualities of object-oriented systems in general, plus mostly-transparent interoperability across location, platform, and language boundaries.

Typical uses: Creation of distributed software systems composed of components running on different hosts. Integration of software components written in different programming languages or for different platforms.

Cautions: Interactions tend to be mostly synchronous and do not take advantage of the concurrency present in distributed systems. Users wanting services provided by middleware (network, location, and/or language transparency) often find that the middleware induces the distributed objects style on their applications, whether or not it is the best style. Difficulty dealing with streams and high-volume data flows.

Relations to programming languages or environments: Implementations in almost every programming language and environment.

CORBA. CORBA is a standard for implementing middleware that supports development of applications composed of distributed objects. Many software development practitioners

will have experience with, or at least heard of, CORBA technology. However, CORBA's relationship to software architecture and architectural styles is not always clear. While CORBA is no longer as popular a technology as it once was, it is emblematic of a collection of technologies (such as DCOM and RMI) that seek to support distributed objects, and hence worthy of study.

The basic idea behind CORBA is that an application is broken up into objects, which are effectively software components that expose one or more provided interfaces. In CORBA, these provided interfaces are specified in terms of a programming-language- and platform-neutral notation called the Interface Definition Language (IDL). IDL closely resembles the way class interfaces are specified in an object-oriented programming language. An IDL description of the interface for the data store component of a lunar lander system might look like the following.

```
interface IDataStore{
    double getAltitude();
    void setAltitude(in double newAltitude);
    double getBurnRate();
    void setBurnRate(in double newBurnRate);
    void getStatus(out double altitude,
                  out double burnRate,
                  out double velocity,
                  out double fuel,
                  out double time);
}
```

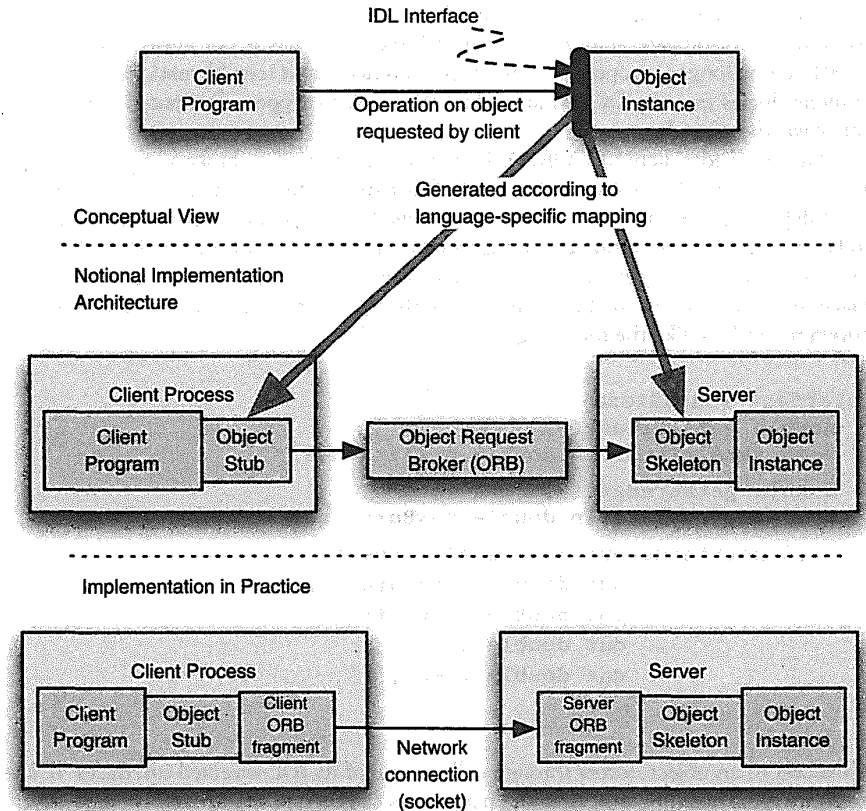
All access to an object occurs through calls to one of its IDL-specified interfaces. IDL is strongly typed, meaning that calls and parameters can be type-checked at compile time.

