

Table 4-1
(Continued)

Style Category &			
Name	Summary	Use It When ...	Avoid It When ...
Peer-to-Peer			
	Peers hold state and behavior and can act as both clients and servers.	... peers are distributed in a network, can be heterogeneous, and mutually independent. ... robustness in face of independent failures and high scalability required.	... trustworthiness of independent peers cannot be assured or managed. ... designated nodes to support resource discovery unavailable.
More Complex Styles			
C2	Layered network of concurrent components communicating by events.	... independence from substrate technologies required. ... heterogeneous applications. ... support for product-lines desired.	... high-performance across many layers required. ... multiple threads are inefficient.
Distributed objects	Objects instantiated on different hosts.	... objective is to preserve illusion of location-transparency.	... high overhead of supporting middleware is excessive. ... network properties are unmaskable, in practical terms.

are often ignored, with a frequent willingness to compromise the quality and longevity of a system in order to, for example, decrease a new version's time to market. This attitude, coupled with the complexity of the involved systems and the sloppiness with which the changes to them are often documented, directly contributes to numerous recorded cases of architectural erosion.

Evolving a system with an undocumented, or documented but degraded, architecture poses tremendous challenges to engineers. It results in a real danger that the modifications intended to provide new functionality will be implemented incorrectly and those intended to remove a particular problem in the existing system will cause other, unforeseen problems. To deal with this issue, researchers and practitioners typically have engaged in architectural recovery, a process whereby the system's architectural design is extracted from its other artifacts, which are likely to be updated more reliably than the design itself. These artifacts may include requirements, formal specifications, test plans, and so on. However, most frequently, design recovery uses a system's implementation as the starting point.

Simply put, then, the task of design recovery is one of examining the existing code base—both source code and possibly binary files for any externally developed functionality that is reused off-the-shelf—and determining the system's components, connectors, and overall topology. As with other architectural tasks, the goals of recovery will be viewpoint-specific.

A common approach to architectural recovery is *clustering* of the implementation-level entities into architectural elements. Based on the approach used for grouping source

code entities, such as classes, procedures, or variables, software clustering techniques can be divided into two major categories: *syntactic* and *semantic* clustering.

Syntactic clustering focuses exclusively on the static relationships among code-level entities. This means that the design recovery activity can be performed without executing the system. The functional characteristics or meaning of the code-level entities are not taken into consideration. Instead, the entities are grouped together based on naming conventions or the existence of particular relationships between them. For example, if source code analysis shows that one class is encapsulated inside another, the two classes can be grouped together to represent (a part of) a single system component. More examples of relationships of this kind include variable and class references, procedure calls, use of packages, association and inheritance relationships among classes, and so on. In addition, syntactic clustering approaches can embody intercomponent (aka coupling) and intracomponent (aka cohesion) connectivity measures.

The major drawback of syntactic clustering is that it may ignore or misinterpret many subtle relationships and interactions among the implemented system's entities. This is particularly the case with *dynamic* information, such as the interaction frequency between two classes or the actual method invoked in the case of polymorphism. This problem is remedied by semantic clustering, which includes all aspects of a system's domain knowledge and information about the behavioral similarity of its entities. This means that all implementation constructs with similar functionality can be grouped together to form a software component. For example, all classes that provide support for checking the users' authenticity can be clustered into a single component. Although this approach may provide more meaningful components, it requires interpreting the system entities' meaning, and possibly executing the system on a representative set of inputs. In order to infer the necessary information, semantic clustering may also require that the system be instrumented and monitored during execution. Another potential difficulty is that semantic clustering techniques tend to be domain- or application-specific, so that their rules cannot be easily applied to an arbitrary system.

Note that, even if correct and complete structural and behavioral information about the system were available to a clustering technique, a key challenge that remains is recovering design intent and rationale. For example, it may not be obvious which of the recovered architectural elements were meant by the system's designers to play a central role in the system. This information may be obscured by the implementation. It may also depend on the architectural style(s) used to guide the system's initial design; at the same time, it may not be clear from the implementation what those styles were and whether they are still appropriate. Without this information, it may be difficult to separate principal characteristics of the system from the less critical and even accidental ones.

An excellent example of fine work in performing architectural recovery is given by the Apache Modeling Project at the Hasso-Plattner-Institute for Software Systems Engineering in Potsdam, Germany. The recovery effort focused on the Apache HTTP server, which is the most widely used HTTP server worldwide. The project used the Fundamental Modeling concepts (FMC) approach. The high quality of the work done and the documentation produced is attested by praise given to the project by the Apache developers—who were not involved in the recovery project. Details can be found at (Gröne et al. 2004).

A similarly excellent recovery project focused on the Linux operating system and compared its recovered architecture with its prescriptive architecture (Bowman, Holt, and Brewster 1999).

**The Apache
Architecture
Recovery
Project**

4.4 ARCHITECTURAL CONCEPTION IN ABSENCE OF EXPERIENCE: UNPRECEDENTED DESIGN

The primary focus of this chapter has been to indicate how to identify a feasible set of “alternative arrangements for the design as a whole” through the application of experience. Refined experience as codified in architectural patterns and styles has been a major focus, since they are so broadly and commonly useful. Even when experience has not been codified, it may still be found in preexisting systems, hence the discussion of design recovery. Novel design challenges do exist, however. As designer J. Christopher Jones (Jones 1970, pg. 24) said, “. . . the principle of deciding the form of the whole before the details have been explored outside the mind of the chief designer does not work in novel situations for which the necessary experience cannot be contained within the mind of one person.” At a minimum, if a designer is ignorant of relevant experience, for that designer the new problem is indeed a novel one.

It should be obvious that the first effort a designer should make in addressing a novel design challenge is to attempt to assure himself that it is genuinely a novel problem. The cost differences are the reason: If experience can be applied, a much quicker and cheaper route to a solution is available. One risk is the “siren song of novelty.” Tackling novel problems is fun; the added challenge and the opportunity to display creativity is a winsome combination. So, be careful to assess a problem carefully—the sheer delight of a new problem can be enjoyed most delectably if the novelty is indeed genuine.

Assuming that the designer has made a good-faith effort to map a new problem to existing solution strategies and has concluded that the problem presents sufficient differences to demand a greenfield effort, a variety of techniques are available to assist the designer. The essence is to adopt a design strategy and to control that strategy.

Basic Strategy

Failure to meet the needs of a new design challenge with approaches based upon past experience demands a rethinking of the problem and approaches to its solution. The multitude of authors who have discussed this situation—which exists in all fields of design, whether software-focused or not—essentially recommend the following process:

1. Divergence
2. Transformation
3. Convergence

Divergence is a step taken to shake off the confines of inadequate prior approaches and discover, or admit, a variety of new ideas, conceptions, and approaches that offer the promise of a workable solution.

Transformation is a combination of analysis and selection: based upon the information from the divergence step, solution possibilities and new understandings of or changes to the problem statement are examined and formulated.

Convergence is the step of selecting and further refining ideas until a single approach is selected for further detailed development.

As Jones said:

... the aim of divergent search is to de-structure, or to destroy, the original brief while identifying these features of the design situation that will permit a valuable and feasible degree of change. To search divergently is also to provide, as cheaply and quickly as possible, sufficient new experience to counteract any false assumptions that the design team members, and the sponsors, held at the start.

(Jones, p. 66)

There are both difficulties and risks in the critical first step. One difficulty is simply breaking free from prior understandings and approaches to the problem area. The desire to cling to the old tried and true approaches is common both in individuals and in organizations. The old approaches were the basis for whatever success the person or company has achieved to date, so moving into new territory likely will be viewed skeptically. Examples abound in ordinary life; whether the topic is new transportation systems, medical treatments, or communication technologies, new approaches often have difficult entry into practice.

The flip side of not adequately stepping away from better-understood approaches and techniques is not anchoring wide exploration in assessments of costs and benefits. Suppose one hundred new solution possibilities emerge from the divergence step. Should all be explored fully, to determine which one is best? Supposing one thousand possibilities emerge? The key is that the designer must make realistic judgments of the magnitude of the perceived penalties for not exploring a given alternative. The cost of analyzing a given approach must not exceed the value obtained by performing the analysis.

Yet another risk is tunnel vision. This occurs when an approach is explored that breaks sharply from previous practice, but enthusiasm for that one approach stifles any attempt to explore other novel alternatives. It is a difficult balance.

While much has been written about the activity of design and creative processes, most of that literature has come from the industrial design community or from academics. It is interesting to see how individuals who come from entirely different backgrounds come to very similar conclusions. John Boyd, a fighter pilot in the U.S. Air Force in the 1950s, went on to designing combat tactics, then combat aircraft, and ultimately to widescale military strategy. Toward the end of his long career he wrote the essay, "Destruction and Creation," which addresses the issue of designing solutions for novel problems. He formulated his approach as a two-step process:

1. Destruction/unstructuring/destructive deduction
2. Synthesis/restructuring/creative induction

He summarized these as follows:

Also, remember, in order to perform these dialectic mental operations we must first shatter the rigid conceptual pattern, or patterns, firmly established in our mind. (This should not be too difficult since the rising confusion and disorder is already helping us to undermine any patterns.) Next, we must find some common qualities, attributes, or operations to link isolated facts, perceptions, ideas, impressions, interactions, observations, etc., together as

**A Fighter
Pilot on
Design**

possible concepts to represent the real world. Finally, we must repeat this unstructuring and restructuring until we develop a concept that begins to match-up with reality.

[John Boyd, "Destruction and Creation," reprinted in (Coram 2002)]

The additional insight shown here, as compared to the three-step process in the main text, is the focus on repeatedly cycling through the activities of destruction and synthesis, until a feasible solution emerges. Note as well the implied focus on understanding and formulating the problem in new terms and in new ways.

Detailed Strategies

Numerous detailed strategies for helping a designer tackle a novel problem have been articulated in books and articles over a long period of time. Most strategies are general and are appropriately focused on the divergence step. They are not specific to software development. This does not diminish their utility, however, as cookie-cutter solutions are by their very nature insufficient for novel situations. The need for creative, deep thought will be uncomfortable and unpleasant for some, but delightful for others. Jones (Jones 1970) catalogs thirty-five techniques; we have selected and adapted below a few of the most relevant for use in the software engineering context.

Analogy Searching. The idea of analogy searching is to examine other fields and disciplines unrelated to the target problem for approaches and ideas that are analogous to the problem at hand, and formulate a solution strategy based upon that analogy.

A common unrelated domain that has yielded a variety of solutions is nature, in particular the biological sciences. One arguable success from this approach has been neural nets: From understanding of how neural networks work in "wetware," that is, the neural systems of animals, ideas are offered for structuring software systems to parallel process certain types of data. Other domains have offered solution ideas such as flow architectures. Whether the inspiration was sewage systems, circulatory systems, or postal mail systems, a variety of software systems have flow structures and characteristics that reflect our common understanding of these physical systems.

Jones's discussion of analogy searching identifies four types of analogies: direct, personal, symbolic, and fantasy. The examples in the preceding paragraph are direct analogies: a concept in the domain of the analogy has a correspondent in the design domain.

Personal analogies are rather different; to quote Jones, "The designer imagines what it would be like to use one's body to produce the effect that is being sought, e.g., what would it feel like to be a helicopter blade, what forces would act on me from the air and the hub, . . ." (Jones 1970). While this type of analogy initially may seem to have limited usefulness in the domain of software architecture, one might consider its use in the development of intelligent systems that interact frequently and via various modes with a human user.

Symbolic analogies focus on metaphors and similes "in which aspects of one thing are identified with aspects of another" (Jones 1970). Perhaps the most obvious example of this in the World Wide Web, and especially with the notion there of broken links—the image of a damaged spider's web is very evocative.

Fantasy analogies are analogies to things which are wishes. A good, and useful, example is a sky hook—a device for lifting that attaches to the object to be carried on one end and to the air on the other end. Fantasy analogies can help to temporarily simplify a problem

such that the remaining parts can be more directly addressed. Naturally one does have to return eventually to the real world but perhaps by then some feasible strategy will be identified.

When talking about the architecture of buildings, every analogy has limitations and it is important to recognize when those limits have been reached. The utility of analogies is in providing rich sets of initial concepts for developing a solution to a design problem. Development of the details can proceed in absence of further consideration of the analogy.

Brainstorming. Brainstorming is the technique of rapidly generating a wide set of ideas and thoughts pertaining to a design problem without (initially) assessing the feasibility, desirability, or qualities of those ideas. After the ideas are generated, a period of categorization and analysis of the results begins. Brainstorming can be done by an individual or, more commonly, by a group. The primary association of groups with brainstorming is based on the belief that expression of an idea by one individual in a group will trigger a creative response from another. As a strategy it is well known.

As many people who have participated in brainstorming sessions have observed, however, potential problems with the technique are numerous. A brainstorming session can generate a large number of ideas—all of which are low-quality. Sessions can degenerate into “coblabberation” or “blamestorming,” to quote Harvard’s David Perkins.

Successful brainstorming appears to require advance planning and careful management. Having people come to the session with a set of ideas already prepared and written can speed the process, remove awkwardness, and provide some egoless participation. The use of external facilitators (meeting supervisors) can help keep the discussion focused and avoid personality-based clashes. Jones’s view is that the chief value of brainstorming is in identifying *categories* of possible designs, not any specific design solution suggested during a session. After a group brainstorming session is over, individuals may continue the process on their own, or the design process may proceed to the transformation and convergence steps.

Literature Searching. The classic design disciplines have used literature searching—the process of examining published information to identify material that can be used to guide or inspire designers—for decades. The advent of the Web, the ability to search electronically, and the availability of free or open-source software has breathed new life into this approach, however, making it especially useful.

Not only are all the historically useful ways of searching literature available, but digital library collections make searching extraordinarily faster and more effective. For software designers, the availability of the ACM Digital Library and IEEE Xplore offer a world of insightful, deep literature. More generally Google, Google Scholar, and other search engines offer the ability to examine a phenomenal base of information.

The importance of these digital resources is all the more important because, all too often, practitioners live in a small world with limited ability to interact with practitioners from other companies or with researchers—those people whose careers are devoted to investigating novel problems. There’s always enough time for yet another group meeting or sensitivity-training seminar, but never enough time to read a technical magazine, attend a conference, or take a professional development seminar. Digital library resources offer some respite from this isolation.

The availability of free and open-source software adds special value to this technique. Even if a direct solution to the problem at hand is not freely available, the designer may be able to devise a solution composed of large pieces of preexisting software, perhaps composed in a manner totally unanticipated by the original developers.

Last, developers should not overlook the use of the patent database as a source of ideas, approaches, and inspiration. While learning to read a patent takes a bit of time, the astounding wealth of ideas in the world's patent databases makes them valuable electronic resources.

Morphological Charts. The essential idea of morphological charts is to (1) identify all the primary functions to be performed by the desired system, (2) for each function identify a means of performing that function, and (3) attempt to choose one means for each function such that the collection of means performs all the required functions in a compatible manner. Put another way, the problem is subdivided into parts, a solution for each subpart is devised, then (it is hoped) a solution for the whole is created by combining the subsolutions to the various parts. The trick, of course, is that the functions and subsolutions need to be independent and compatible with each other. If the functions, or their subsolutions, are not independent then the designer is bound to consider the functions together. If the chosen subsolutions are not mutually compatible, then another combination of subsolutions must be examined.

At first glance, this technique seems to be nothing more than a fancy name for the old, and somewhat discredited, top-down design. Its value, however, derives from the fact that the technique does not demand that the functions be shown to be independent when starting out; similarly, there is no a priori requirement for the subsolutions to a given problem be developed under the constraint of being compatible with all the subsolutions to other functions. Recall the overall aim: divergence. The technique offers help in constructing a variety of approaches to parts of the overall problem. During the transformation step, the functions and their subsolutions may be reshaped such that ultimately, during the convergence phase, a compatible solution is devised.

Removing Mental Blocks. Getting stuck on a problem is nothing new. It is no surprise, then, to hear that many mental strategies have been promulgated to help designers. Perhaps chief among these is transformation. For instance, if you can't solve the problem, change the problem to one you can solve. If the new problem is close enough to what is needed, then closure is reached. If it is not close enough, the solution to the revised problem may suggest new avenues for attacking the original.

A variety of transformation strategies are available, many of which can be seen and applied in software architecture. Elements of a solution may be adapted, modified, substituted, reordered, or combined to offer new structures.

Jones advocates a strategy of reassessing the design situation. This gets somewhat to the issue of tunnel-vision as mentioned above, but has broad utility. "Matchett's insistence . . . upon continually returning to the 'Primary Functional need' (that which *must* be satisfied if the design is to be accepted) is probably the most reliable way of side-stepping a difficulty. In most cases the designer will be reminded that the sub-problems are of his own choosing and that he can satisfy the Primary Functional Need with an entirely different set of sub-needs if he changes his view of the problem." (Jones, 1970). Or indeed, if the problem itself is changed.

Additional strategies are detailed in Stephen Albin's book, *The Art of Software Architecture* (Albin 2003) and other references provided at the end of the chapter.

