

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.

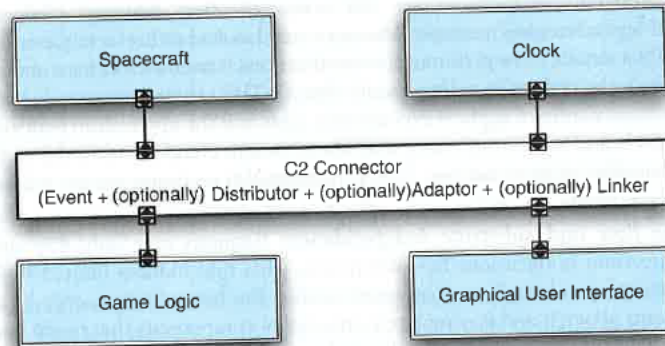
One important note concerns the apparent dependence of a given component on its superstrate, that is, the components above it, since a component can issue a specific request upward in an architecture. If each component is built so that its top domain closely corresponds to the bottom domains of those 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.



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.

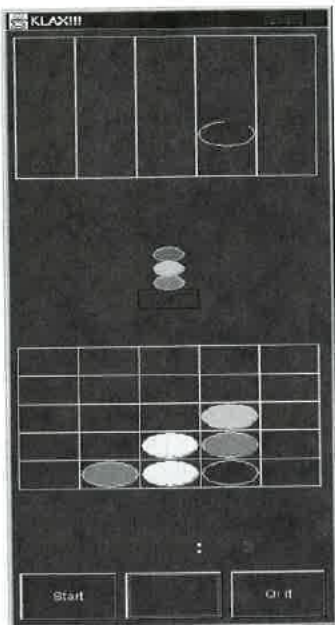
A more instructive application of C2 is presented in Figure 4-25, in which a KLAX-like game⁷ is supported. A screenshot from the game is shown in Figure 4-24. The essence of the game is that colored tiles fall from the chutes at the top of the user's screen. A palette is used to move horizontally across 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.

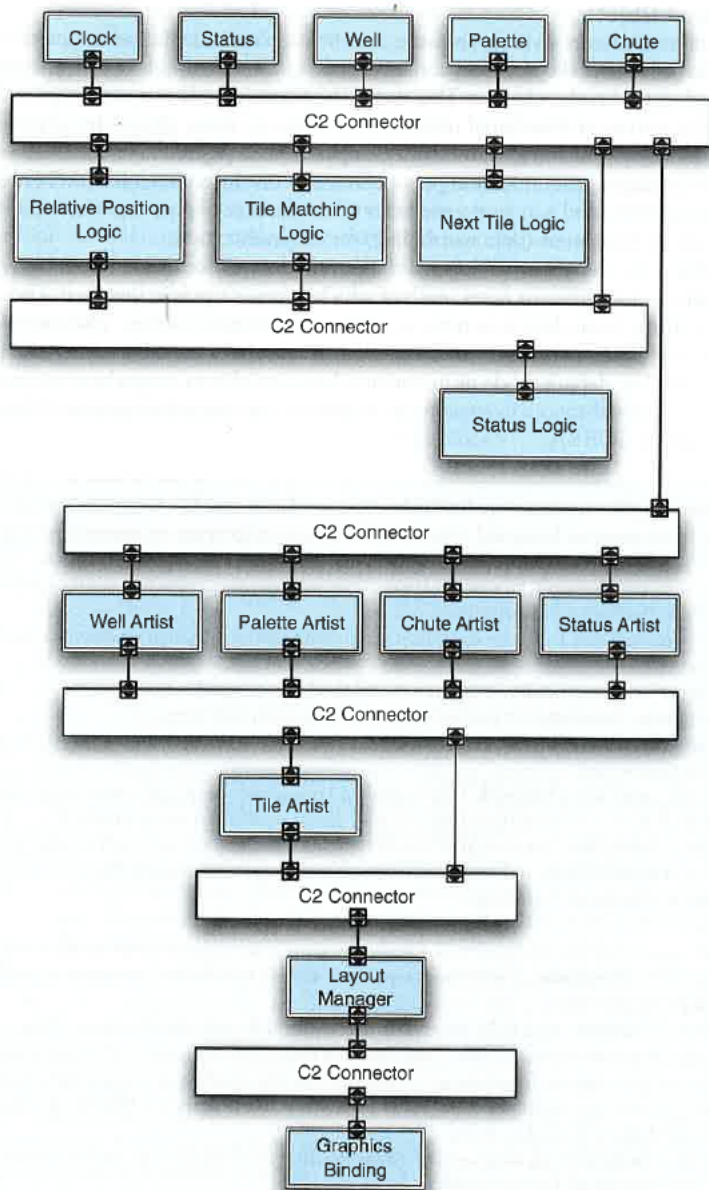
The artist components receive notifications of game-state changes, causing them to update their depictions. Each artist maintains the state of a set of abstract graphical

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



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.



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.

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.

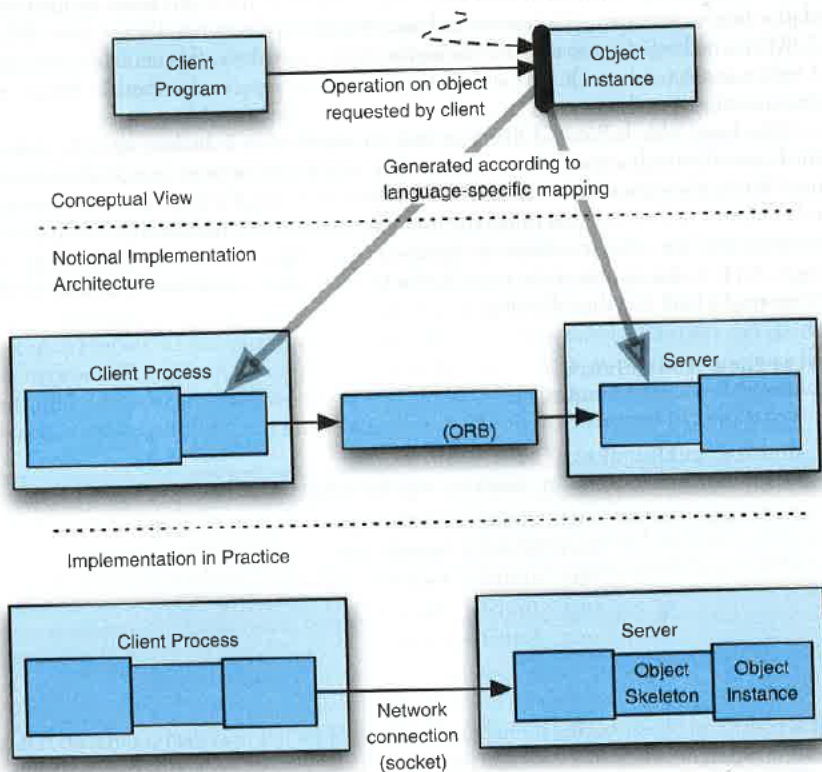
```
interface IDataStore{
    double getAltitude();
    void setAltitude(in double newAltitude);
    double getBurnRate();
    void setBurnRate(in double newBurnRate);
    void getStatus(out double altitude,
                  out double burnRate,
                  out double velocity,
              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



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.

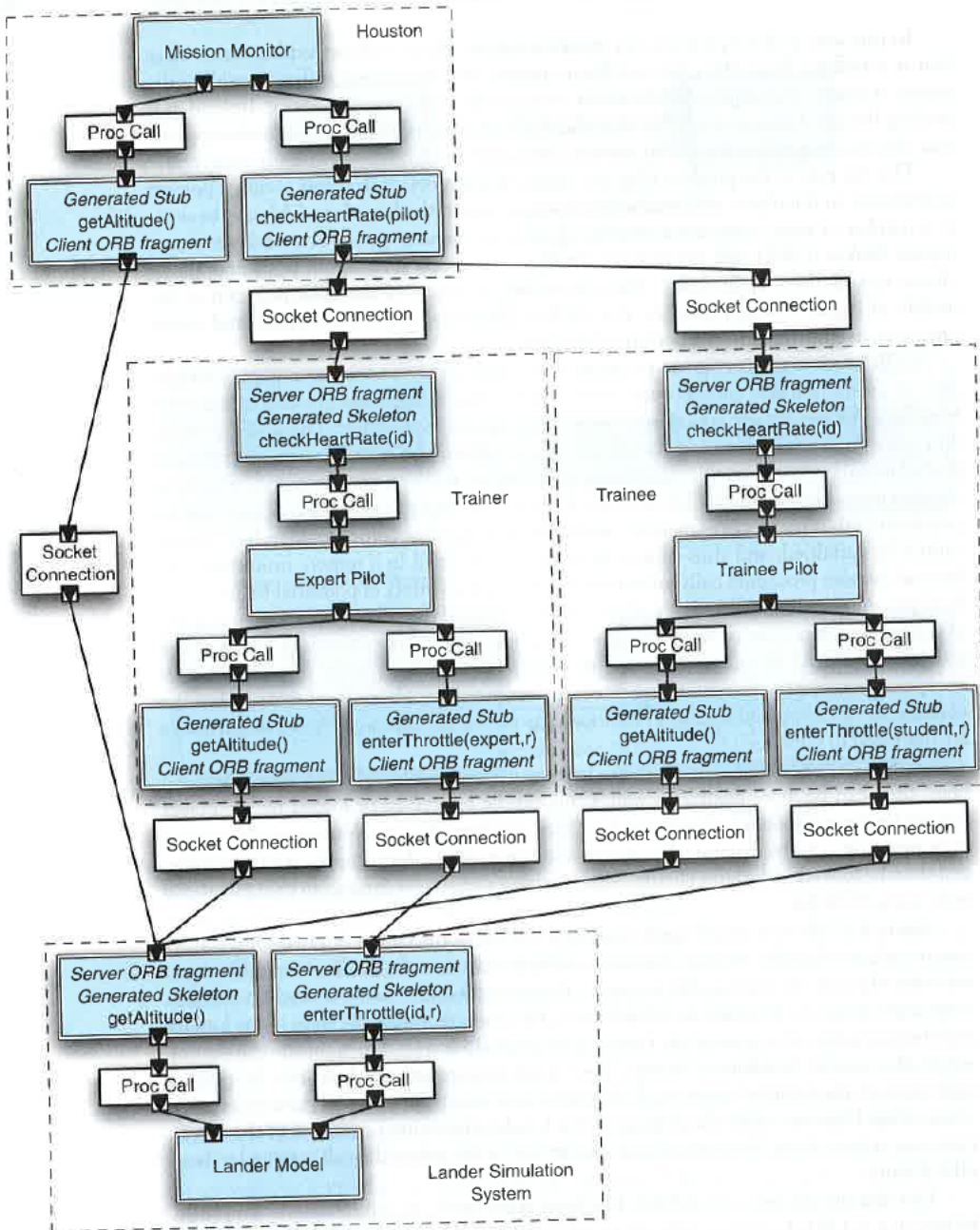
The last part of the puzzle is how the client program/object stub pair obtain a pointer or reference to the object skeleton/object instance pair in the first place. This can be done in a number of ways, most commonly by a lookup or naming service known as an object request broker (ORB) that maintains 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



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. 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."