Figure 4-26 illustrates how CORBA connects objects across machine, language, and process boundaries. The top portion of the diagram shows the conceptual, or logical view, which should be familiar to anyone who has written an object-oriented program. A client program obtains a pointer to an object that can perform a service for it, here labeled the Object Instance. After obtaining this pointer, it makes calls on the object instance through one of its interface methods.

Within the context of a single programming language, in a single process running on a single machine, making an object call like this is straightforward. The pointer in this case is a direct memory reference, and the call represents a simple synchronous procedure call. When language, process, and machine boundaries are involved, things become more complicated, and this is the problem that CORBA attempts to overcome. Implementations of CORBA are accompanied by tools called IDL compilers that can target specific platform and programming language combinations (such as C++ on Windows). When an IDL interface is run through an IDL compiler, two code artifacts are created, known as an *object stub* and an *object skeleton*.

An object skeleton is an object in the target programming language/platform that provides empty implementations for each method defined in the IDL interface. IDL types are translated into local programming language types. The developer is then responsible for

Figure 4-26.
Notional
CORBA
diagram.



implementing these methods in the target programming language. These implementations make up the object instance, also known as the *true object*.

The object stub is the complement of the object skeleton. It is an object that provides implementations for each method in the IDL interface, again in the target programming language. These method implementations do not actually perform the services provided by the true object, however. Instead, when a method on the stub is called, the calling parameters are encoded into a block of binary data in a process called *serialization* or *marshaling*. The stub then sends this block of data across machine or process boundaries, using a network or another interprocess communication (IPC) mechanism, to the object skeleton. The software that facilitates this is known in CORBA as the *object request broker*, or ORB. Depending on the CORBA implementation, this may be a separate piece of software, or it may be part of the stub and skeleton. The skeleton receives the call data, makes the corresponding invocation on the object instance, and then marshals the return value or exception for return to the stub. The stub then returns this data to the caller, which in the figure is the client program.

In this way, a client program can invoke a remote object without explicit knowledge that it is remote. From the client program's perspective, the object stub is providing the service it needs. The object stub, however, is not really providing the service. Instead, it is sending the call data over the network to the skeleton, which calls the object instance—the true object—to perform the actual service. Data is returned by the same path in reverse.

The last part of the puzzle is how the client program/object stub pair obtain a pointer or reference to the object skeleton/object instance pair in the first place. This can be done in a number of ways, most commonly by a lookup or naming service known as an object request broker (ORB) that maintains a directory of objects running on hosts and allows clients to look them up by name. The conceptual placement of the ORB is shown in the middle of Figure 4-26. In practice, the ORB is distributed among the client and server processes, as illustrated in the bottom of the figure.

CORBA does the best it can to ensure that client and server object implementations are not aware that the call between them is traversing a language, process, or machine boundary. However, these boundaries never can be masked completely. There are two main differences between a local and a remote procedure call. First, all data that traverses a remote procedure call must be serializable. That is, the stub and skeleton must be able to transform the data into a block of binary data for transmission over a network. Parameters that contain pointers to arbitrary blocks of memory on the local machine or control objects like threads cannot be serialized, and thus cannot be passed or returned in a remote procedure call. Second, remote procedure calls suffer from a much wider variety of potential failures than local procedure calls. In a local procedure call, it is assumed that the call itself—the transfer of control and data from caller to callee—cannot fail. In a remote procedure call, network problems and dead processes can cause the call itself to fail. In these cases, a run-time exception is raised by the call, but such exceptions would not have been expected in the context of a local procedure call. (Additional problems with network-based applications are discussed in Chapter 11.)