Controlling the Design Strategies

The potentially chaotic nature of exploring diverse approaches to the problem demands that some care be used in managing the activity. If the search for novel solutions is unconstrained, a great deal of time and money can be wasted. This suggests the following four guidelines (two of which are adapted from Jones):

1. Identify and review critical decisions (that is, perform risk analysis). By identifying the critical issues, and the consequences of choices regarding those issues, the designer's focus can be kept on track.
2. Relate the costs of research and design to the penalty for making wrong decisions; "the penalty for not knowing must exceed the cost of finding out." Making a wrong decision may not be the end of the world; if the issue is not of great consequence, then a suboptimal approach may be acceptable. To keep making progress it can be necessary to make a decision, even a potentially poor one, to enable all issues to be surfaced and addressed.
3. Insulate uncertain decisions. Decisions for which the design is unsure, or for which the circumstances dictating the issue may change, argue for carefully insulating that decision from the rest of the design. This is the standard dictum of software engineering due to David Parnas: Matters likely to change should be encapsulated inside boundaries (such as a component with resilient interfaces) so that when change eventually does come, the rest of the system is unaffected. (This is analogous to the concept of shearing layers in building architecture, a topic discussed later, in Chapter 14.)
4. Continually reevaluate system requirements in light of what the design exploration yields. It is the rare system whose requirements are genuinely firm and which cannot be improved, or changed, as the result of divergent design explorations. Costs for meeting requirements must be reexamined, and opportunities for novel, unanticipated functionalities should be pursued.

4.5 PUTTING IT ALL TOGETHER: DESIGN PROCESSES REVISITED

The concepts and techniques for use in the design of software architectures presented here have ranged from the very basic—reminders to use abstraction and separation of concerns—to extensive treatment of architectures, styles, and patterns that encapsulate knowledge gained from experience with many prior systems. The chapter also presented approaches for designing in situations where there is little or no experience upon which to draw.

Faced with a new development task, system architects must sort through all the issues confronting them, choose a design approach, and see it through. As we have noted, the central task is choosing a feasible set of concepts upon which to build the design, then progressively refining and revising the concepts and the ensuing details until a satisfactory architecture is developed.

Perhaps the hardest part of the whole process is knowing whether to attempt to base the new design on some existing design—use the “Grand Tool of Experience”—or to approach the design task as one that is unprecedented. Whether an application presents novel challenges or not, significant insight can come from iteratively working with requirements for the system. Hence we call to mind a few thoughts that were introduced in Chapter 2 and expand upon them briefly. Implementation concerns can also further shape and guide the design process, hence we include below a short section dealing with that topic.

A comprehensive methodology for applying all these concepts such that any arbitrary problem can be tackled in a straightforward manner is beyond the scope of this text—indeed, beyond what anyone has yet credibly set forth. The design process—a good design process—will take many twists and turns after the initial attempt at concept formation. But that’s part of the fun. You get to be creative.

4.5.1 Insights from Requirements

Chapter 2 pointed out the intimate relationship between requirements engineering and software architectures. Preexisting architectures provide a frame of reference for developing new requirements statements. Architectures provide:

- A *vocabulary*, not just of basic concepts, but of means, approaches, and possibilities.
- A *framework* for describing properties, such as by describing relationships between structural elements, or by categorization of types of design decisions.
- A *basis for analysis*, for example, through knowledge of previous design decisions and their consequences one can assess possible consequences of new requirements.

In many cases—perhaps most—new architectures can be created based upon experience with preexisting architectures and improvement to them.

Restated, the organic interaction between past design and new requirements means that many critical decisions for a new design can be identified or made:

- Experience can show what the most critical issues are, or what the most difficult problems are likely to be.
- Experience can suggest what the key levels of discourse are, and what vocabularies to use in each.
- Experience can show successful patterns of product specialization.
- Experience can show what architectural patterns and styles have been effective in the domain.
- Experience can show areas in which novel development has typically been required.

One aspect of this iterative conceptualization of the new application is in the identification of architectural tipping points—decisions that have profound consequences for the rest of the design and implementation. An important example is deciding whether part or all of the new application should be created as a modification of an existing system. This can be a very difficult choice: Modifying an existing system offers the potential of achieving substantial software reuse, with attendant cost and schedule benefits. Yet a time comes in the evolution of every system when achieving new requirements demands more of the preexisting architecture than it can bear, and hence an entirely new design is called for. Core

algorithms, for example, may not be capable of scaling to new performance requirements, thus requiring a new approach. All of these are important aids in getting started with the design of a new application; they derive from the symbiosis of articulating new requirements using a vocabulary of known architectural choices.

In those rare cases where a requirements specification is used that does not make any reference to existing architectures, careful analysis is required to identify the critical factors, to distinguish between the incidental and the essential. A highly dynamic approach to the first critical steps of identifying “a set of alternative arrangements for the design as a whole” is required in such cases. Several such approaches were identified at the end of Section 4.1. You may need to attempt to design a solution using several different structures, applying various styles from the discussion in Section 4.3, then compare the results and the properties obtained, in order to determine the best course to take.

Many touted design strategies are simply transformations of requirements into, in effect, simulations of some part of the real world. As such, the architecture is determined simply by the domain and particular manner of stating the requirements, and not by thoughtful, creative design or the use of sound software design principles.

One example is the “underline the nouns, underline the verbs” approach to design first promulgated by Russell J. Abbot (Abbott 1983). The essence is that the identified nouns determine a program’s data types (objects); the verbs become the names of the methods defined on those noun-objects. This approach to object-oriented design was popularized widely by Grady Booch. In (Booch 1986) he said, “Since we are dealing with a philosophy of design, we should first recognize the fundamental criteria [sic] for decomposing a system using object-oriented techniques: *Each module in the system denotes an object or class of objects from the problem space.*”

A similar tack was taken in the Jackson System Design (JSD) method: “From the late 1970s to the second half of the 1980s I worked with John Cameron and others on the JSD method of analysis and design for information systems. The method is based on separating the construction of a model (for example, in a database) of the real world from the construction of the information functions that extract and display the necessary information from the model. The model focuses chiefly on the behaviours of real-world entities over their lifetimes, representing each entity as one or more sequential processes: the local variables of these processes form the data part of the model of the entity. The method is described in my book *System Development*.” (Jackson 2007)

Design by
Simulation:
“Danger,
Will
Robinson!”

4.5.2 Insights from Implementation

A detailed discussion of techniques and approaches to use when moving from an architecture to an implementation is found in Chapter 9. Here we look at the feedback situation: how information and insights from the implementation domain, or from constraints on that activity, can or must be used in development of a detailed architecture.

First, constraints on the implementation activity may help shape the design. Externally motivated decisions may be principal, and hence be a part of the architecture. For example, external constraints may dictate the use of a particular type or brand of middleware (connector technology). A large project focused on software-defined radios, for instance, mandated the use of CORBA, even before any detailed designs for the radios had been completed. Accommodating such mandates can dramatically affect the system’s architecture—and not always for the better.

Similarly, externally motivated constraints may dictate the use of particular programming languages, development environments, or implementation platforms. The skill set of the staff devoted to the implementation may be limited. These, and other similar constraints, can shape the architecture, perhaps restricting the set of architectural styles that may be used.

A more interesting impact on design may come from considerations of software reuse. The potential cost savings from reusing software—whether internal to the development organization or from free/open source sites—are substantial. But imperfect correspondence between the available software with the prescriptive design can require design modifications. Mismatches possible include functionality (the desired function is not there, but a similar one is), interface (the desired functionality is present, but the manner of invoking it is different from the current design), and nonfunctional properties (for example, the desired functionality and the desired interface are present, but the software runs too slowly or requires too much memory, or has inadequate security features).

Presence of such a situation invites an iteration between the design process and the implementation activity. The most effective course of action may in fact be to change the design to enable easier reuse. The decision of whether to reuse code and the manner in which it should be used (such as encapsulated or modified) is a principal design decision grounded in implementation issues, and hence must be considered during the design process.

Just as development of requirements and development of design must proceed cooperatively and contemporaneously, so too may design and implementation. Initial implementation activities may yield critical performance or feasibility information, allowing the designer to proceed with confidence in other aspects of the system's design. At all costs, architectural erosion must be prevented.

4.6 END MATTER

Designing architectures is the occasion when the designer can most obviously employ and exhibit all his or her innate creativity and talent. Designing well is a skill to be learned and practiced. It requires being willing to take risks, explore dark alleys, and occasionally backtrack. Timid, staid approaches may yield software that satisfies most requirements, but seldom will yield applications that give satisfaction to the user and enable the designer's organization to dominate the marketplace.

The most effective approaches to design will be those that have been refined and seasoned within the domain of the new application. Knowledge of effective design techniques and trade-offs is an important corporate asset; ensuring that such knowledge is shared among projects and between designers should be an important managerial goal.

All the major conceptual elements of architecture-centric software engineering now have been introduced. What remains is placing techniques and tools in the designer's hands to enable productive exploitation of these conceptual elements. The following chapter begins this process by exploring in depth the concept of architectural connectors. As we have noted earlier, the use of components is familiar, but connectors are unfamiliar to many software engineers, and their depth and richness is surprising to most. This topic

is then followed by the lynchpin technology: explicit modeling of architectural decisions. With an architecture description language in hand the designer can proceed to design and document key decisions in a manner that supports the many development goals we have covered.

The essential business case for design is, "What's the alternative?" Creating products that are successful in the marketplace, effective to maintain, and that lead to future successful products demands a thoughtful approach to product design. Design of product families—such as the consumer electronics example discussed several places—is a key means for ensuring corporate growth through an expanding range of economically achieved products.

In addition to designing the software's structure, a company must equally deliberately design a product's user interface, interaction behavior, and "feel." Design of these domain-specific characteristics must proceed cooperatively with design of the software structures that implement them, a theme first emphasized in Chapter 2.

A company's design processes represent a central asset of an organization. The successful and efficient production of new products is at the heart of a company. Design carries corporate value, and processes and traditions outlive most products. Given inevitable turnover of personnel, maintenance of good design processes depends upon production of explicit designs and articulation of the rationale behind the design decisions underlying them.

Since designing is a central asset it should be no surprise that it is an activity that cannot be contracted offshore. Good design demands interaction with marketing, customers, domain experts, and business analysts. Once a design has been fully elaborated to crisp, low-level specifications, their implementation becomes a candidate for outsourcing, but even then caution is in order. The designers must be able to assure themselves that no principal design decisions will be made during implementation—and that can be a very difficult assurance to achieve.

Given the importance of designing to an organization, the training and development of designers throughout the organization should be a continual priority. If an organization sinks to a state where one or two master designers are allowed to dominate development, the organization becomes at risk. Design is teachable and is not the exclusive province of an elite few.

The Business Case

4.7 REVIEW QUESTIONS

- How can experience be used as a design technique? What must be true about experience in order for it to be useful in the design of a new system?
- Refined experience, in the form of architectural styles, for example, can appear and be applied at different levels of abstraction and at different degrees of domain specificity. What are some examples of different styles at different points in this space of abstraction and domain-specificity?
- What types of applications are appropriate for the following styles? Alternatively, describe an example application for each of the following styles: event-based, pipe-and-filter, layered, mobile code.
- Describe a good process for designing an application for which the development team has no prior experience in the domain of the application and no prior architecture or code from which to work.
- How can coding-related decisions, such as choosing a user-interface toolkit, affect an architecture?

4.8 EXERCISES

1. For many years design by stepwise refinement was viewed as an appropriate, effective strategy for program development. Today, it is viewed as quaint and inadequate. Explain what's wrong with the concept of stepwise refinement in the context of modern complex systems development.
2. Suppose an application has been in use by an organization for twenty years. It has been modified and expanded over that period to meet a steady stream of new demands. Now a new set of demands arise, requiring further changes. The software engineers in charge of the application are apprehensive about attempting to further modify the system to meet the new needs. How should the team decide whether to continue to try to modify the existing system or build a new application from the ground up?
3. Top-down or bottom-up? Under what circumstances would it make the most sense to attack the design problem for a large project by working initially on subparts of the problem—without being sure that any subassembly so designed would end up as part of the final design solution?
4. The numerous examples of designs for the lunar lander in the chapter are all sketches in that they do not include the details of how the various components communicate. Elaborate the lunar lander designs to the next level of detail, such that the designs could be handed over to a coder for final implementation. Assume a very simple textual user interface. State the assumptions made by the game in each example with regard to the use (or not) of a clock.
 - a. Layered architecture
 - b. Sense-compute-control pattern programmed as main-program-and-subroutines
 - c. 4-Object-oriented
5. Describe the architecture of a compiler in terms of architectural styles. Is it a single style as found in this chapter? Multiple styles composed together?
6. This chapter uses a lunar lander to illustrate how many different architectural styles may be applied to the same problem. Choose another application (such as a desktop calendar application, a cell phone-based contact and phone number manager, an automated teller machine system, or a controller to be used to control a set of elevators in a building), and present sketches of three different solution architectures, where each architecture exhibits a different style drawn from the set discussed in this chapter.
7. What's the architecture of Microsoft Word? How can you tell? What would you need to see in order to know?
8. Examine the architecture of the Apache HTTP server as characterized by the Apache recovery project (Gröne et al. 2004). Which styles are involved, and how are they combined?
9. Examine the architecture of Linux, as characterized by Bowman (Bowman, Holt, and Brewster 1999). Which styles are involved, and how are they combined?
10. Some designers have argued that "PowerPoint architectures" promote good design, in particular, that the discipline of working within a confined frame (for instance, the amount of space you have on a single PowerPoint slide) force the designer to apply abstraction and hierarchy. Discuss the pros and cons of PowerPoint architectures.

4.9 FURTHER READING

Design has been a topic of study outside of computer science and software engineering for many years. The book by Jones (Jones 1970) cited several times in this chapter is an excellent example of insight about the design process as seen from the industrial design perspective. A revised version was issued in 1992 (Jones 1992). A more recent book, focusing on group and corporate pro-

cess, comes from the founders of the IDEO design firm, (Kelley, Littman, and Peters 2001).

As mentioned at the end of Chapter 1, many excellent books on architectural design have been written, of course, but few provide substantive insights for software developers. Relative to this chapter, however, an important exception is Christopher Alexander's *The Timeless*

Way of Building (Alexander 1979) and a companion book, *A Pattern Language: Towns, Buildings, Construction* (Alexander, Ishikawa, and Silverstein 1977). As mentioned earlier, his work has influenced thinking on patterns for object-oriented programming, and more generally, software architecture styles. Another key classic in the study of design is *The Reflective Practitioner* (Schön 1983), by Schön, who characterizes design as an on-going conversation between the designer, the materials, and the context for the design.

A broad overview of *software* design as a research field—where it has been and where it is going—is found in “Software Design and Architecture: The Once and Future Focus of Software Engineering” (Taylor and van der Hoek 2007). This paper includes a large bibliography of software design books and papers, and constitutes an important reference and starting point for detailed investigations.

The model-view-controller pattern was developed in the late 1970s; an original paper plus a retrospective can be found in (Reenskaug, 1979, 2003). Numerous resources are available on the Web to further demon-

strate its application. The mobile code styles are explored nicely in a paper by Fuggetta, Pico, and Vigna (Fuggetta, Picco, and Vigna 1998). The C2 architectural style has been described and illustrated in the literature in several places, including (Dashofy, Medvidović, and Taylor 1999; Medvidović, Oreizy, and Taylor 1997; Robbins et al. 1998; Taylor et al. 1996). CORBA references abound. Perhaps the best introduction to the concepts is found in Emmerich’s book, *Engineering Distributed Objects* (Emmerich 2000). Abundant source materials can be found on the Object Management Group’s Web site (www.omg.org).

Numerous resources are available in print and online, documenting and explaining the U.S. Apollo missions to the moon. Many of these resources are available directly from NASA as well as secondary publishers, and includes “Apollo 11 The NASA Mission Reports” (*Apollo 11: The NASA Mission Reports* 1999). An excellent introduction to the onboard software and a recounting of the errors that occurred and were overcome during the Apollo 11 Lunar descent can be found in (Eyles 2004).

Connectors

Designing and implementing the *functionality* and managing the *data* of modern large, complex, distributed, multilingual software systems is undoubtedly very difficult. As discussed in the preceding chapters, early software architectural models of system components, their key services, their abstract behaviors, and their nonfunctional properties are indispensable in this endeavor. However, experience has shown time and again that making architectural design decisions that pertain to *integrating* those components effectively and ensuring the components' proper *interaction* in a system is just as important. Moreover, arriving at such design decisions can be even more challenging than those restricted to the development of functionality. The particular importance in modern systems of mechanisms for assembling functional components and supporting their interaction, that is, software *connectors*, mandates that they be treated and studied separately. We will do just that in this chapter, and will return to this subject repeatedly throughout the remainder of the book.