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.

to do—and
effectively medi

constraints enable the use of code
techniques—since patterns of interaction
are efficiently supported standard
constraints—assert
then I know

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

15.2 DOMAIN-SPECIFIC SOFTWARE ARCHITECTURE

A domain-specific software architecture (Tracz 1995; Coglianesi, Smith, and Tracz 1992; Tracz 1994; Tracz, Coglianesi, and Young 1993) comprises a codified body of knowledge about software development in a specific application domain. Barbara Hayes-Roth (Hayes-Roth et al. 1995) provides a useful operational definition of a domain-specific software architecture:

Definition. A domain-specific software architecture (DSSA) comprises:

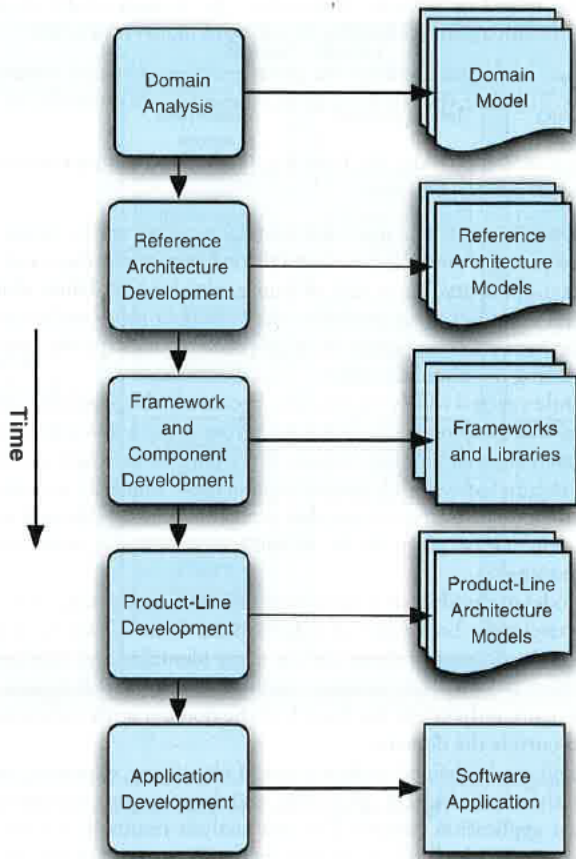
- A reference architecture, which describes a general computational framework for a significant domain of applications.
- A component library, which contains reusable chunks of domain expertise.
- An application configuration method for selecting and configuring components within the architecture to meet particular application requirements.

Figure 15-6 provides an generalized overview of a DSSA-centric software development process. In addition to the usual *application engineering* activities in which developers engage, DSSAs involve a number of *domain engineering* activities. These activities result in models of the application domain's relevant entities, their characteristics, and their relationships; definitions of the key terminology; canonical requirements; design- and implementation-level solutions that are reusable across systems in the domain; and tool support specialized to aid development within the domain.

This body of assets does not come for free. Instead, it is built over time, as engineers amass experience (both good and bad) of building individual systems within a domain and try to generalize that experience for use in future systems. In the remainder of this section we will discuss the characteristics of application domains and domain models, the requirements that remain stable and those that change within a domain, and the corresponding solution strategies. We will illustrate the discussion with an example DSSA derived from the different Lunar Lander architectures discussed in the preceding chapters.

15.2.1 Domain Knowledge

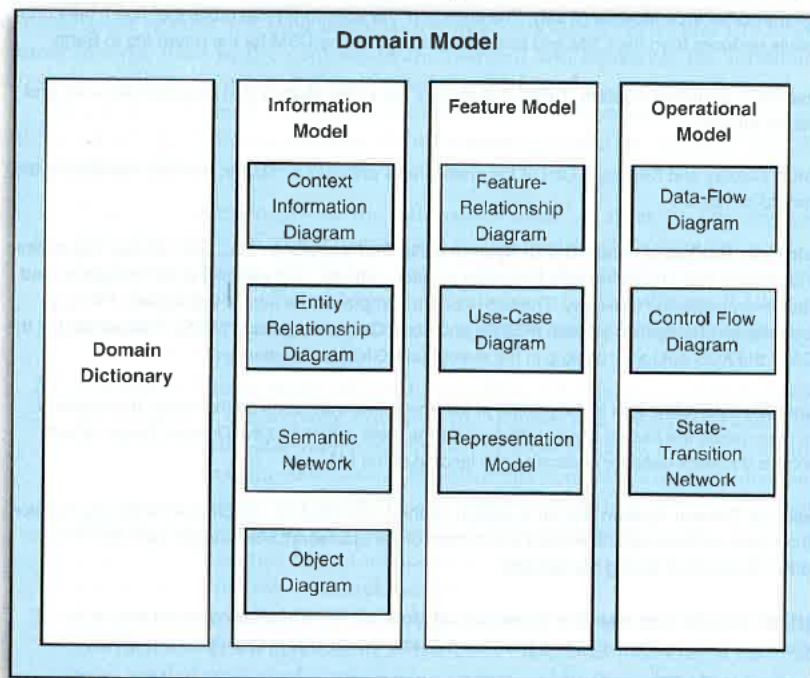
As we have discussed, one of the main concerns of domain-specific software engineering is the problem domain itself. In order to exploit the properties of the domain, its key characteristics must be captured in a domain model. Simply put, a *domain model* (Batory, McAllester et al. 1995) is a representation of what happens in an application domain. This includes the *functions* being performed, the *objects* (also referred to as *entities*) performing the functions and those on which the functions are performed, and the *data* and *information* flowing among the entities and/or functions. In the parlance of the preceding section, a domain model deals with the problem space. Therefore, the above terms are used in their ordinary sense (for example, function as an operation performed on or by an entity "living" in the domain), rather than in the sense that software engineers typically assign to them (for example, function as a method that returns a value in a C++ program).



If we take avionics as an example domain, the functions of interest would be aircraft takeoff, landing, taxiing, flying, pitch, roll, yaw, fueling, refueling (possibly in-flight), maintenance, and so on. Data and information flowing among these functions would be various pilot commands, instrument signals, warnings, status check messages, fuel consumption rates, data collected for the black boxes, and so on. The entities in the domain would include the flight instruments, the aircraft's rudder and wing flaps, the fuel tanks, the fuel transferred into the tanks during fueling and from the tanks to the engine during flight, the hydraulic fluids used to control the aircraft, and so on. It is this type of terminology, entities, their relationships, and operations on them that go into constructing a domain model.

Note that nothing in the above (very brief and informal) domain description belies the fact that a significant portion of a given avionics system's functions, data, and entities





The different models and a broad cross-section of their associated diagram types are shown in Figure 15-7. We now discuss each of them in more detail.

Domain Dictionary

A domain dictionary represents the identification and definitions of terms used in the domain model. These terms may be widely used and understood outside the domain, in which case the goal of including them in the domain dictionary may be to define them carefully and avoid any inconsistencies and ambiguities in their usage. On the other hand, the terms may be specialized, or even invented for the sole purpose of fostering human stakeholders' communication and understanding within the domain. The domain dictionary should be updated over time with new terminology and with new or clearer understandings of existing concepts. A domain dictionary also becomes an indispensable tool for training new engineers working within the application domain. Figure 15-8 shows a partial domain dictionary from a hypothetical Lunar Lander DSSA.

Information Model

An information model is actually a collection of multiple models that may be used in different organizations and different DSSAs. The information model is a result of context analysis and information analysis. Context analysis results in defining the boundaries of



When thrust is applied, the following equation is used:

$$y = \frac{1}{2} * (a_{burner} - a_{gravity}) * t^2 + v_i * t + y_i$$

where:

y = altitude

a_{burner} = constant acceleration upward due to thrust

$a_{gravity}$ = constant acceleration due to gravity on a lunar body

(see **Acceleration** for sample values)

t = time in seconds

v_i = initial velocity

y_i = initial altitude

the domain and preserves information that may be otherwise implicit and scattered across many different systems and their artifacts. Information analysis then takes the result of context analysis (that is, the contours of the domain) and represents the intradomain knowledge explicitly in terms of domain entities and their relationships.

The information model ensures that the DSSA employs appropriate data abstractions and decompositions. Different elements of the information model are used by at least three types of stakeholders.

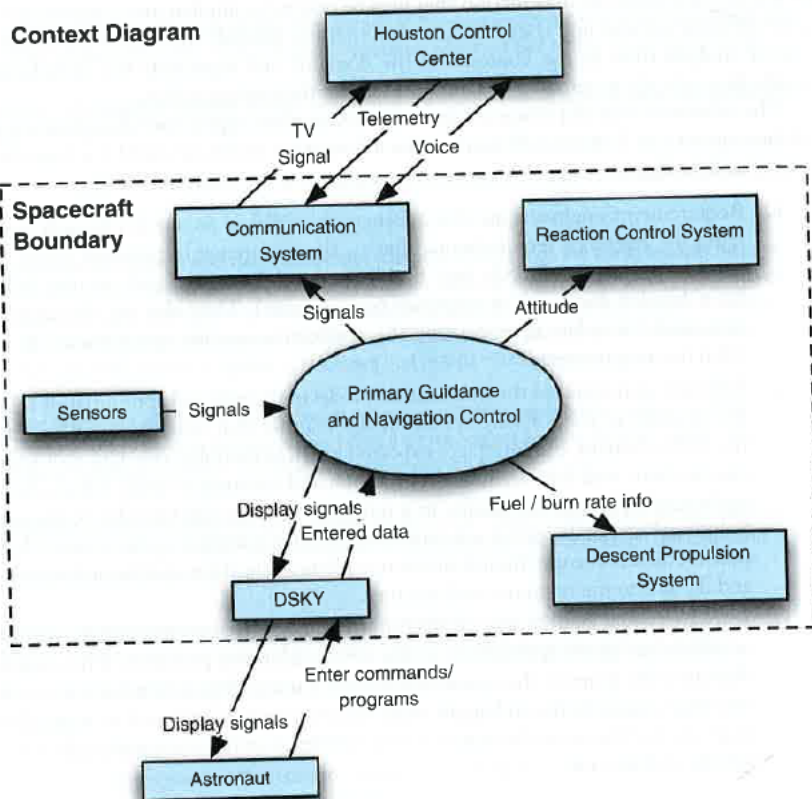
1. Requirements engineers use the information model as an aid in precisely specifying the reference requirements, that is, the requirements common across the applications in the DSSA (see the "Canonical Requirements" section below for a detailed discussion of reference requirements). They also use the information model to relate appropriately the application-specific requirements to the reference requirements.