Although remote procedure calls are in some ways more limited and fragile than local ones, they can be more flexible as well. Calls can be redirected or logged by the object request broker without any changes to the client or server objects. New server objects and even new hosts can be dynamically created and come online during application run-time, and then be looked up by later clients. This imparts a measure of dynamism to applications built using CORBA.

Figure 4-27 shows a simple application of CORBA in the design of a two-pilot trainer version of Lunar Lander. In this version, two independent user clients, Trainer and Trainee, can view the state of the Lander during its descent and both clients can adjust the fuel consumption rate, r . In addition, a third client, Houston, monitors the state of the Lander and the two pilots. The state of the Lander is maintained as a CORBA object on a remote server, the Lander Simulation System. The clients manipulate the burn rate by making calls through the CORBA services. Each pilot client also maintains the state of its pilot. Thus, when Houston views the altitude of the Lander, the request is routed to the server. However, when Client Houston views the heart rate of the pilots, the call is routed to both pilot clients.

The description here of CORBA has been quite brief, and the example simplistic. In practice, CORBA offers a rich set of services for managing distributed objects and constructing sophisticated distributed architectures. Most CORBA implementations are

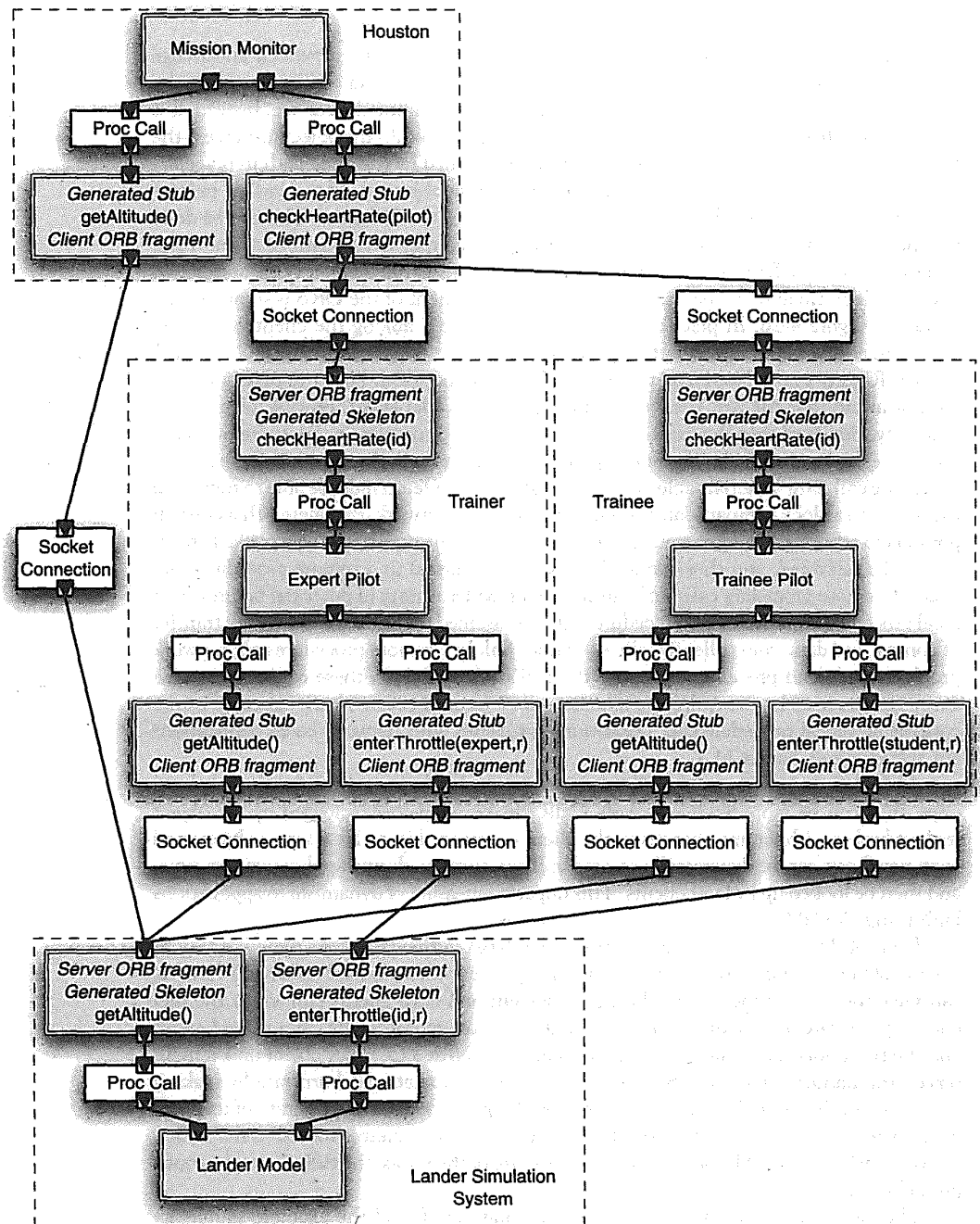


Figure 4-27. Training version of Lunar Lander, in CORBA. Two pilots can have their hands on the controls.

accompanied by CORBA services, objects and interfaces that can be called to deal with object life cycles, relationships, persistence, externalization, naming, trading, events, transactions, concurrency, properties, queries, security, licensing, versioning, notification, and so on.

Selecting CORBA as the basis for a software development project is a double-edged sword. On one hand, using CORBA allows systems to be implemented using components written in different languages on different platforms, and the CORBA middleware automatically masks most of the differences. For developers faced with integrating modern technologies with legacy systems, the allure is almost too good to pass up. However, using CORBA comes with a price. The use of CORBA provides these interoperability benefits, but it also induces applications to be built in the distributed objects style.