Simply put, software connectors perform transfer of control and data among components. Connectors can also provide services, such as persistence, invocation, messaging, and transactions, that are largely independent of the interacting components' functionalities. These services are usually considered to be "facilities components" in widely used middleware standards such as CORBA, DCOM, and RMI. However, recognizing these facilities as connectors helps to clarify an architecture and keep the components' focus on application- and domain-specific concerns. Treating these services as connectors rather than components can also foster their reuse across applications and domains. Perhaps most importantly, connectors allow architects and engineers to compose heterogeneous functionality, developed at different times, in different locations, by different organizations. As such, connectors can be thought of quite appropriately as the guards at the gate of separation of concerns.

A common misconception about software connectors is that they are just calls between two components, in the manner of, say, a Java method invocation. A slightly more advanced view, inspired by middleware-based system development, is that component

interactions take place via message passing and remote procedure calls (RPC). While it is frequently the case that two components will eventually be implemented to communicate using, say, a Java method call, it is important to realize that as a software system's architect you are not restricted to assuming and relying on only such primitive interaction mechanisms.

Traditional software engineering approaches embrace a rather narrow view of connectors that is unlikely to be successfully applicable across all development situations. This problem is further exacerbated by the increased emphasis on development using large, off-the-shelf components originating from multiple sources. As components become more complex and heterogeneous, the interactions among them become more critical determinants of system properties. Experience has shown that integrating components with mismatched assumptions about their environment is hard to do and can lead to many problems. It is the task of connectors to mitigate such mismatches.

An architect has a wide range of connector options that are applicable, and often tailorable, to different development needs and scenarios. For example, an architect can choose to use a connector that distributes a service request to specifically named recipient components or a connector that broadcasts a notification of a locally occurring event to any (unnamed and even unknown) component that may be interested and listening. The connector may require the sender component to suspend its processing until an acknowledgment of its request is received, or it may allow the sender component to continue with its processing; the connector can route each request in the order it received it, or it can try to order, filter out, or combine requests according to some prespecified rule; and so on. Therefore, a very important dimension of a software architect's job is to:

- Understand the component interaction needs.
- Carefully identify all of the relevant attributes of that interaction.
- Select the candidate connectors.
- Assess any trade-offs associated with each candidate.
- Analyze the key properties of the resulting component-connector assemblies.

The goal of this chapter is to provide insights, guidelines, and specific techniques to support accomplishment of these tasks. The remainder of the chapter will introduce:

1. The different roles connectors play in a software system.
2. Different connector types a software architect has at his or her disposal and the roles each type can fulfill.
3. A panoply of variation points for each connector type.

4. A set of general hints and guidelines about a connector's applicability, strengths, and drawbacks.

Throughout the chapter, we will provide examples of specific software connectors to illustrate the discussion. The reader will probably be familiar with many of these connectors, although sometimes under a different name. The chapter starts off with a simple concrete example, to set the stage for the subsequent discussion.

This chapter provides a detailed treatment of connectors. The type and amount of information provided is necessary for their thorough study. At the same time, some of the content may be unsuitable for an inexperienced reader. We recommend that everyone read at least Sections 5.1 through 5.4 as well as Sections 5.7 through 5.10. Section 5.5 contains a set of examples from the domain of large-scale data distribution systems, which can be followed by most readers, but their full appreciation may require prior experience with these types of systems. Section 5.6 is appropriate for all readers, but is targeted, in particular, at the experienced professionals and readers interested in practical guidelines for constructing advanced connectors.

5 Connectors

5.1 Connectors in Action: A Motivating Example

5.2 Connector Foundations

5.3 Connector Roles

5.4 Connector Types and Their Variation

Dimensions

5.4.1 Procedure Call

5.4.2 Event

5.4.3 Data Access

5.4.4 Linkage

5.4.5 Stream

5.4.6 Arbitrator

5.4.7 Adaptor

5.4.8 Distributor

5.5 Example Connectors

5.5.1 Event-Based Data Distribution Connectors

5.5.2 Grid-Based Data Distribution Connectors

5.5.3 Client-Server-Based Data Distribution Connectors

5.5.4 P2P-Based Data Distribution Connectors

5.6 Using the Connector Framework

5.6.1 Selecting Appropriate Connectors

5.6.2 Detecting Mismatches

5.7 End Matter

5.8 Review Questions

5.9 Exercises

5.10 Further Reading

Outline of Chapter 5

5.1 CONNECTORS IN ACTION: A MOTIVATING EXAMPLE

One useful property of connectors is that they are for the most part application-independent architectural elements. That means that they enable us to understand a lot about *how* a software system accomplishes its tasks without necessarily having to know exactly *what* the system does. In other words, connectors directly support two key principles of software

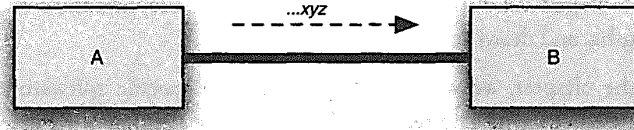


Figure 5-1. A simple pipe-and-filter architecture consisting of two filters, A and B, communicating via untyped data streams through the unidirectional pipe connector P.

engineering: abstraction and separation of concerns. Explicitly focusing on connectors also enables development and use of a specific vocabulary of software interaction. In turn, agreeing on appropriate terminology allows us to communicate easily and reason precisely about a number of software system properties that derive from the system components' interactions. To illustrate this, we use a very simple example involving two different types of connectors. We intentionally use the terminology associated with those connectors without defining it, to draw attention to this new "language." The appropriate definitions and explanations will be given in the subsequent sections. Nonetheless, you will probably find that you are familiar with some, if not most, of this terminology.

Different views of a software connector are useful for different tasks. In order to model a system and communicate its properties, a high-level view is suitable. For example, an architect may make the following concise, but meaningful statement about the configuration shown in Figure 5-1: "Components A and B communicate via a Unix pipe." That statement may be accompanied by a formal specification of the pipe's overall behavior. However, such a high-level description does not help us understand all the properties of the pipe, how it can be adapted, or under what conditions it can be replaced with another type of connector. A more detailed, lower-level view is needed to accomplish that.

In particular, the pipe in Figure 5-1 allows interaction via unformatted streams of data. It is a simple connector: It consists of a single interaction channel, or duct, and facilitates only unidirectional data transfer. Thus, the cardinality of the pipe is a single sender and a single receiver. You will recall from the preceding chapters that a pipe allows components (that is, filters) A and B to exhibit very low coupling: The components do not possess any knowledge about one another. For example, A's task is only to successfully hand off its data to the pipe; the actual recipient (if any) is unimportant to A. In turn, the particular pipe depicted in Figure 5-1 does not buffer the data. It attempts to deliver the data at most once; if the recipient is unable to receive the data for some reason, the data will be lost.

Let us now assume that we need to alter the manner in which A and B interact, such that B can also send information (for example, acknowledgement of data receipt) back to A. Furthermore, we want to ensure the delivery of data: If the recipient is not available, the pipe retries to send until the data is successfully transferred. Both these modifications can be potentially accommodated by pipes. The first modification would require introducing another pipe from B to A, and the second, data buffering in a pipe. Addition of any other components into the system would require further addition and/or replacement of pipes. The penalty we would pay is that these modifications may require substantial system downtime. Pipes can, therefore, accommodate these new

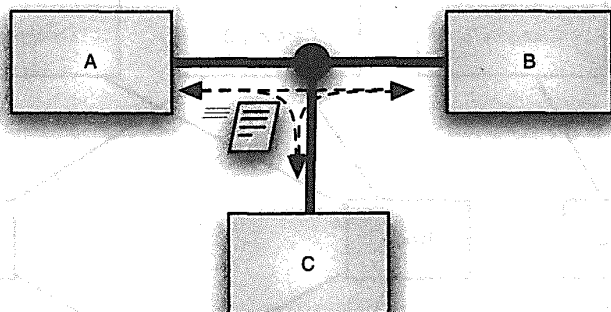


Figure 5-2. An event-based architecture consisting of two components, A and B, communicating via typed discrete data packets (events) through the event bus connector E. The connector allows on-the-fly system modifications, such as adding a new component C.

requirements, even though constantly adding and replacing them may not be the most effective solution.

If, however, we want to change the nature of the data from an unformatted byte stream to discrete, typed packets that can be processed more efficiently by the interacting components, pipes will not suffice. In such a case, an event bus connector will be a more suitable alternative. Although clearly different types of connectors, pipes and event buses exhibit a number of similar properties, such as loose component coupling, asynchronous communication, and possible data buffering. At the same time, event buses are better suited to support system adaptation: an event bus connector is capable of establishing ducts between interacting components on the fly; its cardinality is a single event sender (similarly to the pipe), but multiple observers. Thus, in principle, event buses allow components to be added or removed, and to subscribe to receive certain events, at any time during a system's execution. Figure 5-2 abstractly depicts the use of an event bus to accommodate the above changes.

5.2 CONNECTOR FOUNDATIONS

The underlying, elementary building blocks of every connector are the primitives for managing the *flow of control* (that is, changing the processor program counter) and *flow of data* (that is, performing memory access) in a system. These primitives give enough conceptual power to build sophisticated and complex connectors. In addition to these primitives, every connector maintains one or more *channels*, also referred to as *ducts*, which are used to link the interacting components and support the flow of data and control between them. A duct is necessary for realizing a connector, but by itself, it does not provide any additional interaction services. Very simple connectors, such as module linkers, provide their service simply by forming ducts between components. Other connectors augment ducts with some combination of data and control flow to provide richer interaction services. Very complex

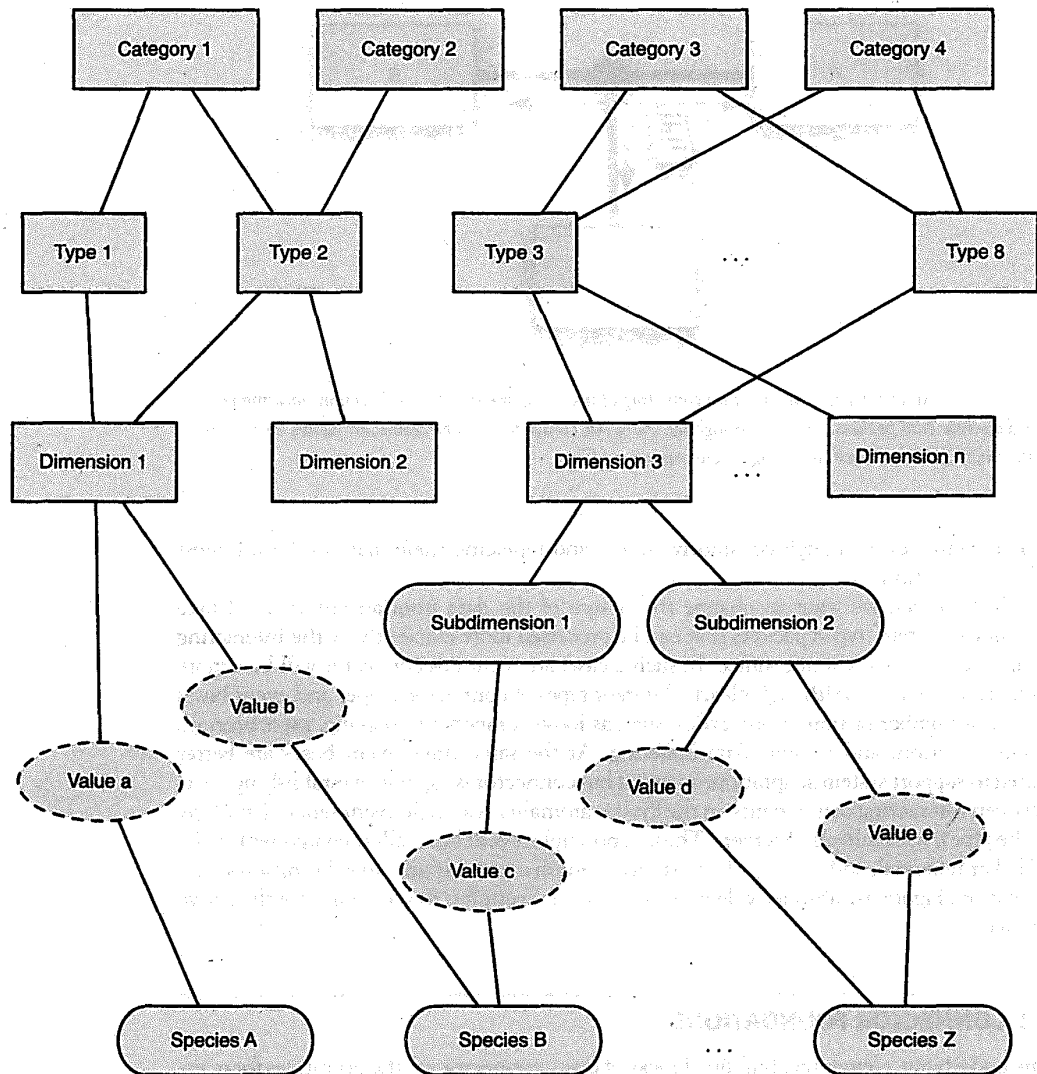


Figure 5-3. A framework for studying software connectors.

connectors can also have an internal architecture that includes computation and information storage. For example, a load balancing connector would execute an algorithm for switching incoming traffic among a set of components based on the knowledge about the current and past load state of components.

Simple connectors are typically implemented in programming languages. On the other hand, composite connectors are achieved through composition of several connectors (and

possibly components), and usually are provided as libraries and frameworks. Simple connectors only provide one type of interaction service, whereas composite connectors may combine many kinds of interactions. Complex connectors can help overcome the limitations of modern programming languages. However, when creating such connectors it is important to be able to reason about their underlying, low-level interaction mechanisms, identify the appropriate design choices, and detect potential mismatches among the interaction mechanisms used to compose a connector.

To this end, we will use the connector classification framework shown in Figure 5-3: Each connector is identified by its primary service category and further refined based on the choices made to realize these services. The characteristics most commonly observed among connectors are positioned toward the top of the framework, whereas the variations are located in the lower layers. The framework comprises service categories, connector types, dimensions (and possibly their subdimensions), and values for the dimensions. A service category represents the broad interaction role the connector fulfills. Connector types discriminate among connectors based on the way in which the interaction services are realized. The architecturally relevant details of each connector type are captured through dimensions, and, possibly, further subdimensions. Finally, the lowest layer in the framework is formed by the set of values a dimension (or subdimension) can take. Note that a particular connector instance (that is, *species*) can take a number of values from different types. In other words, this classification does not result in a strict hierarchy, but rather in a directed acyclic graph.

The remainder of this chapter describes in more detail the classification framework and a comprehensive taxonomy that can be used as the foundation for studying, classifying, and using software connectors.

5.3 CONNECTOR ROLES

A software connector can provide one or more of four general classes of services: communication, coordination, conversion, and facilitation. Put another way, a connector can play one or more of these four roles. Since these services, or roles, fully describe the range of possible software component interactions, the topmost layer in our classification framework from Figure 5-3 is the service category. Discussions and simple examples of the four service classes follow.

Communication

Connectors providing communication services support transmission of data among components. Data transfer services are a primary building block of component interaction. Components routinely pass messages, exchange data to be processed, and communicate results of computations.

Coordination

Connectors providing coordination services support transfer of control among components. Components interact by passing the thread of execution to each other. Function calls and method invocations are examples of coordination connectors. Higher-order connectors, such as signals and load balancing connectors, provide richer, more complex interactions built around coordination services.

Conversion

Connectors providing conversion services transform the interaction required by one component to that provided by another. Enabling heterogeneous components to interact with each other is not a trivial task. Interaction mismatches are a major hindrance in composing large systems. The mismatches are caused by incompatible assumptions made by components about the type, number, frequency, and order of interactions in which they are to engage with other components. Conversion services allow components that have not been specifically tailored for each other to establish and conduct interactions. Conversion of data formats and wrappers for legacy components are examples of connectors providing this interaction service.

Facilitation

Connectors providing facilitation services mediate and streamline component interaction. Even when components have been designed to interoperate with each other, there may be a need to provide mechanisms for further facilitating and optimizing their interactions. Mechanisms such as load balancing, scheduling services, and concurrency control may be required to meet certain nonfunctional system requirements and to reduce interdependencies among interacting components.

Every connector provides services that belong to at least one of these four categories. Commonly though, connectors provide multiple services in order to satisfy the need for a richer set of interaction capabilities. For example, procedure call, one of the most widely used software connector types, provides both communication and coordination services.

5.4 CONNECTOR TYPES AND THEIR VARIATION DIMENSIONS

Interaction services broadly categorize connectors, but leave many details unexplained. This level of abstraction cannot help us build new connectors, and it cannot be used to model and analyze them in an architecture. Hence, we further classify connectors into eight different types, based on the way in which they realize interaction services:

- Procedure call
- Event
- Data access
- Linkage
- Stream
- Arbitrator
- Adaptor
- Distributor

Connector types are the level at which architects typically consider interactions when modeling systems.