2. Software architects use the information model to identify and appropriately relate the modules in the software system in the manner that reflects the characteristics of the domain, the interfaces exported by those modules, the data exchanged among them, and the mechanisms by which and constraints under which data is exchanged. This activity results in a reference architecture (see the "Canonical Solutions" section below for a detailed discussion of reference architectures). Software architects also use this information to relate application-specific architectures and designs to the reference architecture.
3. Software system maintainers use the information model to understand the manner in which any given application in the DSSA addresses problems. This enables them to relate properly the system maintenance and evolution requirements to the reference requirements and application-specific requirements, and to relate their proposed realization in the system to the reference architecture and application-specific architecture.

An information model usually consists of one or more of the below types of diagrams.

Context Information Diagram. This diagram captures the high-level data flow between the major entities in the system, their relationship to the entities outside the system (as well as outside the domain), as well as any data that is assumed to come from external sources. For example, in the Lunar Lander DSSA, the information exchanged among the spacecraft's sensors, actuators, and computers is part of the context information diagram. At the same time, the topology of the moon's (or planet's) surface on which the spacecraft is landing is considered to be outside the domain. An example context information diagram from the Lunar Lander DSSA is shown in Figure 15-9.

Entity/Relationship (ER) Diagram. This diagram captures aggregation ("a-part-of") and generalization ("is-a") relationships among the entities within the domain. For example, in the Lunar Lander DSSA, fuel level is "a-part-of" the spacecraft entity, while a given thruster "is-a" Lunar Lander actuator. An example ER diagram from the Lunar Lander DSSA is shown in Figure 15-10.

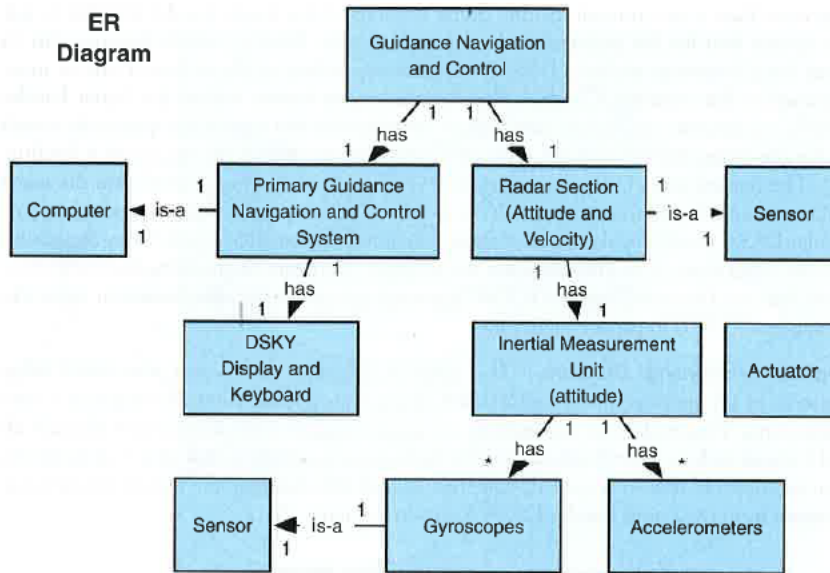
Object Diagram. This diagram identifies the objects in the application domain rather than in the software. In other words, the object diagram describes entities in the problem space;



the solution-space components used to realize the problem-space entities will be captured in the reference architecture, discussed below. The object diagram captures the attributes and properties of each object, as well as their interdependencies and interactions with other objects in the domain. An example object diagram from the Lunar Lander DSSA is shown in Figure 15-11.

Feature Model

The feature model is, in fact, a collection of several models. A feature model results from feature analysis. The goal of feature analysis is to capture and organize a customer's or end-user's understandings and expectations of the overall (shared) capabilities of applications in a given domain. The feature model is considered to be the chief means of communication between the customers and the developers of new applications. The feature model explicitly delineates the commonalities and differences among systems in the domain. Examples of features include function descriptions, descriptions of the mission and usage patterns, performance requirements, accuracy, time synchronization, and so on (Kang et al. 1990).



Such features are meaningful to the users and can assist the engineers in the derivation of DSSA that will provide the desired capabilities.

Features may be defined as mandatory, optional, or variant. Mandatory features are expected to recur across all systems in the domain. In the case of the Lunar Lander system, *Compute Altitude* is a mandatory feature: Any implementation of the Lunar Lander must be able to continuously compute the spacecraft's altitude. Optional features are those that are identified as useful for a subset of the systems within the given DSSA. For example,

Object Diagram

Object	Attributes	Operations
Landing Radar	Attitude Velocity	Sense Altitude Sense Velocity
Descent Engine	Throttle Level Nozzle Direction	
Thrust Engines	Firing Mode Attitude Translation	Set Pulse or Continuous Change Altitude Change Translation

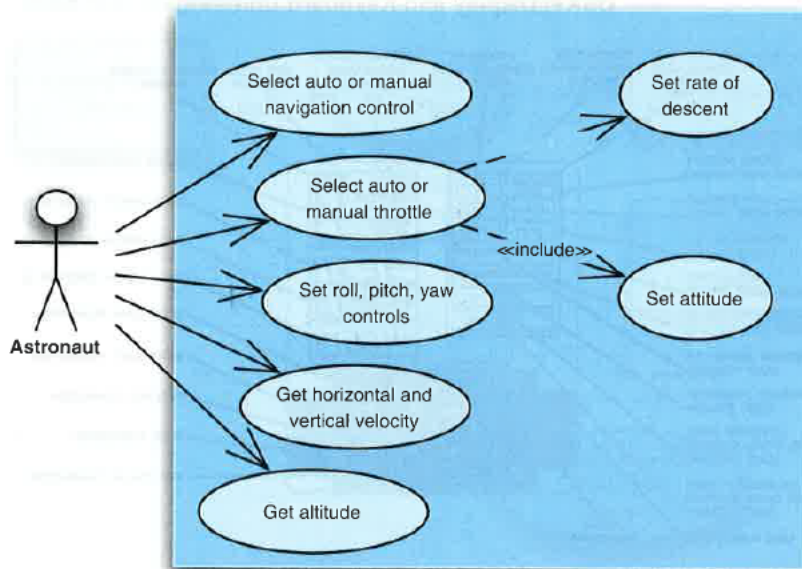
res exist in
ne or more
nar Lander

nchronizati
agram from

Real-time requirements: The thrusters and the descent engine must be able to respond to commands from the computer system in real-time.

Lunar Lander's computer system must function properly after a restart (restart protection).

the context diagram in the information model.

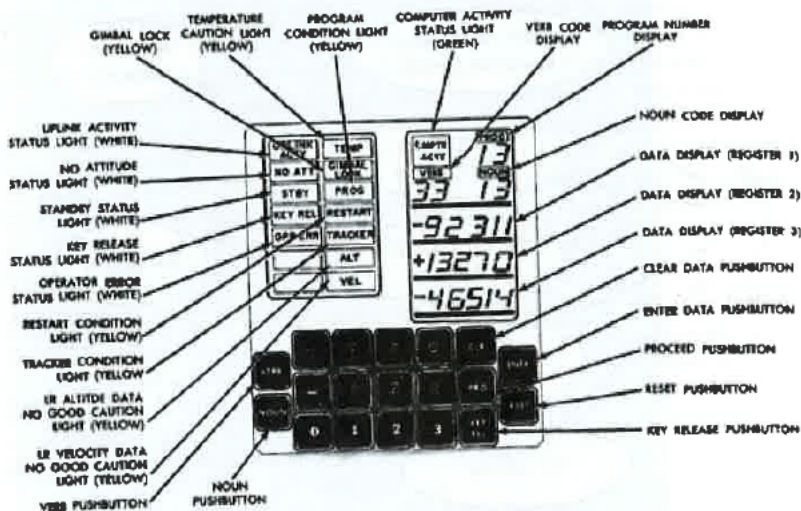


Use-Case Diagram. This diagram captures the usage scenarios in the system. Use cases have become prevalent with the introduction of the Unified Modeling Language (see Chapters 6 and 16). Use-case diagrams are elicited from domain experts, system customers, and system users, and encompass the manner in which a feature is expected to be used, as well as control- and data-flow among multiple features. An example use-case diagram from the Lunar Lander DSSA is shown in Figure 15-13.

Representation Model. This diagram describes how information is made available to a human user. In other words, the representation model details what the appropriate user interfaces are for the systems within the DSSA. This model also captures the information produced for another application: What kinds of input and output capabilities are available, and what is the expected, common format for the exchanged information? An example partial representation model from the Lunar Lander DSSA is shown in Figure 15-14.

Operational Model

The operational model is the foundation upon which the software designer begins the process of understanding (1) how to provide the features for a particular system in a domain, and (2) how to make use of the entities identified in the resulting domain model. The operational model identifies the key operations (that is, functions) that occur within a domain, the data exchanged among those operations, as well as the commonly occurring sequences of those operations. In other words, the operational model represents *how* applications within a domain work.



Three 5-digit registers
(General Purpose)
Three 2-digit registers
(Indicate landing phase)
19 Keys
Warning Lights

Issue commands via VERB (action) and NOUN (object)

- E.g.: VERB 6 NOUN 20
- VERB 6: Display in Decimal
- NOUN 20: Angles

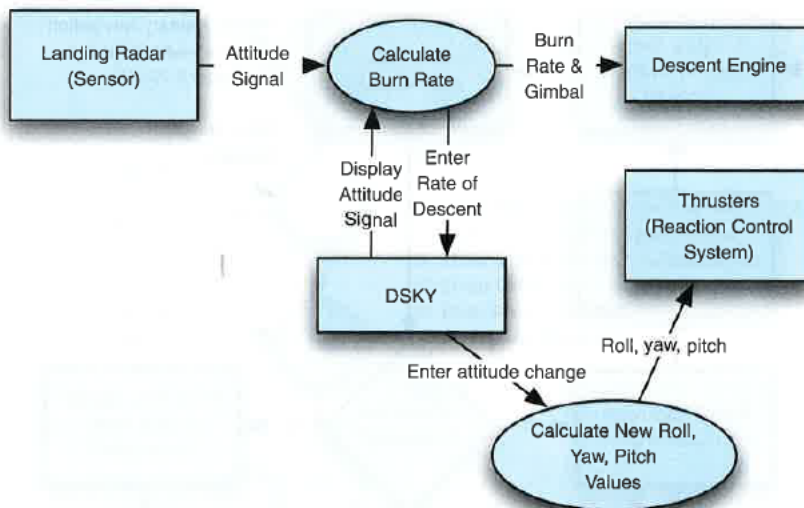
70 pre-defined PROGRAMS

- E.g., PROGRAM for each descent phase:
- P63 (Braking)
- P64 (Approach)
- P65 (Landing)

The operational model is a result of operational analysis, which identifies the commonalities as well as differences in control-flow and data-flow among the entities in a domain. The domain-wide entities identified in the information and feature models, discussed above, form the basis for the operational model: The information model captures the data exchanged by the operations, while the feature model captures the operations themselves. The control- and data-flow of each individual application within the DSSA can be derived from the operational model.