Distributed objects is not an ideal style for every application. Drawbacks include for example, that components in a distributed objects style are required to explicitly specify provided interfaces, but not to specify required interfaces. Dependencies between objects may thus be deeply ingrained. Objects in the CORBA world are constantly created, linked, unlinked, and destroyed.¹ This makes it difficult to understand the configuration of a CORBA application at any given time. Distributed object interactions tend to be synchronous call-return, and many such systems have difficulty with asynchronous invocations or data that travels in streams.

The Big Ball of Mud architectural style has been described by Brian Foote and Joseph Yoder as the most common architectural style of all (Foote and Yoder 1997). The style imposes no constraints, other than "get the job done." It offers the benefits of, "We didn't have to think too much", and "It'll work for now. I hope." Software in the BBoM style is haphazard and thrown together. Such systems often grow by accretion: New bits of code are stuck onto preexisting code to meet new demands, without discipline, plan, or care. The BBoM style is an important style for designers to keep in mind for one simple but critical reason: "If you can't articulate why your application is not a big ball of mud, then it is."

**The Big Ball
of Mud Style**

4.3.6 Discussion: Patterns and Styles

Taking a step back from the myriad details associated with the styles and patterns discussed above, common characteristics are evident. Fundamentally, styles and patterns reflect experience and codify knowledge gained from that experience. Creating a good style entails reflection on experience, abstraction of the knowledge from the details, and possibly generalization. Thus styles are "refined experience in action."

Despite the fact that styles represent hard-won wisdom in system development, their use is sometimes resisted. Why? Because the very nature of styles, the adherence to a style during design, demands that the designer work within the set of constraints that comprise the style. Somehow the very notion of constraining the designer seems counterproductive and limiting. The curious, nonintuitive character of styles and designing, however, is that through the adherence to constraints, great freedom and effectiveness in design is achieved.