Simple connectors can be modeled at the level of connector types; their details can often be left to low-level design and implementation. On the other hand, more complex

connectors often require that many of their details be decided at the architectural level so that the impact of these decisions can be studied early and on a systemwide scale. Those details represent variations in connector instances and are treated as connector dimensions in the below classification. In turn, each dimension has a set of possible values. The selection of a single value from each dimension results in a concrete connector species. Instantiating dimensions of a single connector type forms simple connectors; using dimensions from different connector types leads to a composite (higher-order) connector species.

The remainder of this section will describe the key characteristics of each connector type as well as their variation points. Additionally, we will highlight the interaction role(s) played by each connector type. The discussion draws extensively from the connector taxonomy set forth by Nikunj Mehta, Nenad Medvidović, and Sandeep Phadke (Mehta, Medvidović, and Phadke 2000). It should be noted that the characteristics of different connectors introduced are meant to cover the entire space of connectors. However, it is possible that an individual connector of a given type may not possess each of that type's dimensions. The detailed connector characteristics presented below can be thought of as design templates that software architects can use to select the exact connectors needed in a given system. As will be discussed further in Chapter 6, several software architecture modeling notations support different variations of this selection process.

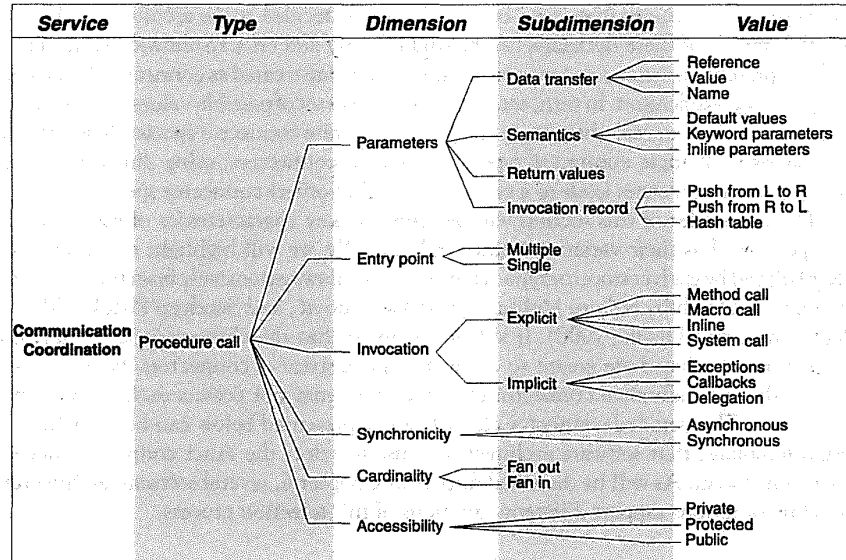
5.4.1 Procedure Call Connectors

Procedure call connectors model the flow of control among components through various invocation techniques. They are thus coordination connectors. Additionally, procedure calls perform transfer of data among the interacting components through the use of parameters and return values. They are thus also communication connectors. These connectors are among the most widely used and best understood connectors, and have been likened to the assembly language of software interaction. Examples of procedure call connectors include object-oriented methods, `fork` and `exec` in Unix-like environments, call-back invocation in event-based systems, and operating system calls. Procedure calls are frequently used as the basis for composite connectors, such as remote procedure calls or RPC, which also perform facilitation services.

The space of options available to a software engineer for constructing procedure call connectors is shown in Figure 5-4. Note that the values for certain dimensions and subdimensions are not shown: multiple versus single entry point, as well as fan in and fan out cardinality. These are numerical subdimensions that either take an obvious value (1 in the case of single entry point) or can take many values (in the case of the remaining three subdimensions). They are hence elided for simplicity; we likewise omit the values of all such dimensions and subdimensions in the case of the other seven connector types.

A typical programming language-level procedure call connector takes on a specific set of values from the choices depicted in the figure. For example, as most Java users will readily know, a procedure (that is, method) call provides the data transfer of its parameters by reference; it may have a return value unless the invoked method is declared as a `void`; it will have a single entry point, at the start of the invoked method; it will be a result of explicit invocation; its synchronicity will be blocking; its fan in and fan out cardinality will both be 1—that is, a Java method call always has a single source and a single destination; finally, its accessibility may be public or private.

Figure 5-4.
Procedure call
connector type
and its variations.



At this point, the reader is not necessarily expected to understand all of this connector type's various dimensions, subdimensions, and values, or all of the variation points of the remaining seven connector types discussed below and depicted in Figure 5-5 through Figure 5-11. One objective of this chapter is to sensitize the reader to the richness and potential complexity of the space of software connectors. The remainder of the chapter will continue to highlight only the representative or particularly interesting facets of each connector type.

5.4.2 Event Connectors

David Rosenblum and Alexander Wolf define an event as the instantaneous effect of the (normal or abnormal) termination of the invocation of an operation on an object, which occurs at that object's location (Rosenblum and Wolf 1997). Event connectors are similar to procedure call connectors in that they affect the flow of control among components, and thus provide coordination services. In this case, the flow is precipitated by an event. Once the event connector learns about the occurrence of an event, it generates messages (that is, event notifications) for all interested parties and yields control to the components for processing those messages. Messages can be generated upon the occurrence of a single event or a specific pattern of events. The contents of an event can be structured to contain more information about the event, such as the time and place of occurrence, and other application-specific data. Event connectors therefore also provide communication services.

Event connectors are also different from procedure calls in that virtual connectors are formed between components interested in the same event topics, and those connectors may appear and disappear dynamically depending on the components' changing interests. Event-based distributed systems rely on the notion of time and ordering of actions.

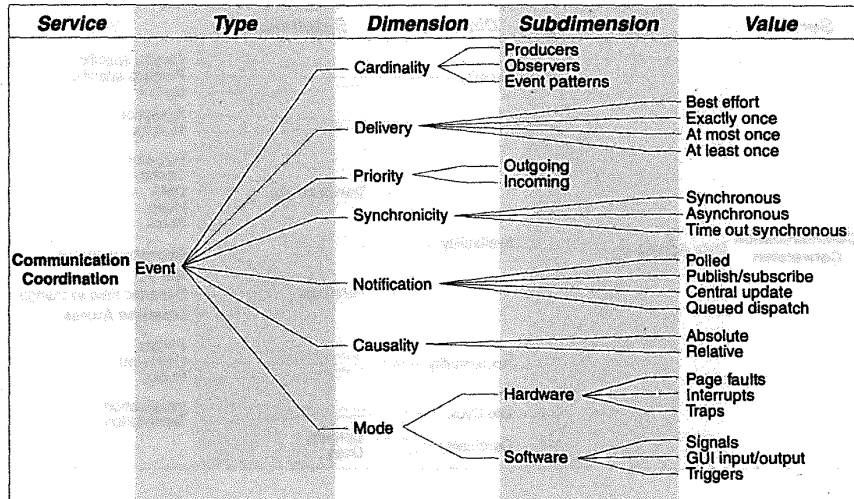


Figure 5-5.
Event connector
type and its
variations.

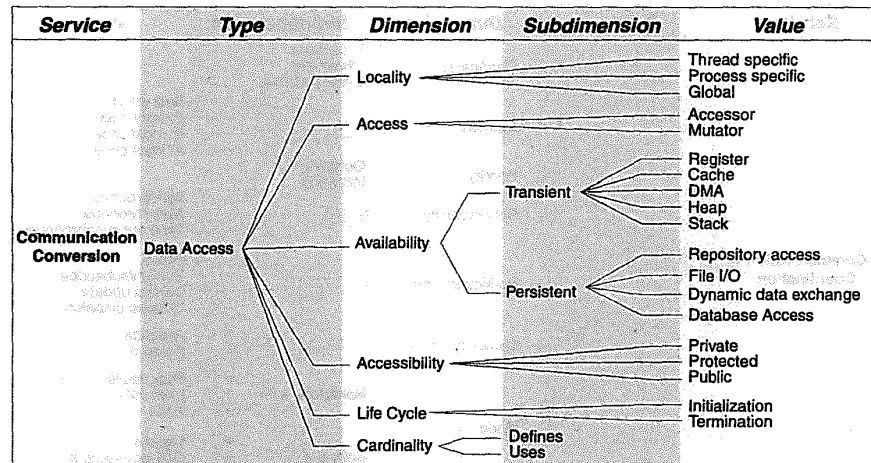
Therefore, dimensions such as causality, atomicity, and synchronicity play a critical role in event connector mechanisms. Event connectors are found in distributed applications that require asynchronous communication. An example is a windowing application (such as X Windows in Unix), where GUI inputs serve as the events that activate the system. Finally, some events, such as interrupts, page faults, and traps, are triggered by hardware and then processed by software. These events may affect global system properties, making it important to capture them in software architectures.

The space of options available to a software engineer for constructing event connectors is shown in Figure 5-5. For example, the cardinality of a multicasting event connector (recall the discussion of the C2 architectural style in Chapter 4) will be a single producer and multiple observer components; ideally the connector will support delivery of data exactly once (whatever is sent by the source component is delivered to the recipient components); its synchronicity may be asynchronous (that is, nonblocking); it could use the publish/subscribe notification mechanism; and so on.

5.4.3 Data Access Connectors

Data access connectors allow components to access data maintained by a data store component. Therefore, they provide communication services. Data access often requires preparation of the data store before and cleanup after access has been completed. In case there is a difference in the format of the required data and the format in which data is stored and provided, data access connectors may perform translation of the information being accessed, that is, conversion. The data can be stored either persistently or temporarily, in which case the data access mechanisms will vary. Examples of persistent data access include query mechanisms, such as SQL for database access, and accessing information in repositories, such as a software component repository. Examples of transient data access include heap and stack memory access and information caching.

Figure 5-6.
Data access
connector type
and its variations.



The space of options available to a software engineer for constructing data access connectors is shown in Figure 5-6. A data access connector could enable global access; allow mutating (that is, changing) the data; it could provide persistent access through file I/O; its cardinality would typically be a single entity that defines the data, but multiple entities that use the data; and so on.

5.4.4 Linkage Connectors

Linkage connectors are used to tie the system components together and hold them in such a state during their operation. Linkage connectors enable the establishment of ducts—the channels for communication and coordination—that are then used by higher-order connectors to enforce interaction semantics. In other words, linkage connectors provide facilitation services.

Once ducts are established, a linkage connector may disappear from the system or remain in place to assist in the system's evolution. Examples of linkage connectors are the links between components and buses in a C2-style architecture (recall Chapter 4) and dependency relationships among software modules described by module interconnection languages (MIL) (DeRemer and Kron 1976).

The space of options available to a software engineer for constructing linkage connectors is shown in Figure 5-7. Compared to the preceding dimensions, this is not a particularly rich connector dimension. The reference to linked components can be implicit, possibly even parameterized and mutable, or explicit. The granularity dimension refers to the size of components and level of detail required to establish a linkage. The subdimensions of granularity (unit, syntactic, and semantic) were directly influenced by Dewayne Perry's foundational study of software interconnection models (Perry 1987):

- **Unit** interconnection specifies only that one component—which can be a module, an object, or a file—depends on another. Examples of unit interconnection are configuration management and system building facilities such as *Make*.

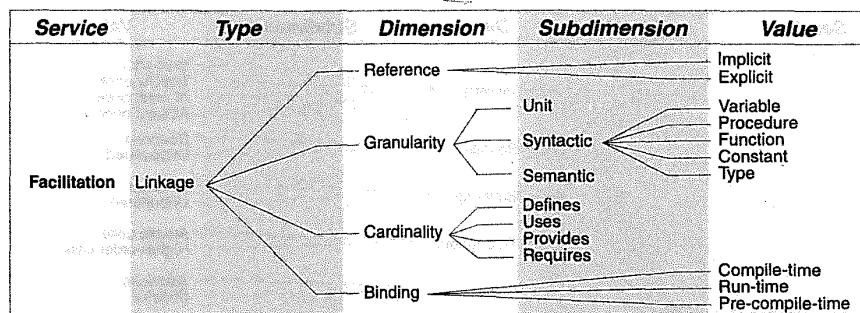


Figure 5-7.
Linkage
connector type
and its variations.

- *Syntactic* interconnection refines this relationship and establishes links between variables, procedures, functions, constants, and types within the linked components. This information can be used in static analysis (For example, locating unreachable code within a module) and smart compilation, where only the changed portions of a system are recompiled.
- *Semantic* interconnection specifies *how* the linked components are supposed to interact. Semantic interconnection ensures that the interaction requirements and constraints are explicitly stated and satisfied. This typically takes the form of an interaction protocol, such as those discussed in the context of architectural modeling and analysis in Chapter 6 and Chapter 8 respectively.

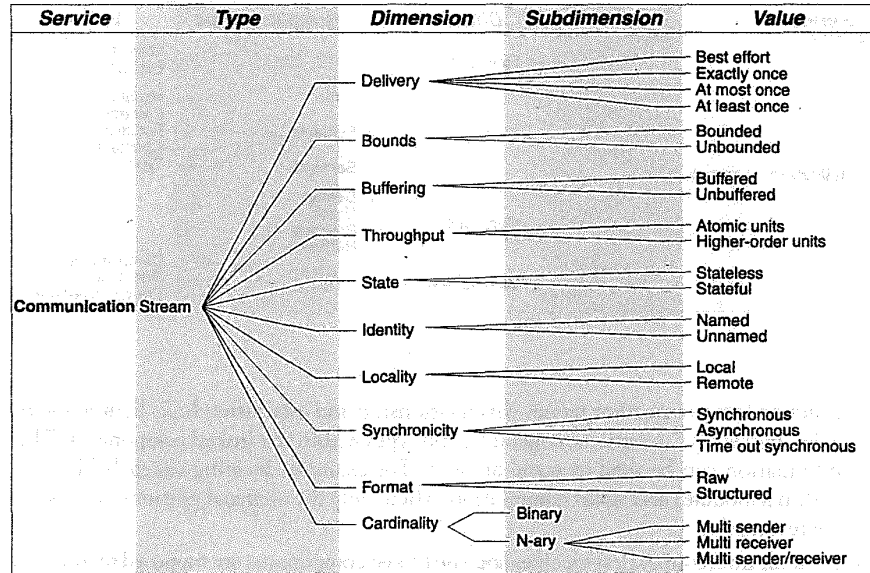
The cardinality of a linkage connector refers to the number of places in which a system resource—such as a component, procedure, or variable—is defined, used, provided, or required. Typically, a resource is defined and/or provided in a single location and used or required from multiple locations. Finally, a linkage connector can establish the binding between components very early (that is, prior to system compilation), early (that is, during compilation), or late (that is, during the system's execution).

5.4.5 Stream Connectors

Streams are used to perform transfers of large amounts of data between autonomous processes. Stream connectors therefore provide communication services in a system. Streams are also used in client-server systems with data transfer protocols to deliver results of computation. Streams can be combined with other connector types, such as data access connectors, to provide composite connectors for performing database and file storage access, and event connectors, to multiplex the delivery of a large number of events. Examples of stream connectors are Unix pipes, TCP/UDP communication sockets, and proprietary client-server protocols.

The space of options available to a software engineer for constructing stream connectors is shown in Figure 5-8. For example, a stream-based connector may be unnamed, as in Unix pipes; it may provide synchronous, remote interaction via structured data; it may guarantee at least one delivery; its cardinality may be binary, that is, a single sender and a single receiver; finally, its state value could be determined by a bounded buffer.

Figure 5-8.
Stream connector
type and its
variations.



5.4.6 Arbitrator Connectors

When components are aware of the presence of other components but cannot make assumptions about their needs and state, arbitrators streamline system operation and resolve any conflicts (thereby providing facilitation services), and redirect the flow of control (providing coordination services). For example, multithreaded systems that require shared memory access use synchronization and concurrency control to guarantee consistency and atomicity of operations. Arbitrators can also provide facilities to negotiate service levels and mediate interactions requiring guarantees for reliability and atomicity. They also provide scheduling and load balancing services. Arbitrators can ensure system trustworthiness by providing crucial support for dependability in the form of reliability, safety, and security.

The space of options available to a software engineer for constructing arbitrator connectors is shown in Figure 5-9. Arbitrator connectors can aid with system fault handling, by determining and trapping component faults before they propagate. They can also ensure the appropriate concurrency semantics among the interacting components. For example, arbitrators can employ mechanisms, such as semaphores or monitors, to control access to the interacting components' resources. Arbitrator connectors may also support transactions and guarantee different levels of security (see Chapter 13). Finally, they can control the interacting components' execution scheduling. A specific example of arbitrator connectors is discussed in Section 5.5.