Three representative types of diagrams used within the operational model are discussed below.

Data-Flow Diagram. This diagram focuses explicitly on the data exchanged within the system, with no notion of control. An example data-flow diagram from the Lunar Lander DSSA is shown in Figure 15-15.

Data-flow Diagram

Control-Flow Diagrams. This diagram focuses on the exchange of control within the system, without regard for data. An example control-flow diagram from the Lunar Lander DSSA is shown in Figure 15-16.

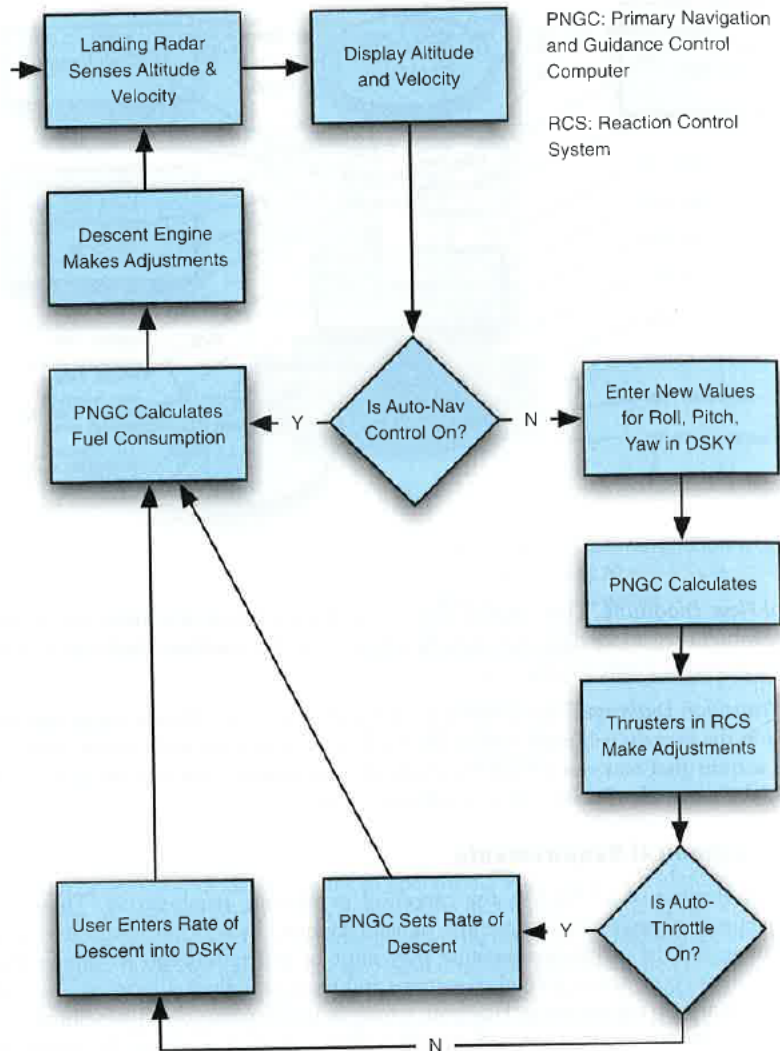
State-Transition Diagram. This diagram models and relates the different states that the entities in the domain will enter, events that will result in transitions between states, as well as actions that may result from those events. An example state-transition diagram from the Lunar Lander DSSA is shown in Figure 15-17.

15.2.2 Canonical Requirements

A critical element of a DSSA is the canonical or *reference* requirements. These are requirements that apply across the entire domain captured by a DSSA. Such reference requirements will be incomplete because they must be general enough to capture the variations that occur across individual systems and because individual systems also will introduce their own requirements. However, reference requirements directly facilitate the mapping of the requirements for each system within a given domain to the canonical domain-specific solution (discussed in the next section).

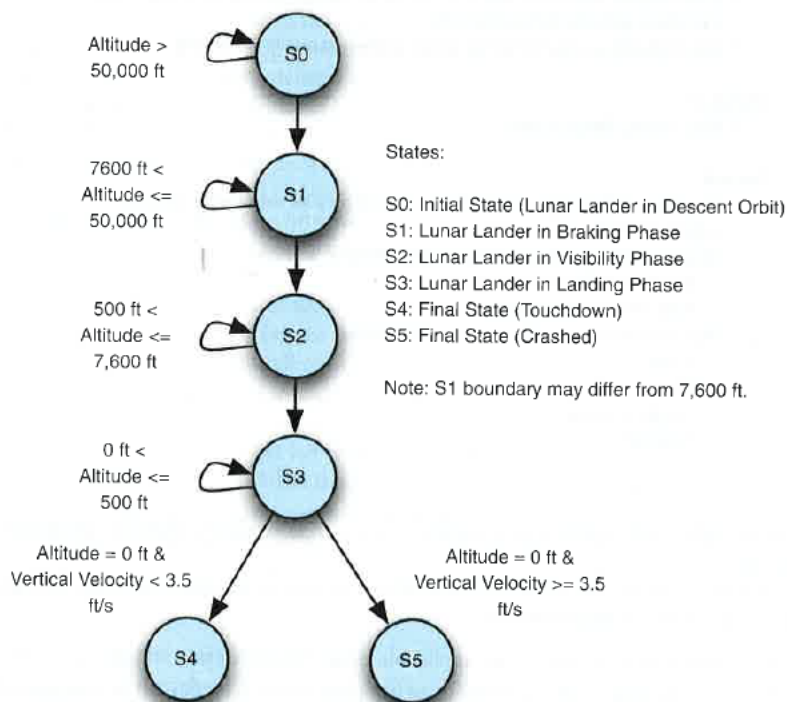
Similar to the requirements for any software system, the starting point for reference requirements is the customer's statement of needs. In a DSSA, this statement is general enough to apply to any system in the domain. A DSSA is an exercise in generalization from specific experience, so the statement may emerge over time from similar statements that were specific to individual systems constructed previously within the domain. The

Control Flow Diagram



customer needs statement identifies the functional requirements for the DSSA at a high level of abstraction. It is usually informal, ambiguous, and incomplete. It is a starting point for deriving (reference) requirements, but it is not the same as reference requirements. As an example, Figure 15-18 shows a partial customer needs statement for the Lunar Lander DSSA.

State Transition Diagram



Reference requirements are then derived from the customer needs statement as well as the requirements of the individual legacy systems within the domain. The reference requirements contain the defining characteristics of the problem space. These are the *functional requirements*. Reference requirements also contain limiting characteristics (that is, constraints) in the solution space. These are the *non-functional requirements* (such as security, performance, reliability), *design requirements* (such as architectural style, user

A family of related systems is needed that

1. Supports the Lunar Lander video game for different platforms (e.g., PC, Mac, Pocket PC, cell phone, etc.).
2. Supports different user interfaces, from simple textual to a sophisticated 3-D GUI.
3. Runs on a single machine or is decentralized across a network.
4. Works in a single-user or multi-user competition mode, and supports different variations (e.g., only landing from a given height, first entering orbit and then landing, landing on different types of terrain, landing on different types of celestial bodies, etc.).

Mandatory

- Must display the current status of the Lunar Lander (horizontal and vertical velocities, altitude, remaining fuel)
- Must indicate points earned by player based on quality of landing

Optional

- May display time elapsed

Variant

- May have different levels of difficulty based on pilot experience (novice, expert, etc.)
- May have different types of input depending on whether:
 - Auto Navigation is enabled
 - Auto Throttle is enabled
- May have to land on different celestial bodies:
 - Moon
 - Mars
 - Jupiter's moons
 - Asteroid

interface style), and *implementation requirements* (such as hardware platform, programming language).

Similar to the feature model discussed above, reference requirements must distinguish among three types of requirements:

1. *Mandatory requirements* are applicable to all systems in the domain.
2. *Optional requirements* are known to be applicable to a certain set of systems within the domain.
3. *Variable requirements* are application-specific, and may be unknown at the time the DSSA, and more specifically reference requirements, are formulated.

In general, each requirement—whether functional, non-functional, design, or implementation—can be of any one of the three types (mandatory, optional, or variable). We provide examples of mandatory, optional, and variable requirements for the Lunar Lander in Figure 15-19.

15.2.3 Canonical Solution Strategies—Reference Architectures

When business goals and technology are applied to a problem within a domain, solutions emerge. A single product architecture represents a single solution. However, as we have covered extensively, when attacking problems in the same domain with similar business goals in mind and leveraging similar technology, similarities will emerge in solution architectures. These similarities can be captured in a *reference architecture*. Here, we repeat the definition found in Chapter 3.

Definition. *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.

As sets of principal design decisions, reference architectures are themselves architectures. However, they are generic or variegated such that they are applicable to multiple systems simultaneously—in the context of the terms we have used in this chapter, to a population or a product line of systems. Points where the architecture can vary from family member to family member are explicitly defined as part of the reference architecture.

There are many different kinds of reference architectures; this definition is intentionally broad. They differ in how they express the commonalities and the points of variation in a population or product line. Three classes of reference architectures are:

Complete single-product architecture: An ordinary product architecture of a simple toy example or a complete system for a particular domain can be considered a reference architecture, since it might serve as an exemplar for future products in the same domain. Such reference architectures are relatively weak because they do not provide much engineering guidance as to how to use the reference architecture to create new systems, and they do not explicitly call out variation points.

Incomplete invariant architecture: A partial architecture can be specified that is constant and unchanging among all products. Parts of the architecture that vary from product to product are left unspecified, although some design guidance might be provided as to how to “fill in the blanks” in the architecture.

Invariant architecture with explicit variation: This kind of reference architecture can be used to simultaneously capture both the invariants and the variants in products in a domain. For each variation point, permitted alternatives are specified in detail.

Developing a Reference Architecture

Developing a reference architecture is a serious decision because it will be the guiding foundation for a number of future products. Determining when, how, and why to develop a reference architecture is critical to success.

When to Develop a Reference Architecture. An obvious question that arises in DSSE is, “When is the right time to develop a reference architecture?” Choosing an appropriate time is critical because a reference architecture will constrain and bind all solutions in the domain. Premature development of a reference architecture can be disastrous if the architecture is not validated or the stakeholders’ understanding of the domain is too incomplete: Instead of affecting just one product, a mistake made in a too-early reference architecture will propagate to many products and vastly increase the costs of that mistake. On the other hand, developing a reference architecture too late can be equally costly: If many diverse solution architectures have already been developed in a domain, then there will be little opportunity to adapt these existing architectures to fit the reference architecture (they will become “legacy systems”), and therefore substantially limit reuse benefits.