The Freedom of Constraints

When viewed as a set of constraints, styles limit the designer in several ways:

- The amount of detail or the number of concepts that the designer is allowed to deal with at any one time may be limited (for example, in three-tier architectures).
- The way elements of the design are allowed to interact may be restricted (for example, in event-based architectures, or in CORBA-based systems).
- There may be specific obligations on what particular elements must do—and not do (for example, requiring inter-component communication to be strictly mediated by first-class connectors).
- The style may constrain the solution of some subproblems to be addressed in a less than optimal way (for example, pipe-and-filter always requires character stream communication, even when complex data structures are communicated). In such a case, the benefits of consistent style, with its understandability and ability to be reused in standard, expected ways, outweighs any other limited benefit.

Where then does the freedom and effectiveness of styles come from? Numerous sources are apparent:

- Styles restrict one's focus, and thus create a space of issues that the design does not have to be concerned about. The styles free up thought space and direct attention to the essentials.
- Styles guarantee the applicability of particular analysis techniques.
- The constraints enable the use of code generation and framework implementation strategies—since patterns of interaction are regularized they can be automatically and/or efficiently supported in standard ways.
- Styles provide invariants—assertions that the designer can always count on (“As long as I follow the rules then I know the following must be true.”).
- Styles make communication more efficient: Once project members know the rules, they form a baseline for communicating the added-value details.
- Similarly, styles promote understanding of design decisions after the fact, such as when the software is taken over by a new development team.

Combination, Modification, and Creation of Styles and Patterns

While studying the long list of architectural styles in the preceding pages may have been exhausting, the list is certainly not exhaustive. A point we have stressed is that styles capture experience and distill it to an essence that is useable in new development situations. Seen this way, each application domain may have a specific style associated with it. More commonly, however, styles appropriate for a particular domain will be aggregations or combinations of simpler styles.

The combination of styles is motivated by the desire to obtain multiple benefits in design. A good example of a composite, and specialized style, is the C2 style discussed in Section 1.3.5. C2 is a combination of model-view-controller, layered systems, and event-based styles. REST, an even more complicated example, will be discussed in Chapter 11. REST was introduced at the beginning of Chapter 1, but was discussed there without reference to how it was developed.

Knowing that styles can be combined, specialized, or developed from scratch leads to the danger of creating new styles unnecessarily or unwisely. Any designer could claim, with some justification, that his experience is what matters most to him, and hence codification of a style that represents that experience is important. Keep in mind, however, that one's own experience needs to be traded off against the aggregate experience of other designers within a company or application domain. Moreover formulation of yet-another-architectural-style (YAAS, indeed) negates one of the advantages of styles: the promotion of effective communication between designers. The lures of expediency and "good enough" seldom outweigh the genuine benefits of carefully applying established, well-known styles.

If, after consideration of substantial experience, a group decides to formulate a new style to capture that experience and their insights, it should be done with an eye to capturing only what is truly central, and avoiding incidental matters. "Less is more" should prevail until adequate validation is obtained for the newly formulated approach. An important goal for such an exercise should be a clear articulation of what invariants are gained as the result of adding constraints.

A brief summary of the styles considered in this chapter appears in Table 4-1. Each of the styles is listed in the left column. A capsule summary of the style is then presented, followed by advice on when to use it, and when to avoid it. Though this table is long, the advice and summaries are extremely brief, and hence omit the nuance and detail that are needed to fully guide their use. Hence this table should only be used as a first cut in determining what styles might be appropriate for a given design challenge. The more detailed tables and discussions which have appeared earlier in this chapter should then be referenced to assess the candidate styles.

4.3.7 Design Recovery

So far we have discussed situations in which an architect has, or has access to, prior experience in designing a particular kind of system for an understood domain. If this is not the case, the problem faced by the architect falls in the province of unprecedented design. However, before we consider approaches for dealing with unprecedented design, we will consider one more source of experience—design recovery.