5.4.7 Adaptor Connectors

Adaptor connectors provide facilities to support interaction between components that have not been designed to interoperate. Adaptors involve matching communication policies and interaction protocols among components, thereby providing conversion services. These

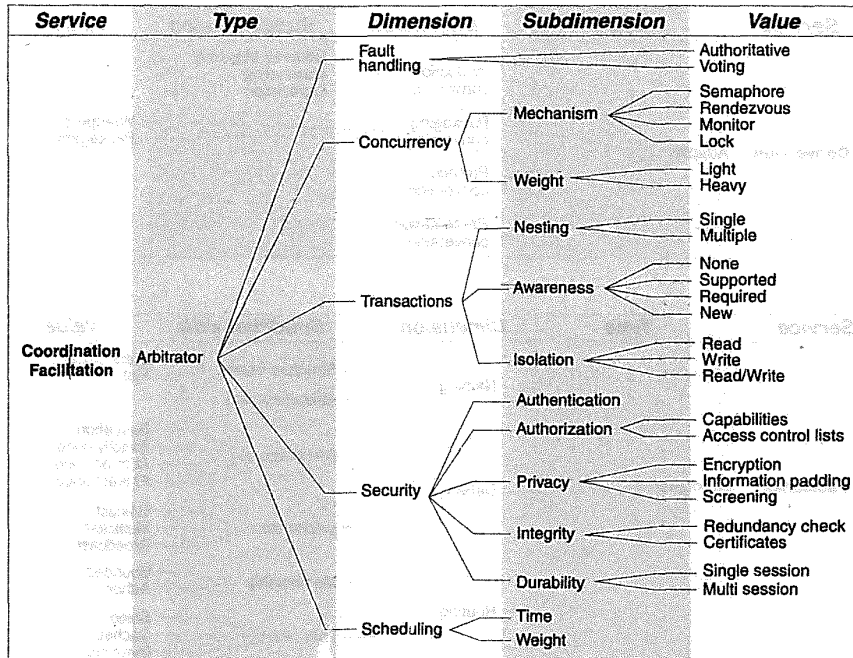


Figure 5-9.
Arbitrator
connector type
and its variations.

connectors are necessary for interoperation of components in heterogeneous environments, such as different programming languages or computing platforms. Conversion can also be performed to optimize component interactions for a given execution environment. For example, a distributed system may rely on remote procedure calls (RPC) for all interactions across process boundaries; if two interacting components are co-located within the same process for a given time period, a remote procedure call may be seamlessly converted to a local procedure call during that time. Adaptors may also employ transformations (for instance, table look-ups) to match required services to the available facilities.

The space of options available to a software engineer for constructing adaptor connectors is shown in Figure 5-10. Examples of adaptors include virtual memory translation; Daniel Yellin and Robert Strom's adaptors (Yellin and Strom 1994), which match incompatible component interaction protocols; virtual function tables used for dynamic dispatch of polymorphic method calls; and Robert DeLine's packagers, which separate a component's internal functionality, referred to as essence, from the manner in which it is accessed (DeLine 2001). XML meta-data interchange (XMI) is a relatively recent approach that supports interchange of models between applications and performs data presentation conversion.

5.4.8 Distributor Connectors

Distributor connectors perform the identification of interaction paths and subsequent routing of communication and coordination information among components along these paths. Therefore, they provide facilitation services. Distributor connectors never exist by

Figure 5-10.
Adaptor
connector type
and its variations.

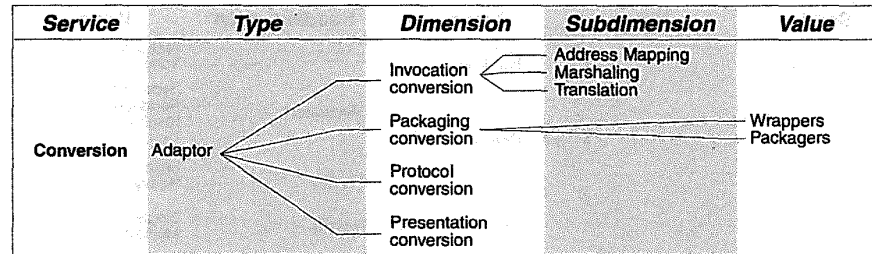
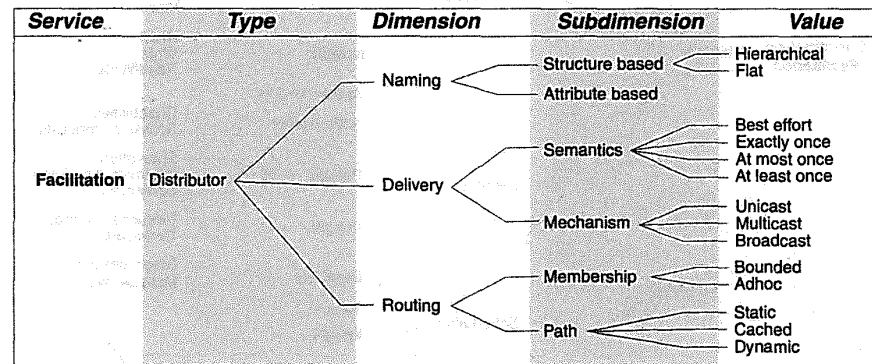


Figure 5-11.
Distributor
connector type
and its variations.



themselves, but provide assistance to other connectors, such as streams or procedure calls. Distributed systems exchange information using distributor connectors to direct the data flow. Distributed systems require identification of component locations and paths to them based on symbolic names. Domain name service (DNS), routing, switching, and many other network services belong to this connector type. Distributors have an important effect on system scalability and survivability.

The space of options available to a software engineer for constructing distributor connectors is shown in Figure 5-11. Issues that are particularly important in distributed systems are resource naming, semantics and mechanisms of data delivery, and characteristics of routing.

5.5 EXAMPLE CONNECTORS

The reader should be familiar at least with the frequently used procedure call and shared data connector types. It likely that the reader has seen or used several of the other connector types, such as distributors or adaptors, and has also been exposed to composite connectors. Furthermore, Chapter 4 introduced a number of connectors in the different Lunar Lander examples.

In this section, we illustrate the connector classification discussed above with four composite distribution connectors that are in wide use today: event-based, grid-based,

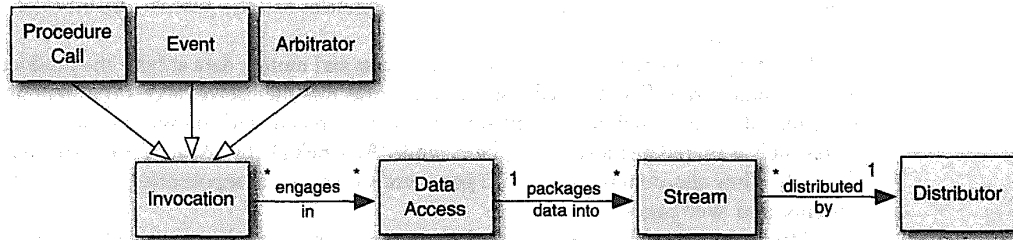


Figure 5-12. Data distribution connectors involve a type of invocation (via arbitrator, event, or procedure call connectors), data access, stream, and distribution. Cardinality is indicated accordingly on the relationship arrows, where * refers to the cardinality of “many.”

client-server-based, and peer-to-peer-based (or P2P-based) connectors. These connectors distribute large amounts of content over a wide area network, such as the Internet. Such connectors are used in disseminating music, movies, scientific data, and so on. Several implementations of these connectors are in wide use by a number of research, industry, and government projects, as well as a large number of individuals around the world.

Distribution connectors can be described as different combinations of six of the connector types defined in the classification of Section 5.4. The exact combination of the six types’ dimensions varies across the different distribution connectors, but in general, each distribution connector performs some form of data access, involving a stream-based reading (and packaging) of data, and distribution of the data to end users. Some connector classes are invoked via procedure calls (for example, client-server-based) while others are invoked via events (for example, event-based), or arbitration (for example, P2P-based). Figure 5-12 illustrates the choices and the relationships.

The remainder of this section describes each of the four data distribution connectors using the connector classification from Section 5.4. The seemingly verbose description of the connectors is necessary: it reflects the number of concerns an architect has to consider when selecting and composing connectors for a specific system, as well as the number of decisions that will be made in the process.

5.5.1 Event-Based Data Distribution Connectors

The event-based data distribution connectors are compositions of four of the connector types from Section 5.4: event, data access, stream, and distributor. These connectors send and receive data through asynchronous notifications called *events*. Events may arrive according to some fixed periodic schedule or at discrete aperiodic intervals. Typically, there are many producers and consumers of data in event-based distribution connectors. For example, science data servers (producers) may alert scientists (consumers) of the availability of new data sets. Event-based distribution connectors often employ an asynchronous best-effort delivery method for data, making no guarantees as to the completion or time of data deliveries. Data delivery events sent through the connectors can be prioritized, tailored to different use cases or deployment environments, and specified by system users. Events can be delivered using prioritized threaded queues, be locally or remotely

managed, or be delivered using user registered preferences and the publish-subscribe delivery mechanism.

Event-based distribution connectors can access and mutate data at both the producer and consumer ends. Event-based distribution connectors themselves access transient and persistent data, both public and private. They communicate with transient session-based stores such as shared memory stores [for example, Apache's Derby (Apache Software Foundation)]. They also communicate with persistent stores such as repositories, databases, file systems, and Web pages.

After data access by event-based distribution connectors, data is typically packaged into streams, both structured and raw, using a best effort approach, with bounded packet size and data buffering. Streams can be identified via named uniform resource identifiers (URIs), and constructed using the asynchronous mode of operation. Streams may be constructed either locally or remotely, depending upon how the data was accessed.

Stream-based data is distributed from data producers to data consumers using a naming registry, via either a hierarchical or flat naming model. Data streams are delivered as events using best effort delivery and any combination of the unicast, broadcast, and multicast delivery mechanisms. Routing of events is typically determined by the specific network layer of the connector's deployment environment.

Example instances of event-based data distribution connectors include the Siena publish-subscribe middleware (Carzaniga, Rosenblum, and Wolf 2001), the Prism-MW middleware (Malek, Mikic-Rakic, and Medvidovic 2005), and its subsequent extension allowing grid resource location and discovery, called GLIDE (Mattmann et al. 2005).

5.5.2 Grid-Based Data Distribution Connectors

The grid-based data distribution connectors are compositions of four of the connector types from Section 5.4: procedure call, data access, stream, and distributor. These connectors move and deliver large amounts of data between software components deployed in the grid environment—a virtual network of shared computing and data resources. Although the grid is further discussed in Chapter 11, this section will outline the architecture of the corresponding connectors. The reader may find it useful to return to this section again after reading Chapter 11.

Grid-based distribution connectors are invoked via a named, synchronous procedure call, often as a Web service call sent using the Simple Object Access Protocol (SOAP) (Mitra 2003). User authentication credentials are provided to the connector for integration with the Grid Security Infrastructure (GSI), a highly secure toolkit based on key pairs and certificate authorities. URLs sent via the connector invocation describe where the data is and where it is to be sent (that is, who the consumers are). Parameters are passed by value using “keyword equals value” semantics, with control messages being logged to the Grid API log layer.

Grid-based distribution connectors access and mutate transient and persistent data, both private and public, as long as there is some type of standard API (for instance, dynamic data exchange over XML, a repository access, or file I/O) for accessing it.

Data access through a grid connector is typically packaged as a stream of bytes (or blocks, configured via a parameter) with exactly-once delivery and (configurable) bounded buffering provided by the network's TCP/IP level. Data can be structured or raw. Buffering

and throughput of data are parallelized via multiple concurrent time-out synchronous TCP/IP streams. Streams are named stateful URLs and URIs, and can exist both locally and remotely, depending upon where the data was accessed. A stream can be delivered to many consumers.

Grid-based distribution connectors distribute data using the grid environment for naming and location, including the Grid Meta-data Catalog Service (MCS) and Replica Location Service (RLS) components, and named URLs and URIs. Naming can be hierarchical, with resources and locations sharing parent-child relationships via the MCS. Alternatively, naming and discovery can be flat, structure-based. Data is delivered via parallelized unicast exactly once using TCP/IP concurrent streams, flooding the underlying network to its saturation, thus making the most efficient use of the available bandwidth possible. Reliability measures allow redelivery of lost packets if necessary, as well as partial data transfers, both sequential and out of order. Routing of data is handled by the underlying TCP/IP network layer.

Example instances of grid-based distribution connectors include the GridFTP software, bundled along with the Globus Grid Toolkit (The Globus Alliance), and the large files transfer protocol, or bbFTP (Farrache 2005).

5.5.3 Client-Server–Based Data Distribution Connectors

The client-server-based distribution connectors allow seamless distribution of data between distributed systems using RPC. They are compositions of four of the connector types from Section 5.4: procedure call, data access, stream, and distributor. These connectors are invoked via a synchronous remote procedure call that appears (to the consumers of data) as if it were a local method call. Keyword (named) parameters are passed to the method call to specify the distribution constraints, with parameters being passed by value or reference, depending upon the underlying remote procedure call implementation. Methods can have return values, which typically must conform to a standard set of primitive data types that are supported on all machines involved in the data distribution. Methods are interactions between one sender and one receiver, typically the producer of the data and its consumer, respectively.

Upon invocation, the client-server-based distribution connector engages in a data access, grabbing and mutating persistent and transient data. After data is accessed, it is packaged into streams that are delivered using exactly-once delivery and bounded packet size managed by the underlying TCP/IP protocol. Data can be raw or structured in the stream. Stream throughput is measured in atomic units (bits per second), and streams are identified using stateful Uniform Resource Names (URNs). Streams can be local or remote, depending on where the data was accessed or packaged.

Data is distributed from client-server-based connectors using a naming registry to locate the requesting consumer of data. Data is delivered through method (or interface) parameters and return values. It is sent exactly once, using the unicast delivery mechanism. Routing is performed by the underlying TCP/IP network using static membership identified via a naming registry.

Example instances of client-server-based data distribution connectors include HTTP/REST (recall Chapter 1), Java RMI, CORBA (recall Chapter 4), FTP, SOAP, and many commercial UDP technologies.

5.5.4 P2P-Based Data Distribution Connectors

The P2P-based data distribution connectors are compositions of four of the connector types from Section 5.4: arbitrator, data access, stream, and distributor. These connectors are unlike the other three categories. Rather than providing some sort of initial procedural or event-based invocation, these connectors typically rely on arbitration as a means of synchronization and invocation. Arbitration involves control flow redirection between distributed resources, or peers, operating in a networked environment. Arbitrators can negotiate protocols, scheduling, and timing issues. Fault handling and parameter passing is handled in an egalitarian fashion via voting and point-to-point or point-to-many communication between peers or groups of peers. P2P-based distribution connectors use rendezvous as a mechanism to achieve concurrency and scheduling. Transaction support is often available, and invocation can be rolled back if necessary.

Upon invocation, peers in a P2P-based distribution connector engage in data access, accessing transient (both process- and thread-specific) and persistent (for instance, repository, file I/O, and so on) data. Since peers can come and go in a P2P-based scenario, for the most part data access is transient in nature: A peer need not stick around for the entire distribution.

Data is accessed and packaged via streams, using a best effort bounded mechanism. Data can be structured or raw, as long as it can be organized into identifiable chunks that can be retransmitted and shared between peers handled by the connector. Streams are named URNs, typically identified using some hashing algorithm (such as SHA-1 or MD5), and are available both locally and remotely depending upon where the data was accessed or packaged.

Data is distributed in the P2P-based distribution connectors by locating other peers that have pieces of data a user wants to obtain or distribute. Location of other peers is based upon attributes such as resource type, SHA-1, and other domain-specific attribute meta-data (for instance, for movie files “production company” might be used or for music “artist name”). Delivery occurs in the form of chunks, sent to and from remote peers using best effort or at least once semantics over unicast and multicast transmission channels. Routing, while influenced by the underlying network layer, is handled by some sort of tracking mechanism, sometimes called trackers or super peers. Tracking mechanisms allow location of chunks of data streams that need to be distributed or obtained in a P2P-based connector.

Example instances of the P2P-based distribution connector class include BitTorrent (Cohen 2003) and JXTA-based data distributors, such as the Peerdata project at NASA's Jet Propulsion Laboratory. P2P systems employing such connectors are discussed in Chapter 11.

5.6 USING THE CONNECTOR FRAMEWORK

This section provides general guidelines for software architects in selecting connectors that meet their needs. The section discusses the issue of connector compatibility: Understanding which connectors, or connector dimensions, are and are not compatible is particularly important in cases where composite connectors are needed.

5.6.1 Selecting Appropriate Connectors

In selecting a software connector that meets the particular needs of a given (sub)system, a software architect must perform at least the following steps. Note that the descriptions of the steps are relatively general, meaning that this process places significant responsibility on the software architects and requires a great deal of familiarity with the technical details of connectors that are at their disposal.

1. Select the specific set of interacting components. Different sets of components, even in the same system, may have different interaction needs. This is why it is important for the architect to focus only on those components for which the desired connector is needed. This and the subsequent steps are repeated for each such component set, that is, for as many distinct connectors as are needed in the system.
2. Determine the interaction services the components need. It is critical to identify the precise characteristics of the components' interaction. This will likely involve studying the components' architectural descriptions, and may also require considering the implementation language and/or framework.
3. Based on the identified interaction services, determine a subset of the eight connector types that comprise the initial candidate set for providing those services. It is sufficient to establish that the connector types chosen in this step may be good candidates for supplying the needed services.
4. Evaluate each connector type from the chosen subset based on the details of the interaction requirements. Study these in light of the connector types' dimensions, subdimensions, and values. Eliminate any connector types whose usage is deemed to result in a suboptimal interaction solution for the specific set of components under consideration.
5. For each of the remaining candidate connector types, set the values for the necessary dimensions and subdimensions as appropriate. Identify the best (most natural) candidate connectors. This will require performing a trade-off analysis among multiple possible solutions. It may also result in selecting a composite connector, which combines features of multiple types, as further discussed below.