A good tactic that mitigates some risk is to adopt an incremental approach with respect to reference architecture. The reference architecture should still be developed with the entire family of products in mind—if this is not done, it may take on the characteristics of a single product and later have to be significantly adapted to incorporate the full family. If, as is often the case, a number of products have already been developed and they are being adapted to use a reference architecture, it is usually easier to adapt one or two products

architecture is c
lows the referenc
cts simultaneo

Choosing
architectu

Architecture. Deciding how to captu reference
he stakeholders, and the underlying bu ss needs of
leling techniques (such as the of explicit

offers the guidanc architects
ts (unless roducts hat family
ence archi e). If th rm is chose

f complete product architectures in a family to illustrate this.
plete invariant architectures have the advantage that they tell architects directly
be present in a family member's architecture. However, they may lack details
on process. Again, important documentation
d proceed is important. Another (somewhat

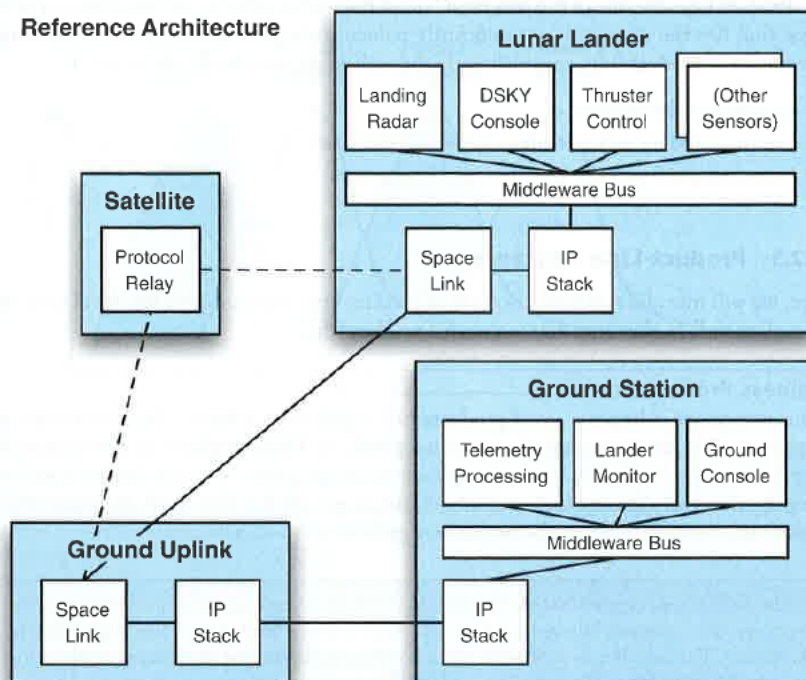
1

possible alternatives will far exceed the busin
example, imagine such a reference architectur
may be included or not included. This referenc
possible architectures. Not all of these archite
conflicts) or desirable to produce (due to mar

multiple co nated views, exten document tion cap
points, guidance on how efine the ference
cation. ionale for the vari decision

one is not enough to interpret its intent; sup-
example, it may be that the components shown
the system, or implementers may be advised to
ertain points of variation are obvious: whether
middleware, the implementation of the space
on. The intended interpretation of all these

Reference Architecture



elements should be contained in ancillary documentation; this is typical of a reference architecture.

15.2.4 Product Lines and Architecture

There are many significant differences between designing a single-product architecture and a reference architecture; chief among them is that a reference architecture must serve as the basis for many different products simultaneously. This introduces a whole host of new issues: Not only must the architecture suffice for a single solution, but must be developed, visualized, and evaluated in terms of a large number of potential solutions (many of which may never be realized).

Here is where the technologies and techniques developed in the *product-line architecture* community are applicable. Product-line architectures give stakeholders tools with which to diversify an ordinary product architecture into an artifact suitable for describing many similar/related products: a *product line*. The techniques presented here allow stakeholders to develop multi-product architectures without losing the benefits that we have identified in the single-product case: explicit models, visualization, analyzability, and so on.

significant
s directly

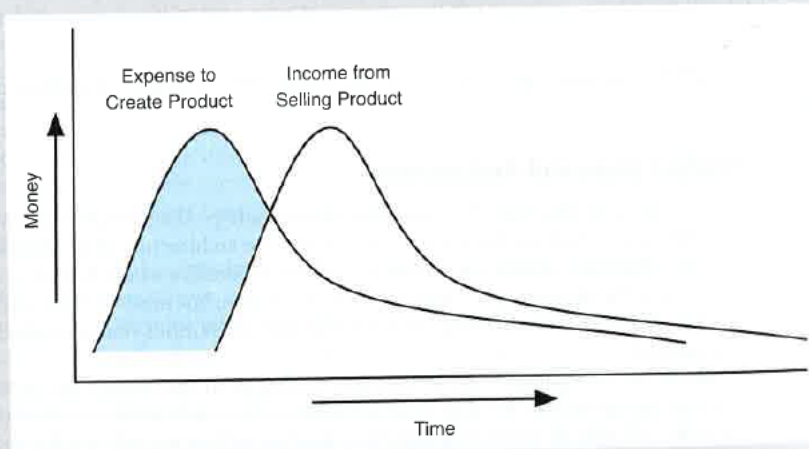
15.2.5 Product-Line Concepts

Here, we will introduce different notions of product lines and how they are developed and defined, as well as elements that make up a product line.

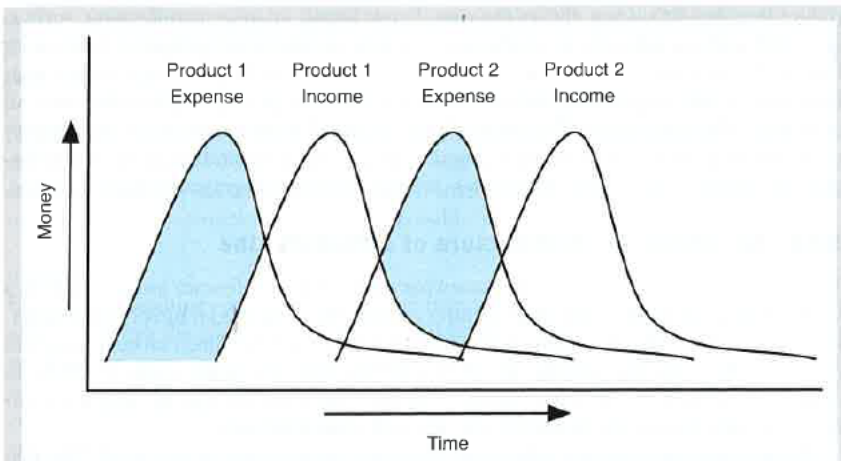
Business Product Lines

A business product line is a set of products tied together to achieve a business/monetary purpose, such as increasing sales by bundling products. These products do not necessarily have to have any similarity from an engineering perspective—it is not uncommon for a company that has just acquired new subsidiaries to put all the new products into a single product line to better market the products together.

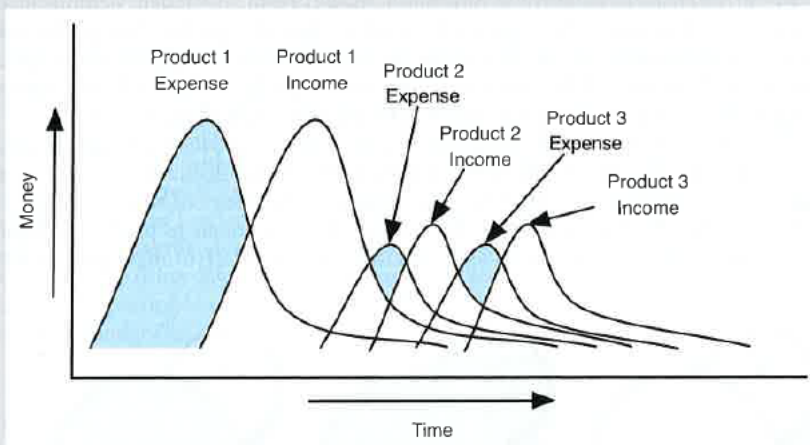
In a business based on product sales, there is a gap between expense and income. Money is spent during product creation, but income does not begin to arrive until the product is available in the market. This situation is depicted in the following graph, with the shaded area representing the initial outlay of capital required for product development.



If traditional development practices are used, additional products will have similar expense/income curves, like so:



The use of product lines, however, lowers the development cost and time for making a new product in the product line. In doing so, additional revenue can come in earlier, reducing the effect of expenses that are incurred, like so:



Note that the shaded areas (times when expenses exceed revenues) are much more limited when product lines are used.

Engineering Product Lines

Products in an engineering product line are tied together by similarities in how they are designed or constructed. Often, a product line will be both a business and engineering

product line, but this is not always the case. For example, an auto manufacturer, such as Ford, will build a number of cars on the same engineering platform but market them under different brand names, such as Lincoln, Mercury, and Mazda. Engineering product lines often share significant portions of their architecture, and the products within them may all conform to a broader DSSA. The product lines discussed in the remainder of this chapter are engineering product lines; however, Section 15.2.10 discusses strategies for unifying disparate products (perhaps those in a business product line) into an engineering product line.

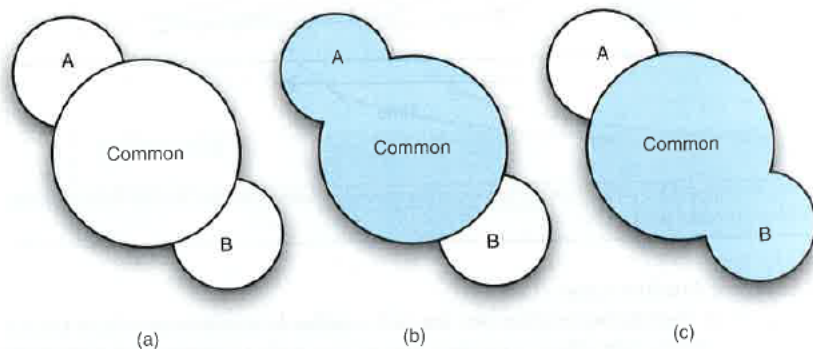
15.2.6 Specifying the Architecture of a Product Line

The architectures of product lines are somewhat different from reference architectures in a DSSA. In general, product-line architectures capture the complete architectures of multiple related products simultaneously. Conversely, reference architectures can be incomplete or partial. They may, for example, specify the architecture of a single product with loose guidance about how to adapt it to other contexts. They may also specify only invariant parts of an architecture and leave the rest up to solution developers.