Different software design methods clearly differ in the details of how developers arrive at a software system. However, to one extent or another, they all rely on the assumption that a software project starts with some form of requirements gathering and specification, reaches its major milestone with system implementation and delivery, and then continues, possibly indefinitely, into an operation and maintenance phase. The software system's architecture is in many ways the linchpin of this process: It is supposed to be an effective reification of the system's technical conception and to be faithfully reflected in the system's implementation. Furthermore, the architecture is meant to guide system evolution, while also being updated in the process.

Sadly, for many systems this turns out to be an idealized picture. Some systems are hacked without careful consideration or documentation of architectural concerns. Other systems are designed carefully and those initial designs are documented, but repeated modifications are subsequently made with little concern for maintaining intellectual coherence, resulting in substantial architectural erosion. Software evolution certainly requires in-depth understanding of an application, its complexity, its overall architecture, its major components, and their interactions and dependencies. Unfortunately, many of these requirements

Table 4-1
A Comparison
Table of
Architectural
Styles

Style Category	Name	Summary	Use It When ...	Avoid It When ...
Language-influenced styles				
Main program and subroutines		Main program controls program execution, calling multiple subroutines.	... application is small and simple.	... complex data structures needed (because of lack of encapsulation). ... future modifications likely.
Object-oriented		Objects encapsulate state and accessing functions.	... close mapping between external entities and internal objects is sensible. ... many complex and interrelated data structures.	... application is distributed in a heterogeneous network. ... strong independence between components necessary. ... very high performance required.
Layered				
Virtual machines		Virtual machine, or a layer, offers services to layers above it.	... many applications can be based upon a single, common layer of services. ... interface service specification resilient when implementation of a layer must change.	... many levels are required (causes inefficiency). ... data structures must be accessed from multiple layers.
Client-server		Clients request service from a server.	... centralization of computation and data at a single location (the server) promotes manageability and scalability. ... end-user processing limited to data entry and presentation.	... centrality presents a single-point-of-failure risk. ... network bandwidth limited. ... client machine capabilities rival or exceed the server's.
Dataflow Styles				
Batch-sequential		Separate programs executed sequentially with batched input.	... problem easily formulated as a set of sequential, severable steps.	... interactivity or concurrency between components necessary or desirable. ... random-access to data required.
Pipe-and-filter		Separate programs, aka filters, executed, potentially concurrently. Pipes route data streams between filters.	[as with batch-sequential] ... filters are useful in more than one application. ... data structures easily serializable.	... interaction between components required.

Table 4-1
(Continued)

Style	Category	Name	Summary	Use It When ...	Avoid It When ...
Shared Memory					
Blackboard			Independent programs, access and communicate exclusively through a global repository known as blackboard.	... all calculation centers on a common, changing data structure. ... order of processing dynamically determined and data-driven.	... programs deal with independent parts of the common data. ... interface to common data susceptible to change. ... interactions between the independent programs require complex regulation.
Rule-based			Use facts or rules entered into the knowledge base to resolve a query.	... problem data and queries expressible as simple rules over which inference may be performed.	... number of rules is large. ... interaction between rules present. ... high-performance required.
Interpreter					
Interpreter			Interpreter parses and executes the input stream, updating the state maintained by the interpreter.	... highly dynamic behavior required. High degree of end-user customizability.	... high performance required.
Mobile code			Code is mobile, that is, it is executed in a remote host.	... it is more efficient to move processing to a data set than the data set to processing. ... it is desirous to dynamically customize a local processing node through inclusion of external code.	... security of mobile code cannot be assured, or sandboxed. ... tight control of versions of deployed software is required.
Implicit Invocation					
Publish-subscribe			Publishers broadcast messages to subscribers.	... components are very loosely coupled. ... subscription data is small and efficiently transported.	... middleware to support high-volume data is unavailable.
Event-based			Independent components asynchronously emit and receive events communicated over event buses.	... components are concurrent and independent. ... components heterogeneous and network-distributed.	... guarantees on real-time processing of events is required.

(Continued)