The classification of connectors described in Section 5.4 thus serves as the foundation for synthesizing new connectors as well as analyzing the compatibility of connector dimensions. Instantiating dimensions of a single connector type by choosing one or more values forms a simple connector species. On the other hand, using values of dimensions from different connector types leads to a composite (higher-order) connector species. Many real-world scenarios will force an architect to compose such higher-order connectors to satisfy all application requirements.

The reader should note that creating unprecedented, composite connectors is not a trivial task. At the least, this requires developing a deep understanding of the connectors' complementary, orthogonal, and incompatible characteristics. Without such an understanding, the resulting integration process may be misguided and the developed solutions suboptimal, or worse, completely ineffective. Examples of this have been documented in the literature [for example, in the well known case of the interactions among the large components used in the Aesop system (Garlan, Allen, and Ockerbloom 1995)].

Figure 5-13.
Connector
compatibility
matrix.

		Procedure Call					Event					Data Access							
		Parameters	Entry Point	Invocation	Synchronicity	Cardinality	Accessibility	Cardinality	Delivery	Priority	Synchronicity	Notification	Causality	Mode	Locality	Mode	Availability	Accessibility	Cardinality
Procedure Call	Parameters	&	&	!	!									o	o	o			
	Entry Point		&	!	!	o	&		o	&	&				!	o			
	Invocation		!	&	!	o					!	&			o	!			
	Synchronicity		&	&		!			&	o	o	!			!	!			
	Cardinality		&	&	!		o					!							
	Accessibility	&	&	&	&										!	!	o		
Event	Cardinality			!	!	o					!				o				!
	Delivery	o	!	o	o			&		o	!	!		&	&	!			
	Priority												&	&					
	Synchronicity	!	!	o	!	o			!	o		!		!	!	!			
	Notification		o	o	!	&	!		&	&				&		o	x	!	
	Causality		!	o	o										&	&			&
	Mode	!				o		!	o				&	&	&	&		&	
Data Access	Locality		!		!	!	o	!	o		o	o		o					
	Mode		o	o		o	o	o	&	o	o	o	o	&	!				
	Availability		o	!	!	o				o				!	!				
	Accessibility		o	o	o	o	!	o	o		o	o	o	o	&				
	Cardinality		o	o		!	!				!			&					
Stream	Delivery	&			o	o	!								!	&			
	Format		o	o															
	Directionality	&	&	&	o		!									!			
	Cardinality	&	&	!	o	o	!	o											
	State	&	o	o	o	o	o								&				
Linkage	Granularity	!	!	o		!		&	!	!	!	!	&	!	&	!		&	
	Cardinality			!	o	&	!	o	!		!								o
	Reference		!	o	!	&	!							!	&	!			
	Binding	&	&	o	!	o	!							!	&				
Arbitrator	Fault handling	&	&	&	&	&	o	&	!	&	&	&	&	&	&				
	Concurrency	&	&	&	!	&	&	&	!		!	&	&	&	!	o			
	Transactions	&	!	&	!	&		&	!	!	!	&		&	&				
	Logging	&	&	&	&	o	o	&		&	&			&	&	&	&		
	Security	&	&	&	!	&	&	&	!	!	!	&	&	&	&	&	&		
	Scheduling	&	&	&	!	&		&	!	o	&	&	&	&	&	&			
	Pooling	&	&	o	!	&	o	&		&	&	&	&		&				
Adaptor	Invocation	&	&	&	&	o	x	!	&	o	&	&		&	&	&	&		&
	Presentation	&	&	&	o	o			&	o	&			&	&	&	&		
	Packaging	&	&	&	!		x				&			&					
	Protocol		o	o	o		&				&	!			&	&			
Distrib.	Naming	&	&	!	o			&				&	&	o				!	&
	Delivery	&	!	&	&	!	!	&	!	&	!	&	&	&	!	&	o	!	
	Routing	&	&	&				&	&		&	&	&	&	&	&	&		&

1. *La prima* è la *definizione* di *funzione* (o *funzione* di *una* *variabile* o *funzione* di *due* *variabili* o *funzione* di *tre* *variabili* o *funzione* di *quattro* *variabili* o *funzione* di *cinque* *variabili* o *funzione* di *sei* *variabili* o *funzione* di *sette* *variabili* o *funzione* di *otto* *variabili* o *funzione* di *nove* *variabili* o *funzione* di *dieci* *variabili* o *funzione* di *undici* *variabili* o *funzione* di *doce* *variabili* o *funzione* di *tre* *decine* *di* *variabili* o *funzione* di *quattro* *decine* *di* *variabili* o *funzione* di *cinque* *decine* *di* *variabili* o *funzione* di *sei* *decine* *di* *variabili* o *funzione* di *sette* *decine* *di* *variabili* o *funzione* di *otto* *decine* *di* *variabili* o *funzione* di *nove* *decine* *di* *variabili* o *funzione* di *dieci* *decine* *di* *variabili* o *funzione* di *undici* *decine* *di* *variabili* o *funzione* di *doce* *decine* *di* *variabili* o *funzione* di *tre* *centine* *di* *variabili* o *funzione* di *quattro* *centine* *di* *variabili* o *funzione* di *cinque* *centine* *di* *variabili* o *funzione* di *sei* *centine* *di* *variabili* o *funzione* di *sette* *centine* *di* *variabili* o *funzione* di *otto* *centine* *di* *variabili* o *funzione* di *nove* *centine* *di* *variabili* o *funzione* di *dieci* *centine* *di* *variabili* o *funzione* di *undici* *centine* *di* *variabili* o *funzione* di *doce* *centine* *di* *variabili* o *funzione* di *tre* *migliaia* *di* *variabili* o *funzione* di *quattro* *migliaia* *di* *variabili* o *funzione* di *cinque* *migliaia* *di* *variabili* o *funzione* di *sei* *migliaia* *di* *variabili* o *funzione* di *sette* *migliaia* *di* *variabili* o *funzione* di *otto* *migliaia* *di* *variabili* o *funzione* di *nove* *migliaia* *di* *variabili* o *funzione* di *dieci* *migliaia* *di* *variabili* o *funzione* di *undici* *migliaia* *di* *variabili* o *funzione* di *doce* *migliaia* *di* *variabili* o *funzione* di *tre* *decine* *di* *migliaia* *di* *variabili* o *funzione* di *quattro* *decine* *di* *migliaia* *di* *variabili* o *funzione* di *cinque* *decine* *di* *migliaia* *di* *variabili* o *funzione* di *sei* *decine* *di* *migliaia* *di* *variabili* o *funzione* di *sette* *decine* *di* *migliaia* *di* *variabili* o *funzione* di *otto* *decine* *di* *migliaia* *di* *variabili* o *funzione* di *nove* *decine* *di* *migliaia* *di* *variabili* o *funzione* di *dieci* *decine* *di* *migliaia* *di* *variabili* o *funzione* di *undici* *decine* *di* *migliaia* *di* *variabili* o *funzione* di *doce* *decine* *di* *migliaia* *di* *variabili* o *funzione* di *tre* *centine* *di* *migliaia* *di* *variabili* o *funzione* di *quattro* *centine* *di* *migliaia* *di* *variabili* o *funzione* di *cinque* *centine* *di* *migliaia* *di* *variabili* o *funzione* di *sei* *centine* *di* *migliaia* *di* *variabili* o *funzione* di *sette* *centine* *di* *migliaia* *di* *variabili* o *funzione* di *otto* *centine* *di* *migliaia* *di* *variabili* o *funzione* di *nove* *centine* *di* *migliaia* *di* *variabili* o *funzione* di *dieci* *centine* *di* *migliaia* *di* *variabili* o *funzione* di *undici* *centine* *di* *migliaia* *di* *variabili* o *funzione* di *doce* *centine* *di* *migliaia* *di* *variabili* o *funzione* di *tre* *migliaia* *di* *centine* *di* *variabili* o *funzione* di *quattro* *migliaia* *di* *centine* *di* *variabili* o *funzione* di *cinque* *migliaia* *di* *centine* *di* *variabili* o *funzione* di *sei* *migliaia* *di* *centine* *di* *variabili* o *funzione* di *sette* *migliaia* *di* *centine* *di* *variabili* o *funzione* di *otto* *migliaia* *di* *centine* *di* *variabili* o *funzione* di *nove* *migliaia* *di* *centine* *di* *variabili* o *funzione* di *dieci* *migliaia* *di* *centine* *di* *variabili* o *funzione* di *undici* *migliaia* *di* *centine* *di* *variabili* o *funzione* di *doce* *migliaia* *di* *centine* *di* *variabili* o *funzione* di *tre* *decine* *di* *migliaia* *di* *centine* *di* *variabili* o *funzione* di *quattro* *decine* *di* *migliaia* *di* *centine* *di* *variabili* o *funzione* di *cinque* *decine* *di* *migliaia* *di* *centine* *di* *variabili* o *funzione* di *sei* *decine* *di* *migliaia* *di* *centine* *di* *variabili* o *funzione* di *sette* *decine* *di* *migliaia* *di* *centine* *di* *variabili* o *funzione* di *otto* *decine* *di* *migliaia* *di* *centine* *di* *variabili* o *funzione* di *nove* *decine* *di* *migliaia* *di* *centine* *di* *variabili* o *funzione* di *dieci* *decine* *di* *migliaia* *di* *centine* *di* *variabili* o *funzione* di *undici* *decine* *di* *migliaia* *di* *centine* *di* *variabili* o *funzione* di *doce* *decine* *di* *migliaia* *di* *centine* *di* *variabili* o *funzione* di *tre* *centine* *di* *migliaia* *di* *centine* *di* *variabili* o *funzione* di *quattro* *centine* *di* *migliaia* *di* *centine* *di* *variabili* o *funzione* di *cinque* *cent*

In the absence of a set of concrete, well-understood, and widely adopted rules for defining composite connectors, it becomes difficult to guide this process. Recent research by, for example, Bridget Spitznagel and David Garlan (Spitznagel and Garlan 2003) suggests some specific formalisms and strategies that can be adopted to address connector composition. However, ultimately a comprehensive classification of connectors, such as the one presented in this chapter, is necessary (though not sufficient) to create a general set of guidelines that specify the conditions under which two or more connectors can be composed. We discuss this issue in more depth next.

5.6.2 Detecting Mismatches

The connector taxonomy introduced in Section 5.4 can be leveraged to identify potential mismatches between incompatible connector dimensions, and to avoid such combinations. Figure 5-13 shows a compatibility matrix for the connector dimensions identified in the classification from Figure 5-4 through Figure 5-11. The matrix provides guidance regarding which combinations of dimensions are necessary and which should be avoided; the empty cells in the matrix denote combinations that are possible, but not required. The sparseness of the matrix suggests that the allowed design space of connectors is very large based on their current understanding. At the same time, some pitfalls are known to exist. Four kinds of rules for combining connector dimensions can be identified: requires, cautions, restricts, and prohibits.

The *requires* rule indicates that the choice of one dimension in one connector species mandates that another dimension be selected in another connector species; it also allows all possible combinations of the values of the given dimension and its required co-dimension. For example, if a distributor and an adaptor connector are composed, distributor's delivery requires that the adaptor support presentation conversion, that is, that it allow data to be marshaled for transport across address spaces.

Requires is a chaining rule and is used as the starting point for constructing the base connector species. For example, an event connector that requires delivery semantics also needs a notification dimension, which in turn requires cardinality, synchronicity, and mode. This chaining rule results in identification of dimensions that are mandatory in all species of a connector type, and those that are optional. In Figure 5-13, the mandatory dimensions are in **boldface**, whereas the optional dimensions are in regular font.

The *cautions* rule indicates that certain combinations of values for two connector dimensions that are required to be used in tandem, while valid, may result in an unstable or unreliable connector. For example, a component being invoked implicitly should not have multiple entry points since an implicit invocation mechanism cannot choose among the entry points. Another example is the relationship between arbitrator connector's concurrency dimension and data access connector's locality dimension: the granularity level at which concurrency is supported by the former (heavy-weight versus light-weight) should match the level at which locality is supported by the latter (process-specific versus thread-specific).

The *restricts* rule is used to indicate that the two dimensions are not required to be used together at all times, and that there are certain combinations of their values that are invalid. For example, thread-specific data access cannot use heavy-weight concurrency (see the intersection of data access-locality with arbitrator-concurrency). Likewise, passing

parameters by name when those parameters have only transient availability can cause the parameters' values to disappear, be changed, or be affected by some read-modify-write condition; the problem is exacerbated when these parameters are passed by reference and the procedure has no place to store their new values or to access the old ones (see the intersection of procedure call-parameters with data access-availability).

Finally, the *prohibits* rule is used to exclude any combination of two dimensions from being used and indicates total incompatibility of the dimensions. For example, stream delivery cannot be built on transactional atomicity (see the intersection of stream-delivery with arbitrator-transactions). As can be seen in Figure 5-13, there are relatively few instances of the prohibits relationship. This is a positive indicator, as it shows that most connector combinations are legal at least under some circumstances. At the same time, the large numbers of restricts and cautions relationships in Figure 5-13 also indicate that software architects must be careful when composing connectors.

While the above discussion was restricted to the binary combinations of connector dimensions, the compatibility relations between dimensions are transitive. In other words, the compatibility rules can be successively applied to determine the *n*-ary compatibility between dimensions. The binary relations outlined in this section would serve as a necessary starting point for analyzing the *n*-ary relations.

Explicit, targeted study of software connectors is a relatively recent occurrence. There still are many details that need to be uncovered and understandings improved. Figure 5-13 is indicative of this: The sparser sections of the compatibility matrix argue for the need for greater understanding of certain combinations of connector types (such as data access and stream). It is reasonable to expect that, as our understanding of connectors improves over time, the compatibility matrix, as well as the rationale behind it, will also evolve.

5.7 END MATTER

Every software system employs connectors, frequently many of them. In complex distributed systems in particular, connectors can be the key determinants of whether the desired system properties will be met. This observation is often made, yet connectors are not always given first-class status in software systems, and have certainly not been studied to the same extent as components.

This chapter has presented a detailed study of software connectors. At the most basic level, each connector establishes a conduit (which can be thought of, and referred to, as a virtual channel or duct) for two or more components to interact by exchanging control and data in a software system. The actual manner in which data and/or control are exchanged, however, and the specific characteristics of that exchange, can vary widely. This is what makes the space of software connectors quite large and complex.

As this chapter has argued, the space of connectors can be better understood by considering the role (or roles) played by a given connector and the type (or types) of interaction supported by the connector. At the same time, the large number of variation points for each connector type (the dimensions, subdimensions, and values introduced in Section 5.4) require that the (in)compatibilities among connectors be considered carefully. While this area clearly requires further study, the existing body of knowledge, if applied judiciously, can be a great aid to software architects and developers.

The following several chapters will use this as well as the preceding chapters as the foundation for introducing the reader to a number of key activities in architecture-driven software development. Chapter 6 focuses on the principles of architecture modeling and presents the details of a number of software architecture description languages. Chapter 7 introduces the reader to the approaches for visualizing software architectures. Chapter 8 provides an overview of architectural analysis. Chapter 9 discusses the challenges and solutions for architecture-based system implementation. Finally, Chapter 10 focuses on the postimplementation concerns of system deployment and run-time mobility.

The Business Case

The engineering of complex software systems is difficult and expensive. In particular, composing large components, many of which may not have been developed to “play” together, can cause bloated systems with unacceptable performance, project schedule slippages, and budget overruns. Examples of software projects that fit this general description abound.

If connectors in such scenarios are not considered explicitly, then every pair of components that needs to be integrated may need to be dealt with separately. While past experience may prove useful to architects and developers of such systems, the usually applied techniques (for instance, building wrappers and “glue code”) will need to be reapplied in each individual case. Furthermore, this integration code will often be tightly coupled with one or more of the interacting components, rendering its reuse difficult or impossible. Perhaps most importantly, by constantly going back to the same integration toolbox, software engineers are likely to miss more appropriate, and perhaps less costly, component integration and interaction solutions.

Having an intimate understanding of connectors, and treating them separately from the components, can potentially address all of the above concerns. One useful property of connectors, which is not shared by many components, is that they are largely application-independent. This means that there is actually a finite (although very large) number of component interaction and integration challenges. Many such challenges may be similar across like products (for instance, in a product family) or within a particular application domain.

A given software development organization could thus develop and grow a large, reusable arsenal of explicit connectors, which would, in turn, help to curb the costs of future projects.