Product-line architectures differ from reference architectures in two ways. The first difference is one of scope: Rather than attempting to describe the architectures of many (potentially diverse) solutions within a domain, product-line architectures focus on a specific, explicit set of related products, often developed by a single organization. The second difference is one of completeness: Product-line architectures generally capture multiple complete product architectures rather than leaving parts undefined.

Recall our characterization of architecture as the set of principal design decisions about a system. A product-line architecture can be similarly defined: A product line architecture captures simultaneously the principal design decisions of many related products. Some of these design decisions will be common among all the products, some will be common among a subset of the products, and some will be unique to individual products.

Figure 15-21 shows a product-line architecture with two products conceptualized as a Venn diagram. The large center circle represents the design decisions common to all products in the product line. The “A” shape represents the decisions that are unique to Product A. The “B” shape represents the decisions that are unique to product B. Figure 15-21(a) shows the three groups of design decisions. Figure 15-21(b) highlights Product



A: the union of the common design decisions and the “A” design decisions. Figure 15-21(c) shows Product B: the union of the common design decisions and the “B” design decisions.

To further discuss product-line architectures, we turn again to the Lunar Lander example. We have discussed different variants of the Lunar Lander game in earlier chapters, but each has been discussed in isolation. Product-line architectures give us the power to specify and discuss a family of Lunar Lander games in terms of explicit differences among the family members. The constructs that make this possible are *variants* and *versions*.

Variants

Extending an ordinary software architecture into a product line architecture can be accomplished through the addition of *variation points* to create *variant* architectures. Each variant architecture represents a different product—a member of the product line. A *variation point* is a set of design decisions that may or may not be included in the architecture of a given product. Variation points identify where the design decisions for a specific product architecture diverge from the design decisions for other products. Each variation point is accompanied by a condition that determines when the design decisions for that variation point are included in a product’s architecture. In the above example, the inclusion of the “A” design decisions can be seen as a variation point. The condition for this variation point is simple: The decisions are only included if the product being built is Product A.

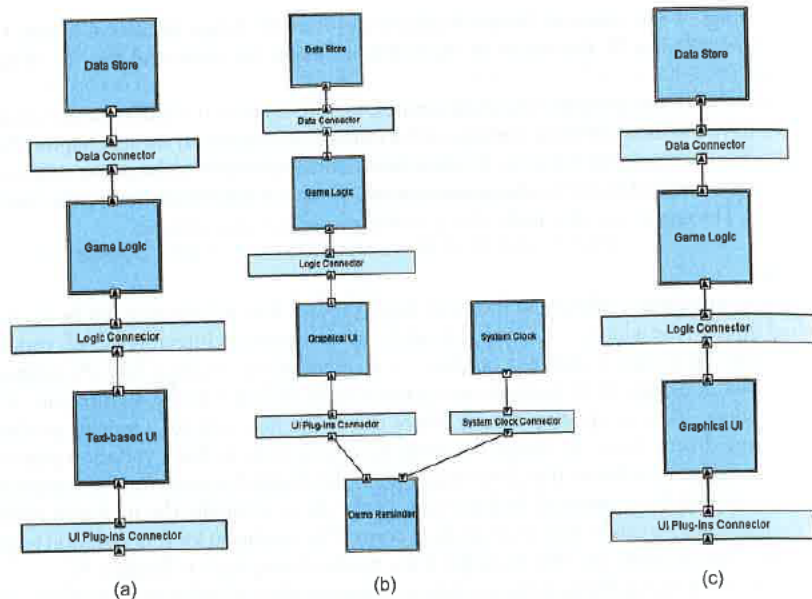
By isolating and codifying variation points, a large number of product architectures can be described compactly by exploiting the different combinations of variations. Consider a hypothetical product line of Lunar Lander games that consists of three slightly different Lunar Lander products: Lunar Lander Lite, Lunar Lander Demo, and Lunar Lander Pro.

Lunar Lander Lite is intended as a freely distributed Lunar Lander game with a text-based user interface. It resembles the Lunar Lander games discussed in earlier chapters, such as those implemented in Chapter 9. Lunar Lander Pro is intended as a commercially sold Lunar Lander game with a fancier graphical user interface. Internally, the data store and game logic are identical to Lunar Lander Lite’s; the only difference is in the user interface component. Lunar Lander Demo is a freely distributed but time-limited version of Lunar Lander Pro. It has the same graphical user interface as Lunar Lander Pro, but it expires and is not playable thirty days after installation. Periodically, it reminds users that if they want to continue playing beyond the thirty-day limit, they are required to purchase or upgrade to Lunar Lander Pro.

The architectural structures of all three Lunar Lander products are shown in Figure 15-22.² Figure 15-22(a) shows Lunar Lander Lite, consisting of a data store and game logic component, along with a text-based UI. Figure 15-22(b) shows Lunar Lander Demo, with a graphical UI component in place of the text-based UI, and an additional demo reminder and system clock that plug into the user interface to remind the user to register. Figure 15-22(c) shows Lunar Lander Pro, identical to Lunar Lander Demo but missing the Demo Reminder and System Clock components.

Table 15-1 shows the elements—in this case components and connectors—included in each of the three product-line members. This kind of table is good for deciding whether

²The Lunar Lander example in this chapter will focus on structural design decisions; however, the techniques described apply, in general, to broader kinds of design decisions that go beyond just components and connectors.



and how to create a product line. Here, it is obvious that there is significant commonality and overlap among the three products—good motivation for creating a product line.

Once the creation of a product line begins, it is a good idea to group low-level elements into variation points. Recall that each variation point represents a set of design decisions that may or may not be included in the architecture. This allows product-line architects to think about products in terms of features rather than low-level elements. For example, while it is conceivable to think of a product that contains a System Clock Connector but no System Clock, in practice this does not make much sense.

Components and Connectors	Product		
	Lite	Demo	Pro
Data Store	X	X	X
Data Store Connector	X	X	X
Game Logic	X	X	X
Game Logic Connector	X	X	X
Text-Based UI	X	X	
UI Plug-Ins Connector	X	X	X
Graphical UI		X	X
System Clock		X	
System Clock Connector		X	
Demo Reminder		X	

<i>Components and Connectors</i>	<i>Core Elements</i>	<i>Text UI</i>	<i>Graphical UI</i>	<i>Time Limited</i>
Data Store	X			
Data Store Connector	X			
Game Logic	X			
Game Logic Connector	X			
Text-Based UI		X		
UI Plug-Ins Connector	X			
Graphical UI			X	
System Clock				X
System Clock Connector				X
Demo Reminder				X

A mapping of elements to variation points is shown in Table 15-2. Here, elements common to all products are grouped into Core Elements. Three other variation points are defined as well: the inclusion of a textual UI, a graphical UI, and whether or not the product is time-limited. The grouping of architectural elements or design decisions into features is often motivated by domain constraints, business goals, and the individual dependencies and exclusion relationships among the elements. Ultimately, these groupings must be chosen and defined by system architects—their creation is not an algorithmic process. Variation points will have dependencies and exclusion relationships as well: For example, all variation points depend on the inclusion of the Core Elements, and at least one of the two user interface variation points must be included in a product. These relationships and constraints must be carefully documented.

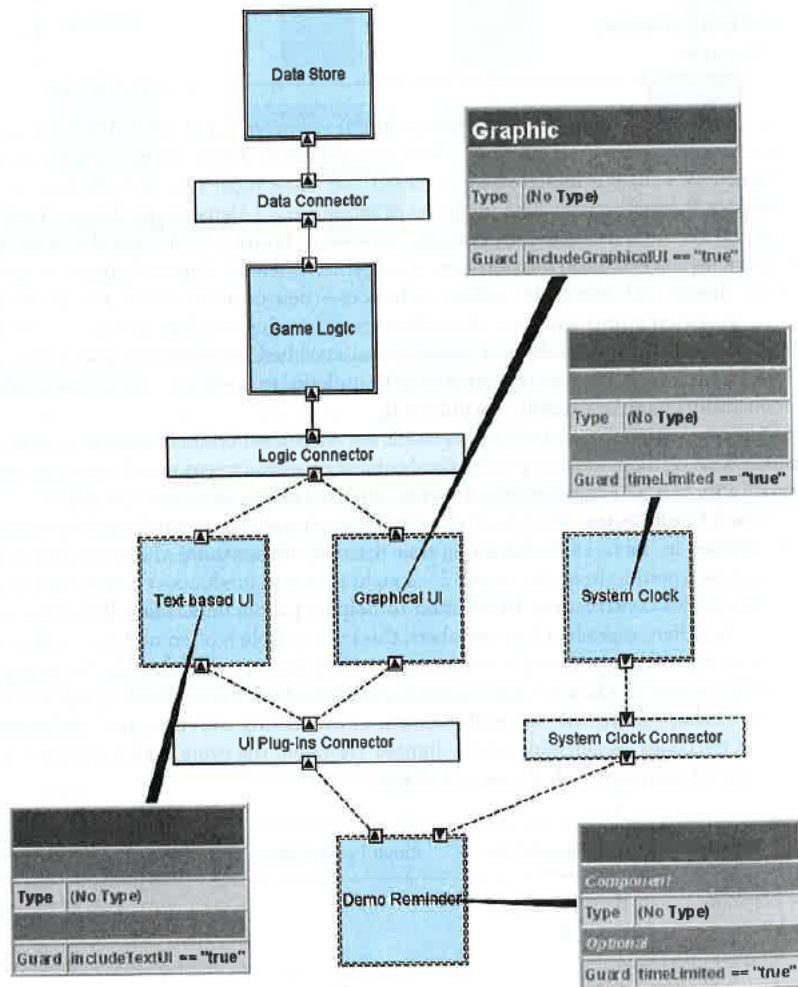
Once the variation points have been defined, individual products can be specified as combinations of the variation points. For business reasons, or reasons of variation point compatibility, not all combinations of variation points will be made into products.

Table 15-3 shows the creation of the three Lunar Lander products from the variation points defined in Table 15-2. Assuming that the core elements are always included, the three variation points altogether create 2^3 or eight potential products, of which three have been selected for construction. In addition to helping people understand the differences between the different product-line members, this kind of table is often useful in generating ideas for new products using the variation-point combinations *not* selected. For example, one could conceive of a Lunar Lander game that includes both a textual *and* a graphical user interface, perhaps giving the user multiple ways of interacting with the game and viewing its state. One could also imagine a time-limited version of the game with a textual UI, or a time-limited version with both user interfaces.

	<i>Lunar Lander Lite</i>	<i>Lunar Lander Demo</i>	<i>Lunar Lander Pro</i>
Core Elements	X	X	X
Text UI	X		
Graphical UI		X	X
Time Limited		X	

By unifying common features and documenting these explicit variation points, we can combine these three individual architecture descriptions into a single product-line architecture. A few architecture description languages such as Koala (van Ommering et al. 2000) and xADL (Dashofy, van der Hoek, and Taylor 2005) allow you to do this directly.