5.8 REVIEW QUESTIONS

1. Define a software connector.
2. How are connectors different from components?
3. What is a duct? What purpose does it serve?
4. What is the difference between the transfer of control and the transfer of data?
5. Name and describe the four possible roles played by a connector.
6. How is facilitation different from conversion?
7. Name and briefly describe the eight different connector types.
8. Why is connector composition challenging?
9. Are there connector types that can always be composed?
10. Are there types that can never be composed?
11. What are data-intensive connectors?
12. What characteristics do all data-intensive connectors share?

5.9 EXERCISES

1. For each connector type discussed in this chapter, try to identify a specific connector species that belongs to that type.
2. For the connectors identified in the previous question, enumerate the dimension and subdimension values that each of the connector species takes.
3. Analyze the connectors used in the different Lunar Lander examples in Chapter 4 to determine the connector types to which they belong. Specify the values they take for different dimensions and subdimensions.
4. Consider the Aesop system developed by Garlan et al. and described in (Garlan, Allen, and Ockerbloom 1995). Develop a detailed strategy for integrating Aesop's four major components by explicitly treating its connectors. Would your strategy have helped Aesop's developers to deal with the architectural mismatches among the components more effectively than their adopted strategy?
5. Select a software system with which you are intimately familiar. Isolate one of the connectors in the system (for example, a procedure call connector). Replace that connector with a connector of a different type (for example, an event connector). What were the required steps in doing so? Discuss the lessons learned.
6. Select a software system with which you are intimately familiar and which uses mostly procedure call connectors. Replace all procedure call connectors in the application with explicit, named connectors, thereby completely decoupling the interacting components. Discuss the relative merits of the two applications' architectures.
7. Repeat the above exercise by changing one or more connectors that support synchronous interaction between components with connectors that support asynchronous interaction. What challenges did you face in accomplishing this task? Does the modified application exhibit the same behavior as the original application? Why or why not?
8. Several open-source systems, available from Source Forge.net and similar online repositories, are accompanied by thorough design documentation. Select at least two open-source systems and study their architectures. Identify their connectors. Classify the connectors according to their types. Can you spot any trends in the systems' connector usage?
9. Mehta et al. (Mehta, Medvidović, and Phadke 2000) have argued that the Linux Process Scheduler subsystem identified in the study by Bowman et al. (Bowman, Holt, and Brewster 1999) is, in fact, a connector. Investigate the implications of this decision. In order to do so, you may want to consider treating Process Scheduler as a component, or as a subarchitecture comprising a composition of finer-grained components and connectors. Were Mehta et al. correct in their decision? Does their decision fundamentally alter the architecture of Linux, or one's understanding of it?
10. This chapter has suggested that, architecturally, the different classes of data-intensive connectors are similar to a certain extent. Select two such widely used connectors from different categories—for instance, client-server and peer-to-peer. Based on their architectures discussed in Section 5.5, outline a strategy for converting one connector into the other.
11. Study the Question 10's two connectors' architectures and implementations in more detail. Was your conversion strategy viable? If so, proceed with the conversion. If not, discuss the reasons why, adjust your strategy as appropriate, and proceed with the conversion.

5.10 FURTHER READING

Software connectors have been an important facet in developing large, complex software intensive systems for a long time. While they may not have been the primary focus of study, and may not have been identified explicitly, connectors figured prominently in the operating systems, programming languages, distributed systems, computer

networks, and middleware literature [for instance (Reiss 1990), (Colouris, Dollimore, and Kindberg 1994), (Yellin and Strom 1994), (Orfali, Harkey, and Edwards 1996)].

The explicit focus on software connectors is more recent and has emerged from the body of work dealing with software architecture. Perry (Perry 1987) provided the

needed foundation in his study of software interconnection models. Perry and Wolf (Perry and Wolf 1992) were the first to suggest connectors as first-class entities in a software architecture, and Shaw and Garlan (Shaw 1994; Shaw, DeLine, and Zelesnik 1996; Shaw and Garlan 1996) soon followed suit. Allen, Garlan, and Ockerbloom (Garlan, Allen, and Ockerbloom 1995) demonstrated that explicit connectors can provide a substantial aid in system specification, and can carry significant, inherent analytical power.

Several researchers have tried to classify connectors as an aid to software architects in making the most appropriate design choices. An early such effort by Hirsch, Uchitel, and Yankelevich (Hirsch, Uchitel, and Yankelevich 1999) was limited in scope, but it directly inspired the more detailed study by Mehta, Medvidović, and Phadke

(Mehta, Medvidović, and Phadke 2000), which has been referenced extensively in this chapter. Bálek and Plášil (Bálek and Plášil 2001) have proposed a formal connector model for dealing with software deployment (recall Chapter 3). In particular, they posit that the basic set of dynamic services that a connector should provide to aid deployment maps directly to the key service categories in Mehta et al.'s taxonomy: control and data transfer, interface adaptation and data conversion, access coordination and synchronization, communication intercepting, and dynamic component linking. Bálek and Plášil additionally demonstrate that their connectors can be composed to provide more advanced interaction services, in a similar vein to Spitznagel and Garlan's later work (Spitznagel and Garlan 2003).

Modeling

As we have stated, every software system has an architecture, whether it is a “good” architecture or not. We have previously defined architecture as the set of principal design decisions about a system. Once design decisions have been made, they can be recorded. Design decisions are captured in *models*; the process of creating models is called *modeling*. From Chapter 3 we have:

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

Models can capture architectural design decisions with varying levels of rigor and formality. They enable users to communicate about, visualize, evaluate, and evolve an architecture. Without models, architectures are inscrutable.

Throughout the chapter, we also discuss how architecture modeling *notations* are used to capture design decisions:

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

Available architecture modeling notations range from the rich and ambiguous (such as natural language) to the semantically narrow and highly formal (such as the Rapide architecture description language). While some models conform to a single notation, a model may also use a mix of different notations. For example, a single model may use the Unified Modeling Language (UML) class diagram notation to describe the classes in a system, but annotate them with natural language descriptions of their functions.

This chapter introduces a broad variety of modeling concepts that are captured in architectural models, from basic architectural elements (for instance, components and connectors) to more complex properties of systems such as behavioral models. It then covers some of the many available options for capturing different aspects of an

architecture, from the semantically weak (such as PowerPoint modeling) to those with formal semantics. Finally, it discusses the strengths and weaknesses of some of these notations as applied to some of the systems we've discussed already, such as Lunar Lander and the World Wide Web.

Outline of Chapter 6

6 Modeling	6.4 Evaluating Modeling Techniques
6.1 Modeling Concepts	6.5 Specific Modeling Techniques
6.1.1 Stakeholder-Driven Modeling	6.5.1 Generic Techniques
6.1.2 Basic Architectural Concepts	6.5.2 Early Architecture Description Languages
6.1.3 Elements of the Architectural Style	6.5.3 Domain- and Style-Specific ADLs
6.1.4 Static and Dynamic Aspects	6.5.4 Extensible ADLs
6.1.5 Functional and Non-Functional Aspects	6.6 When Systems Become Too Complex to Model
6.2 Ambiguity, Accuracy, and Precision	6.7 End Matter
6.2.1 Ambiguity	6.8 Review Questions
6.2.2 Accuracy and Precision	6.9 Exercises
6.3 Complex Modeling: Mixed Content and Multiple Views	6.10 Further Reading
6.3.1 Views and Viewpoints	
6.3.2 Consistency among Views	

6.1 MODELING CONCEPTS

In this chapter, we discuss a broad spectrum of *kinds* of things that can be modeled in an architecture, and then discuss how various notations can be selected and used to facilitate this modeling. Specifically, for each concept we identify, we will identify what it takes to effectively model that concept—which notational features, and so on.

6.1.1 Stakeholder-Driven Modeling

One of the most critical decisions that architects and others stakeholders will make in developing an architecture is to choose:

1. The architectural decisions and concepts that should be modeled.
2. The level of detail.
3. The amount of rigor or formality.

These decisions should be based on *costs* and *benefits*. Architects should balance the benefits of having certain models in certain forms or notations with the costs of creating and maintaining those models. Thus, the choice of what to model, and at what level of detail, will be *stakeholder driven*. A good rule of thumb is that the most important or critical aspects of a system should be the ones that are modeled in the greatest detail with the highest degrees of rigor/formality.

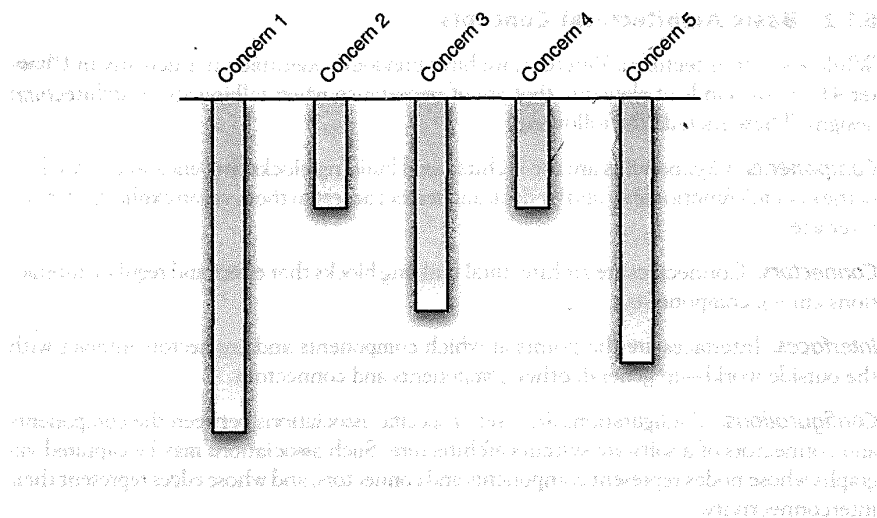


Figure 6-1.
Modeling
different concerns
with different
depths (from
Maier and
Rechtin).

Figure 6-1, borrowed from Mark Maier and Eberhardt Rechtin's book, *The Art of Systems Architecting* (Maier and Rechtin 2000), graphically depicts this concept. It shows five concerns about the system identified by stakeholders. In this particular case, Concern 1 is of great importance, and will be considered and modeled deeply—in great detail. Concerns 2 and 4 are less important and will be modeled shallowly. Because the concerns and their relative levels of importance will vary from project to project, each project will have somewhat different modeling needs.

Modeling is an activity, and as such it is often governed by a process. It will undoubtedly be part of the larger process of architecture-centric software development that we have been discussing in this book. However, modeling itself is a subprocess, and the process of modeling can vary widely from project to project.

The basic activities behind stakeholder-driven modeling are to:

1. Identify relevant aspects of the software to model.
2. Roughly categorize them in terms of importance.
3. Identify the goals of modeling for each aspect (communication, bug finding, quality analysis, generation of other artifacts, and so on).
4. Select modeling notations that will model the selected aspects at appropriate levels of depth to achieve the modeling goals.
5. Create the models.
6. Use the models in a manner consistent with the modeling goals.

Although the steps outlined above are in rough chronological order, they can be incorporated into an iterative process. It is almost never clear from the outset of a project what all the important aspects of a software system are, what the goals of modeling are, and whether those goals are achievable using available notations, technology, time, and money. As such, these estimations must be reevaluated and refined multiple times through a system's development and the models and modeling efforts adjusted accordingly.

6.1.2 Basic Architectural Concepts

While each architecture is different, we have previously identified (particularly in Chapter 4) certain kinds of elements that are of importance when talking about architectural designs. These include the following.

Components. Components are the architectural building blocks that encapsulate a subset of the system's functionality and/or data, and restrict access to them via an explicitly defined interface.

Connectors. Connectors are architectural building blocks that effect and regulate interactions among components.

Interfaces. Interfaces are the points at which components and connectors interact with the outside world—in general, other components and connectors.

Configurations. Configurations are a set of specific associations between the components and connectors of a software system's architecture. Such associations may be captured via graphs whose nodes represent components and connectors, and whose edges represent their interconnectivity.

Rationale. Rationale is the information that explains why particular architectural decisions were made, and what purpose various elements serve.

These concepts form a starting point for architectural modeling. At the most basic level, modeling these concepts requires a notation that can express a graph of components and connectors, preferably with well-defined connection points (interfaces). These languages can be relatively simple—basic box and arrow diagrams or lists with appropriate internal references will suffice. Rationale is somewhat different because it is more amorphous. Rationale is primarily used for communicating information and intent among stakeholders, and is generally not represented explicitly in the actual, built software system. As such, languages for expressing rationale need to be more expressive and less constrained—rationale is most often expressed using natural language for this reason.

Modeling at only the most basic level (that is, enumerating the existence and interconnections of the various components and connectors) provides a small amount of value, but these models will not suffice for most complex projects. Rather, these models must be extended and annotated with a host of other concepts: How are the functions partitioned among the components? What are the nature and types of the interfaces? What does it mean for a component and a connector to be linked? How do all these properties of a system change over time? These questions are at the heart of architectural modeling.

Depending on the nature of the system being developed and the domain, it may or may not be straightforward to represent these basic elements. For example, in a desktop application such as a word processor or spreadsheet, the software components and connectors will be relatively static and few in number. Even with a very complex desktop application containing, say, 500 to 1000 components and connectors, it is feasible to enumerate and describe each of these elements as well as the interconnections among them.

Applications that are large, dynamic, and distributed may be harder to model. For example, it is basically impossible to enumerate all the software components that are part of the World Wide Web or their configuration: There are too many of them and they are

constantly coming and going. For these applications, it is probably more reasonable to model only parts of the system, or specific configurations that represent expected use cases. Alternatively, it may be more effective to model the architectural style that governs what kinds of elements may be included in the architecture and how they may be configured.

6.1.3 Elements of the Architectural Style

Recall that an architectural style is a collection of architectural design decisions that are applicable in a given development context, constrain architectural design decisions that are specific to a particular system within that context, and elicit beneficial qualities in each resulting system. In addition to modeling basic architectural elements, it is often useful to model the style that governs how these elements have been (and may be) used.

Note that architectural styles, like architectures, are made up of design decisions. These design decisions can also be modeled. Explicitly modeling the architectural style can be helpful for a number of reasons. It reduces confusion about what is and is not allowed in the architecture. It can help to reduce architectural drift and erosion. It makes it easier to distinguish whether a specific design decision in an architecture was made to conform to a stylistic constraint or for some other reason. It can help to guide the evolution of the architecture. It can be more feasible and useful than structural modeling (component and connector graphs) for large or dynamic systems. Because styles are generally applicable to many projects, style models can be reused from project to project. Styles represent a single place to capture cross-cutting concerns and rationale for an architecture.

The kinds of design decisions found in an architectural style are generally more abstract or general than those found in an architecture. Some kinds of design decisions that might be captured in a style model include:

Specific Elements. A style may prescribe that particular components, connectors, or interfaces be included in architectures or used in specific situations. This can be facilitated by a modeling approach that supports templates or base models that are then elaborated.

Component, Connector, and Interface Types. Specific kinds of elements may be permitted, required, or prohibited in the architecture. Many modeling approaches are accompanied by a type system, although they often have different semantics.

Interaction Constraints. Constraints on interactions between components and connectors may exist. These constraints can take many forms. They may be temporal (“calling components must call `init()` before any other method”). They may be topological (“only components in the *client* layer are allowed to invoke components in the *server* layer”). They may specify particular interaction protocols, either by name (FTP, HTTP) or specification (formal protocol specifications in a language such as CSP or sequence charts). Modeling approaches that support constraints generally leverage some sort of logic—first-order logic, temporal logic, and so on.

Behavioral Constraints. Constraints on the behavior of architectural elements, or kinds of elements, may be included. Constraints can run the gamut from simple rules to full-blown complete behavioral specifications for components expressed in a notation such as finite state automata. Here again, modeling approaches that support behavioral constraints use either logic or other formal models (such as automata diagrams).

Concurrency Constraints. Constraints on which elements perform their functions concurrently and how they synchronize access to shared resources can also be included in an architectural style. Approaches that support concurrency modeling often employ formal behavioral models of various architectural elements; many also include temporal modeling techniques, such as sequence charts and statecharts.

6.1.4 Static and Dynamic Aspects

Architectural design decisions may address both static and dynamic aspects of a system. Static aspects of a system are those that do not involve the system's behavior while it is executing. Dynamic aspects of a system involve the system's runtime behavior.

Static aspects are generally easier to model simply because they do not involve changes over time. Typical models of static system aspects might include component/connector topologies, assignments of components and connectors to processing elements or hosts, host and network configurations, or mappings of architectural elements to code or binary artifacts.

Dynamic aspects of a system can be harder to model because they must deal with how a system behaves or changes over time. Typical models of dynamic aspects may be behavioral models (describing the behavior of a component or connector over time), interaction models (describing the interactions between a set of components and connectors over time), or data flow models (describing how data flows through an architecture over time).

The static/dynamic distinction is often not a clear line. For example, a system's structure may be relatively stable, but may change occasionally due to component failure, the use of flexible connectors, or architectural dynamism (see Chapter 14). In these cases, models that capture both static and dynamic system aspects may be employed: For example, a (static) base topology may be accompanied by a set of transitions that describe a limited set of changes that may occur to that topology during execution.