The combined product line is shown in Figure 15-23, here in xADL's graphical visualization using xADL's product-line features (Dashofy and van der Hoek 2001) to express the points of variation. All elements from all products are shown together. Elements such as the *Data Store* and *Game Logic* components that are part of the core are represented normally. Elements that are included in one or more variation points are represented as



optional. Each optional element is accompanied by a guard condition that indicates when it should be included in a product. In this product line, three variables—`includeTextUI`, `includeGraphicalUI`, and `timeLimited`—correspond to the variation points defined in Table 15-2. By setting the value of any of these variables to “true,” the elements corresponding to that variation point will be included in the architecture.

xADL’s use of a Boolean expression language to document these guard conditions gives it additional expressive power, including some ability to constrain relationships between features. For example, the *Text-based UI* and *Graphical UI* features could be made mutually exclusive by unifying the `includeTextUI` and `includeGraphicalUI` variables into a single variable, say ‘`uiType`,’ whose value would be either “graphical” or “textual,” but not both.

Selection

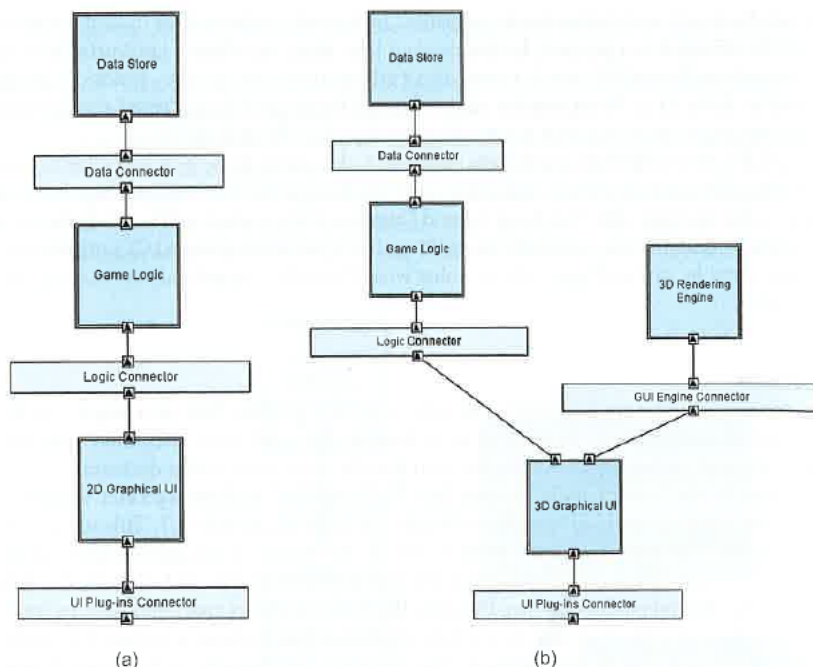
The process of reducing a product line into a smaller product line or a single product is known as *selection*. Selection involves evaluating the conditions associated with each variation point and either including or excluding the associated design decisions.

Consider the Lunar Lander product line. By fixing the `includeTextUI` variable to “false,” we have selected only products that do not include a textual UI. This reduces the total number of potential products from 8 to 4. By also fixing “`includeGraphicalUI`” to “true,” we can select only those products that include a graphical UI. Now, the total number of potential products is two. By fixing the final variable, `timeLimited`, to “true,” we can reduce the product line to a single product—one without a textual UI, with a graphical UI, and that is time-limited. This product, incidentally, is the Lunar Lander Demo product—something easily seen by referring to Table 15-3.

Without tool support, product-line selection must be performed manually—either on the fly in the minds of the stakeholders interpreting architectural models or by creating and maintaining explicit models of each product in the product line. Using an approach that supports the explicit specification of variation points and product lines such as Koala or xADL can make the selection process substantially easier. Specifically, in these approaches product-line selection can be performed automatically through the use of tools. xADL’s product-line selector tool allows users to bind values to guard variables, and then reduce a product-line architecture such as the one shown in Figure 15-23 into a single-product architecture such as the one shown in Figure 15-22 with a few clicks of the mouse.

15.2.7 Capturing Variations Over Time

The products in a product line are often alternatives to one another, intended for simultaneous development and release, as is the case with the Lunar Lander product line described above. However, product-line architectures also can be used to capture different versions of the same product over time. In general, different versions of the same product will have relatively similar architectures (unless something drastic such as a complete redesign or rewrite occurs). When this is the case, it is possible to enumerate the design decisions in both architectures and determine what changed from one version of a product to the next. These can be turned into variation points and encoded into a product line, where the different products are not alternatives, but rather the same product at different times.



Consider an evolving Lunar Lander game. The first version of the game might include a graphical user interface with two-dimensional graphics. A second version of the same game might differ by including a graphical user interface with three-dimensional graphics, supported by an off-the-shelf three-dimensional rendering engine. These two product architectures are shown in Figure 15-24(a) and (b), respectively.

Clearly, this situation mirrors the situation seen earlier when the products were different games to be marketed, rather than versions of the same game. Using the same mechanisms described above, the two versions of the game might be encoded into a single product line, with each product representing one version of the game. This product line is shown in Figure 15-25. Here, the version is selected using a single variable, *version*, rather than by choosing a combination of variation points.

15.2.8 Using Product Lines as Tools for What-If Analysis

Product lines can be used to capture variations across related products and across different versions of the same product. Another application of product lines is to capture design alternatives. Design is often an exploratory process—it involves making decisions and trade-offs that will affect the properties and construction of the target products. Questions arise: Should we include this component or that component? Should this part of the application run on the client or on the server? Often, it will be unclear which path is best, so multiple alternatives are considered and pursued until more information becomes available.

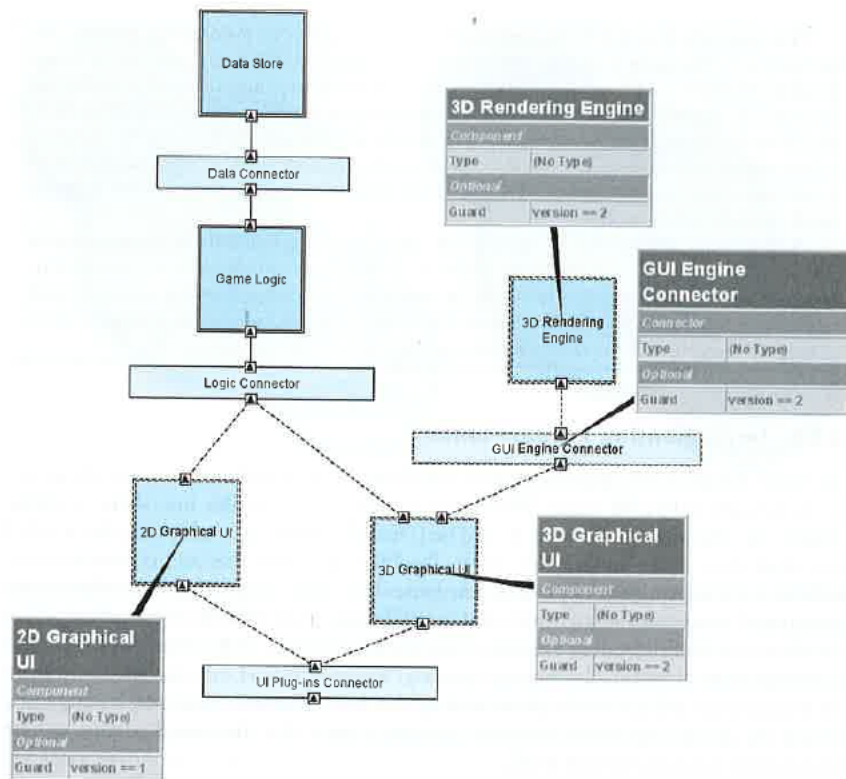


Figure 15-25.
Product line of
Lunar Lander
games where each
product is a
version of Lunar
Lander.

When this happens, product-line techniques can be used to simultaneously capture alternative architectures. Here, each product is not a system that is intended to be built, but one of many potential product designs. Points of variation in these product lines are decision points. The selection process can be used as a form of “what-if” analysis.

Software engineers have had to deal with variations, evolving artifacts, forking, branching, merging, and related concepts for years, particularly in the area of source code. In general, they are assisted in these regards by software configuration management systems. Configuration management systems can also be used to help address these concerns when dealing with artifacts such as architectural models, especially when the artifacts themselves offer no explicit support for these concepts.

One simple approach is simply to put architecture models into a version control system such as RCS (Tichy 1985), CVS (Fogel 1999), Subversion (Collins-Sussman, Fitzpatrick, and Pilato 2004), or ClearCase (Atria Software 1992). This is a viable option for versioning architectures as a whole, and it helps track the evolution of architectures over time. To some extent, alternatives can be created through branching revisions.

Configuration Management of Architectures

This approach does not work particularly well for creating or maintaining product-line architectures. Managing a product-line architecture means establishing a core set of design decisions and then combining those core decisions with variation points consisting of additional decisions. In general, design decisions are spread throughout and across architectural models, and this is why architecture description languages such as Koala and xADL embed the variation points within models. Because version control systems generally work at the level of files, it is difficult to use them to manage elements within files—for example, a component specified inside an architectural model.

A compromise is to attempt to separate out core parts of architectures and variation points into separate files, and then manage the evolution of these elements independently in a version-control system. This works in theory, but in reality it is difficult and inconvenient to have related design decisions for a single product-line member scattered across files, especially when automated merging tools are not available to merge them together again.

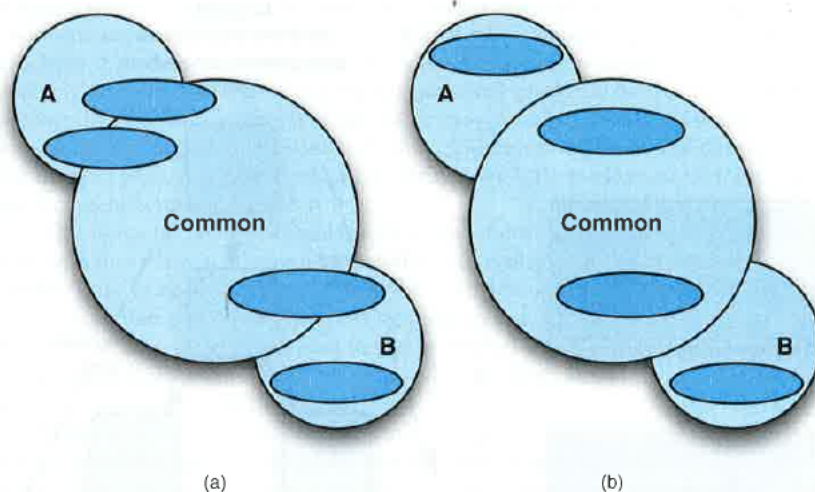
15.2.9 Implementing Product Lines

The major reductions in implementation costs from using a product line come about primarily through extensive reuse: Where two products in a product line share a similar architecture, the implementations should be (virtually) identical. Achieving this level of reuse must start in the architecture. Here, flexibility in connectors and communication methods is a primary driver of product-line reuse. If an architecture is chosen where components and connectors are tightly bound, it will be extremely difficult to introduce points of variation without recoding existing components. However, if flexible connectors and communication styles (such as message passing) are used instead, it is easier to introduce variations in the architecture without making extensive changes. However, having flexibility at the architectural level does not necessarily mean that the same flexibility will be present at the implementation level.

Chapter 9 discussed how implementing a system built using software architecture techniques is largely a mapping problem: understanding and controlling how design decisions made in the architecture map to implementation artifacts. In general, all the advice in Chapter 9 applies here as well. Implementing a product-line architecture is still a mapping problem, except that multiple products composed of core elements and variation points are involved. In this case, separating concerns in the implementation artifacts along the boundaries of variation points in the product line is critical.

Consider the situation presented in Figure 15-26. In Figure 15-26(a), good separation of concerns has not been maintained in the implementation. Various source files (File1.java, File2.java, and File3.java) are responsible for implementing both core and product-specific concerns. In this case, neither Product A nor Product B can be constructed without importing at least part of the implementation of the other. This makes it hard to evolve the products independently, and to integrate new products into the product line. Figure 15-26(b) shows a good separation of concerns: Here, implementation artifacts corresponding to products are kept separate from the common core and from each other. Both products can be built without using artifacts that (partially) implement other products.

One way of maintaining this separation of concerns is to use flexible architecture implementation frameworks, as discussed in Chapter 9. Many implementation frameworks help to enforce loose coupling between component and connector implementations, such

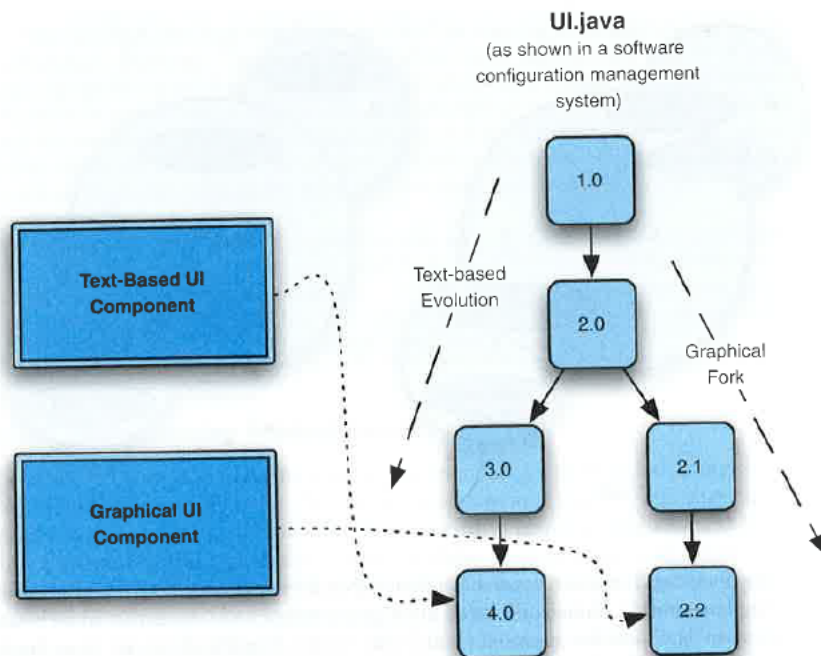


that components and connectors can have their dependencies changed easily and automatically. Implementation frameworks often allow components and connectors to be bound to each other “late”—either at product build time or even dynamically at run time. Implementations that employ late bindings of components generally enforce better separation and independence of components and help to drive reuse. Interface-implementation separation is also important: Well-defined implementation-level interfaces help to establish the services that are provided and required by a component (or set of components) without requiring a particular implementation. This separation of concerns further increases flexibility: Changes to the implementation of one component will not require changes to other components if the contract of the interface is maintained.

This is not always easy to achieve. Often, legacy products are incorporated into a product line. These products may not use flexible components, connectors, and communication methods. In this case, a careful refactoring should be considered, driven by the product-line architecture. Tight bindings in a product line are workable as long as they do not interfere with or cross the points of variation. For example, if three core components that are part of every product in the product line are tightly coupled, this is not a major problem: These components never will be separated, so making their bindings to each other flexible will be of limited near-term value. Instead, developers and architects should focus on making the bindings to components that exist only in a few products more flexible.

Product Lines and Source-Code Management Systems

Any mature development project will leverage some form of configuration management tool set to keep track of different versions of lower-level implementation artifacts: code files, resource files, and so on. Successfully managing product-line implementations requires an understanding of the relationship between versions of architecture-level elements (components, connectors) and versions of implementation-level elements (code, resource files)



as stored in a configuration management repository. Implementations that are cleanly separated along the lines of architectural elements greatly simplify the mapping problem.

One phenomenon that occurs more often in product-line development than in single-product development is the use of multiple versions of the same implementation artifact (for example, source file) in different places in the product line architecture. For example, recall the Lunar Lander product line shown in Figure 15-23. Perhaps, for whatever reason, the implementation of the text-based user interface and the graphical user interface were based on the same code, a single file called UI.java, that has evolved over time. At some point in the past, a fork of UI.java was created in the configuration management repository, without renaming the file. Thus, there are two development branches of UI.java: one for a textual UI and one for a graphical UI.

This situation is depicted in Figure 15-27. Here, the *Text-Based UI* component uses version 4.0 of UI.java, while the *Graphical UI* component uses version 2.2. In a situation like this, implementation mappings must not only refer to individual files, but also individual versions of individual files.

15.2.10 Unifying Product Architectures with Different Intellectual Heritage

Unfortunately, product lines are not always considered at the inception of a group of related products. The idea of using product lines often appears on an organization's radar only after

it has developed many products that have similar functionality and has duplicated an enormous amount of effort in doing so. Alternatively, company mergers and acquisitions can leave a single organization with the responsibility for developing, maintaining, and evolving very similar (and at one time, competing) products. At this time, the use of product lines becomes extremely attractive as a way of cutting costs by reusing engineering knowledge and architectural decisions from one product to the next. The difficulty is that the existing products in the product line often share little intellectual heritage, and the product architectures differ widely.

This is one of the most difficult problems in product-line development. The primary tension is that the goal of using a product line is to enable reuse, but the existing products exhibit little to no reuse. When this is the case, attempting to develop a product-line architecture that includes all the existing products may be of limited value, except as a historical artifact. Instead, processes should be forward-looking, attempting to extract the architectural lessons of previous projects and unify them into a product line for future products (or future product versions).

Before a product-line architecture can be developed, individual product architectures must be obtained. Architecturally savvy organizations already will have documented architectures for their products, but, as we have discussed, many product architectures evolve organically or diverge from their original intentions. In this case, architectural recovery (discussed in Chapters 3 and 4) is the first step toward product-line development.

The next step of the process is to normalize the existing product architectures (if necessary). Often, even when architectures for each product are available, they may not capture the same aspects of the systems under development, or capture those aspects at the same level of detail. They may not use the same terminology to refer to similar concepts. When this is the case (and it nearly always is) some amount of normalization must occur—some of the architectures must be adapted so that the commonalities and differences among the product architectures can be more readily identified. Luckily, architecture elaboration in this situation is made easier by the fact that there are actual, implemented products from which to extract the additional necessary information.

When normalized product architectures are available, they can be compared to identify common aspects and points of variation. Unless all of the architectures are expressed in the same notation at the same level of detail, it is probably not possible to automate much of this process. Instead, it is more useful to gather the architects and other stakeholders from each product into a series of meetings. Developing tables similar to Table 15-1 and Table 15-2 in order to understand the overlap between features and elements in the various products may be a useful exercise.

Developing a product-line architecture from this point forward is then a stakeholder-centric activity. There is no one right way to go about this, and depending on the domain and the organizational goals, the effort may go in one of many different directions:

No Product Line: It may be determined that, after examining the architectures, less reuse is possible than previously imagined. Perhaps the products were thought to perform many similar functions, but in reality they do not. Alternatively, the products may be so architecturally different that attempting to unify future versions of the products into a product line would cost more than the potential reuse would save. Here, the best alternative might be to abandon the idea of developing a product line

entirely, and focus on further distinguishing the products from one another in separate development efforts.

One Master Product: Often, the easiest course of action in developing a new product line is to identify the best existing product and use its architecture as a basis for a product line. In this way at least one product will fit naturally into the emergent product line, and innovative features from other products can be integrated as appropriate.

Hybrid: When no one product can be identified as the master for the purpose of developing a product line, the “best of breed” features of each product in the domain can be extracted and unified into a hybrid product line. Here, no one existing product will fit totally naturally into the product-line architecture. This is most useful when new products are not intended to be direct descendants of old products or when existing products are being unified.

15.2.11 Organizational Issues in Creating and Managing Product Lines

This chapter has largely covered the creation and management of product lines and product-line architectures from a technical perspective. However, using product lines is not a solely technical activity. A product-line-based approach needs to be as much organizational and cultural as it is technical. Creating a product line often means integrating the needs and opinions of stakeholders from fundamentally different perspectives into the same effort. This can foster all kinds of organizational and technical issues: friction between the teams, feature creep, difficulty prioritizing customers, least-common-denominator architectures, and so on.

Furthermore, an organization's investment strategy in its products needs to change to match a product-line approach. Creating a product line incurs substantial costs that do eventually pay off, but not in the context of a single product. Rather, they are recovered through lower maintenance expenses across multiple products, the increased profits from selling more product variants, and the lower costs of developing future versions of the products. This requires organizations to think *horizontally*, across product and team boundaries, which often requires both an organizational and cultural shift.

Paul Clements and Linda Northrop have written extensively about these issues in their book *Software Product Lines: Practices and Patterns* (Clements and Northrop 2001); we encourage those interested to look to this and other sources for more detailed guidance in this regard.

15.3 DSSAS, PRODUCT LINES, AND ARCHITECTURAL STYLES

We have discussed domain-specific software architectures (DSSAs) and product lines, as well as their relationship to each other. A natural question arises at this point: What are the relationships among DSSAs, product lines, and architectural styles?

Recall that architectural styles represent a reusable package of design decisions that can be applied to different products to elicit desired qualities. In some sense, both DSSAs and product-line architectures can be seen in the same light. Usually, however, architectural styles are far more general than DSSAs or product lines. Most architectural styles can be applied across many domains. For example, the pipe-and-filter style has been used to develop