There is an important distinction between modeling static and dynamic *aspects* of a system and using static or dynamic *models*. The former refer to properties of the system being modeled. The latter refer to changes to the models themselves. A *model of a dynamic aspect* of a system describes how the *system* (often, its internal state) changes as it executes. A *dynamic model* actually changes itself.

Dynamic models are not necessary to capture dynamic aspects of a system. Consider a statechart. A statechart can capture the behavior of a component over time (a dynamic aspect), but the statechart itself does not generally change (for example, acquire new nodes or transitions) as the component executes. If the statechart were connected to a running system such that the current state of the system was always highlighted in the statechart, then the statechart would become a dynamic model. Such a statechart would be useful for visualizing the behavior of an application, perhaps for debugging purposes.

Supporting dynamic models is more difficult than supporting static models. Once developed, static models can be incorporated into a system in "read-only" mode—they are static resources that can be used as a basis for implementation, comparison, and analysis. Dynamic models must be incorporated in "read-write" mode—the system's execution can change the model itself. This requires tool support to keep the model and the system synchronized and consistent. For these mechanisms to operate automatically, the

model must be stored in a machine-readable and writable form, and the model must be appropriately mapped to the implemented system. Visualizations of the model must also be suitably dynamic, reflecting changes to the model on-the-fly if possible.

6.1.5 Functional and Non-Functional Aspects

Architecture can capture both functional and non-functional aspects of a system. Functional aspects relate to *what* a system does. Non-functional aspects relate to *how* a system performs its functions. A good rule of thumb for thinking about this distinction is that functional aspects of a system can be described using declarative, subject-verb sentences: *The system prints medical records*. Non-functional aspects of a system can be described by adding adverbs to these sentences: *The system prints medical records **quickly and confidentially***. An extensive survey of non-functional properties from an architectural perspective is contained in Chapter 12.

Because functional aspects are generally more concrete, they are easier to model and can often be modeled rigorously and formally. Typical functional models of a system might capture the services that are provided by different components and connectors and the interconnections that achieve the overall system functions. They may capture the behavior of components, connectors, or subsystems, describing what functions those elements perform. Functional aspects of a system can be static or dynamic.

Most modeling notations focus on capturing functional aspects of systems. Notations and approaches differ in the aspects of a system that can be described, and how. Any selected notation or modeling approach must employ a sufficient number of concepts to express the aspect; the underlying semantics of the approach will determine the kinds of reasoning and analysis that can be applied.

Non-functional aspects of systems tend to be qualitative and subjective. Models of non-functional aspects of systems may be more informal and less rigorous than functional models, but this does not mean they should not be captured. Often, functional aspects of systems are developed specifically to correspond with or achieve non-functional objectives. For example, a non-functional model might simply prescribe that the paycheck-processing component be fast. The functional model of the paycheck component may describe how the paycheck-processing component uses caches and local processing—two functional strategies that help to achieve the modeled non-functional aspect.

Much like design rationale, non-functional aspects of systems are difficult to model with rigor. Expressive, free-form notations such as natural language are often used for capturing non-functional aspects. It is useful, however, to employ approaches with support for traceability, which allow modelers to explicitly map non-functional properties to functional design decisions.

6.2 AMBIGUITY, ACCURACY, AND PRECISION

Architectures are abstractions of systems. They capture information about some aspects of the system and leave out other aspects. Ideally, the principal, most important aspects of a system will be well defined by the architecture. The parts that are specified may describe the nominal state of the system and leave out unusual states. This is, to some extent, normal: Architectures are not meant to be complete implementations of a system. Consequently,

the notations used to capture architectures do not have to be completely unambiguous, accurate, and precise.

Three key concepts can be used to characterize architectural models: ambiguity, accuracy, and precision.

6.2.1 Ambiguity

Definition. A model is *ambiguous* if it is open to more than one interpretation.

Because conflicting interpretations of a model may lead to misunderstandings, bugs, and errors, it is generally desirable to eliminate ambiguity in design. Incompleteness is a primary reason for ambiguity in models: When some aspect of a system is left unspecified, different stakeholders may make different assumptions about how the gap should be filled in. Architectures are necessarily incomplete: They address *principal* design decisions about a system, not *every* design decision.

For this reason, it is generally impossible to completely eliminate ambiguity in architectural models. Additionally, the costs of attempting to do so will nearly always outweigh the benefits. Therefore, a balance must be struck. A good guideline is to allow aspects of the system to be ambiguous with the consent of appropriate stakeholders, proceeding when they agree that the architecture is “complete enough” and remaining decisions can be made in a future development activity. While this evaluation proceeds, it is useful to specifically identify and document ambiguous aspects of the architecture as a kind of design rationale.

6.2.2 Accuracy and Precision

Many conceptions of accuracy and precision, including dictionary definitions, conflate the two terms. Here, we will adopt an interpretation that more clearly delineates them.

Definition. A model is *accurate* if it is correct, conforms to fact, or deviates from correctness within acceptable limits.

Definition. A model is *precise* if it is specific, detailed, and exact.

Accuracy deals with correctness, while precision deals with exactness. In architectural terms, a model is accurate if it conveys correct information about the modeled system. A model is precise if it conveys a lot of detailed information about the modeled system. It may seem as if these concepts go hand in hand, but in fact they are somewhat orthogonal.

Figure 6-2 graphically depicts the distinction between accuracy and precision as a set of targets that have been shot. Consider each shot to be an assertion about a system being designed. Figure 6-2(a) shows a set of shots (assertions) that are neither accurate nor precise: they are not close together and are not near the target. Architecturally, this represents design decisions that are both incorrect (that is, inaccurate) and vague or general (that is, imprecise). Figure 6-2(b) shows shots that are accurate, but not precise: They are clustered around the target, but are not close together. These assertions are accurate (correct) but vague and lacking detail. Architecturally, this might be a result of ambiguity or correct descriptions of only a few aspects of the system. Figure 6-2(c) shows shots that are precise, but not accurate: They are clustered close together, but they are not near the

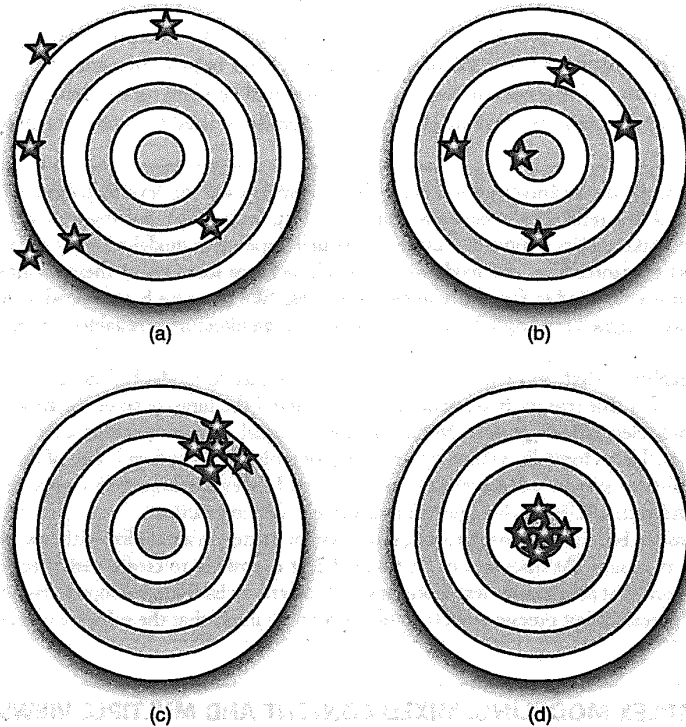


Figure 6-2.
A graphical
depiction
distinguishing
accuracy and
precision.

target. This could be interpreted as assertions about the system that are highly detailed, but wrong. Figure 6-2(d) shows shots that are both precise and accurate: The assertions are both detailed and correct.

In developing an architecture, accuracy should generally be favored over precision. Precision and completeness are, of course, desirable, but later phases of detailed design and implementation will necessarily flesh out the missing detail. However, inaccurate architectural models that mislead stakeholders in these later phases will generally lead to costly errors in later development activities.

Notations and modeling approaches can have a significant effect on the ambiguity, accuracy, and precision of models. Some notations include elements that are purposefully ambiguous—they allow stakeholders to interpret their meaning in the manner most useful to their project. Other notations are based on formal semantics, and encourage less ambiguous and more precise specifications. In these approaches, the designers specify what each element means and how it should be interpreted. Often, these interpretations are built directly into visualization and analysis tools.

Stakeholders can find utility in all kinds of modeling approaches, from the ambiguous and imprecise to those with formal semantics. Approaches that promote reduced ambiguity and increased accuracy and precision are often more costly to use because they have higher

learning curves and require more detailed modeling. For these reasons, stakeholders should generally choose notations that allow unambiguous, accurate, and precise modeling of the aspects of the system *that are the most important*. Other approaches can be used for less important aspects of the system. Since no notation will be ideal for modeling every aspect of a system, combinations of notations should generally be used to capture an architecture.

The Illusion of Quality

Ultimately, we use architecture to ensure that the system we are developing attains specific desired qualities: reliability, robustness, maintainability, evolvability, and many others.

Often, stakeholders assume that using a particular notation or modeling approach will (substantially) guarantee that the modeled system will achieve some set of these qualities. This sentiment is embodied in familiar statements such as, "We know we have a good architecture because we designed it using UML," or "Of course our architecture is reliable—it conforms to DoDAF."

The reality is that, in many notations, it is just as easy to model bad design decisions as good ones. For this reason, it is important to understand the limitations of the notations and approaches selected for a project. Not all are created equal; in some notations, for example, it may be possible to formally verify (with the help of tools) that a system is free of deadlock, or that a real-time system will never miss its deadlines. However, compared to more permissive approaches such as UMLs, these notations are rarely used in practice.

It is easy to be fooled. Consider a model of a system written in a notation with low ambiguity and high precision. As discussed in Section 6.2.2, it is possible to create very precise system specifications that are nonetheless wrong or simply describe a bad design. Good software is often modeled precisely, but precise models do not necessarily imply that the software is good.

6.3 COMPLEX MODELING: MIXED CONTENT AND MULTIPLE VIEWS

Architecture models are complex artifacts. They attempt to capture all the design decisions about a system that are important to a wide variety of stakeholders. Additionally, different aspects of the same concept will be captured simultaneously, for example, a component's interconnections to other components, its behavior, and its version history. There is simply too much information to deal with all at once, and attempting to do so is not productive: Stakeholders generally want to interact with the parts of the architecture that are most important from their own perspective.

In general, no single approach will be able to capture all the aspects of an architecture for a project. This means that various parts of the architecture may have to be modeled using different approaches. For example, non-functional aspects of a system may be captured using natural language, structure with a component-connector graph, and various component behaviors with statecharts.

6.3.1 Views and Viewpoints

This situation induces the notion of *views* and *viewpoints*.

Definition. A *view* is a set of design decisions related by a common concern (or set of concerns).

Definition. A *viewpoint* defines the perspective from which a view is taken.

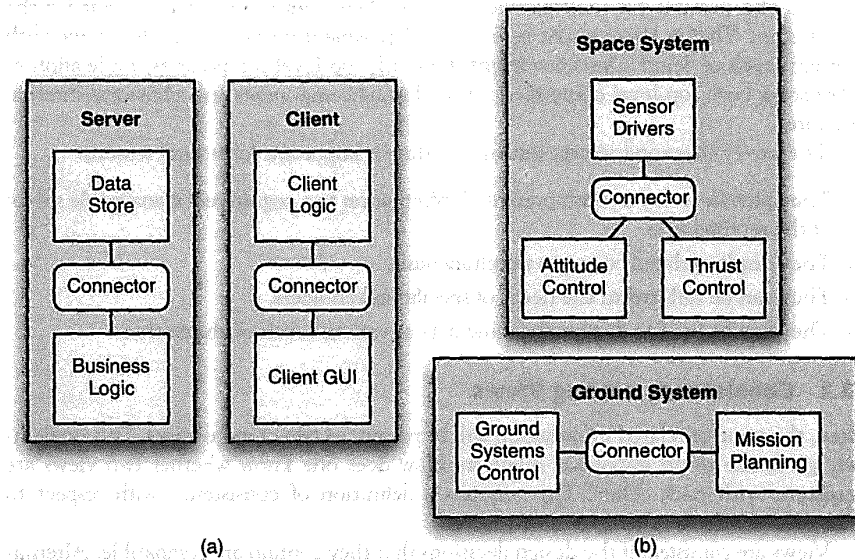


Figure 6-3.
The deployment
views of the
architectures of
two different
systems.

A view is an instance of a viewpoint for a specific system. Put another way, a viewpoint is a filter for information, and a view is what you see when you look at a system through that filter. For example, consider the domain of systems in which software components are distributed across multiple hosts. The *deployment viewpoint* is a perspective on such systems that captures design decisions related to how components are assigned to hosts. The *deployment view* of a particular system captures how components are assigned to hosts for that particular system.

Figure 6-3 shows the deployment views of two different systems. Figure 6-3(a) shows the deployment view of a client-server system. Figure 6-3(b) shows the deployment view of a distributed space-ground system like Lunar Lander. However, both of these views are taken from the deployment viewpoint; that is, they both capture design decisions dealing with the deployment of components onto hosts.

In general, viewpoints are associated with one concern. Examples of viewpoints that are commonly captured include:

- **Logical Viewpoint:** Captures the logical (often software) entities in a system and how they are interconnected.
- **Physical Viewpoint:** Captures the physical (often hardware) entities in a system and how they are interconnected.
- **Deployment Viewpoint:** Captures how logical entities are mapped onto physical entities.
- **Concurrency Viewpoint:** Captures how concurrency and threading will be managed in a system.
- **Behavioral Viewpoint:** Captures the expected behavior of (parts of) a system.

It is also possible for multiple views to be taken from the same viewpoint for the same system. That is, there might be multiple deployment views for a single system, with different levels of detail. One view might show only top-level components, while another might show both top-level components as well as subcomponents and additional internal structure.

The use of views and viewpoints in modeling is important for several reasons:

- They provide a way to limit presented information to a cognitively manageable subset of the architecture.
- They display related concepts simultaneously.
- They can be tailored to the needs of specific stakeholders.
- They can be used to display the same data at various levels of abstraction.

6.3.2 Consistency among Views

Often, the same or related information will be present in two or more views. This immediately gives rise to an important question: How does one know whether two views are consistent with each other? This requires a definition of consistency with respect to architectural views.

Views are *consistent* if the design decisions that they contain are compatible. Alternatively, consistency can be seen as the absence of inconsistency. An *inconsistency* occurs when two views assert design decisions that cannot both be simultaneously true. Many kinds of inconsistency can arise. Some kinds of inconsistency that might occur in a multiple-view architecture description include the following.

Direct Inconsistencies. These inconsistencies occur when two views assert directly contradictory propositions such as “the system runs on two hosts” and “the system runs on three hosts.” These inconsistencies can often be detected by automatic mechanisms that employ appropriate constraints and rules.

Refinement Inconsistencies. These inconsistencies occur when two views of the same system at different levels of detail assert contradictory propositions. For example, a “top level” structural view contains a component that is absent from a structural view that includes subarchitectures. These inconsistencies can also be automatically detected with appropriate consistency rules, provided both the top-level and refined views contain enough information to understand the relationship between the two.

Static Aspects versus Dynamic Aspects. In this inconsistency, a view of a static aspect of a system conflicts with a view of a dynamic aspect. For example, a message sequence chart view might depict the handling of messages by a component that is not contained in the structural view. These inconsistencies can be somewhat harder to automatically detect, depending on how explicit the dynamic aspect’s specification is.

Dynamic Aspects. In this inconsistency, two views of dynamic aspects of the system conflict. For example, a message sequence chart depicts a specific interaction between components that is not allowed by the behavioral specifications contained in those components’ statecharts. These inconsistencies are often extremely difficult to detect automatically because it would require extensive state exploration or simulations.

Functional versus Non-Functional Aspects. These inconsistencies occur when a non-functional property of a system prescribed by a non-functional view is not met by the design expressed in functional views. For example, a non-functional view of a client-server system may express that the system should be robust, but the physical view of the system may show only a single server with no evidence of failure-handling machinery. These inconsistencies are the most difficult to detect because of the general and abstract nature of non-functional properties.

Figure 6-4 shows two hypothetical views of a distributed Lunar Lander system. The physical view (a) depicts three hosts, a Ground System, a Command Module Computer, and a Lander Computer. However, the deployment view (b) shows components assigned to only two hosts, a Ground System and a Lunar Lander. It is fairly easy to see that something is not right here—there is an inconsistency between the physical and deployment view. The inconsistency fits our above definition—the physical view asserts the design decision that Lunar Lander runs on two hosts, while the deployment view asserts the decision that Lunar Lander runs on three.

This inconsistency is relatively easy to spot just by looking. However, other, more subtle inconsistencies (such as inconsistencies between different behavioral specifications of an architecture) are harder to detect and more costly to fix.

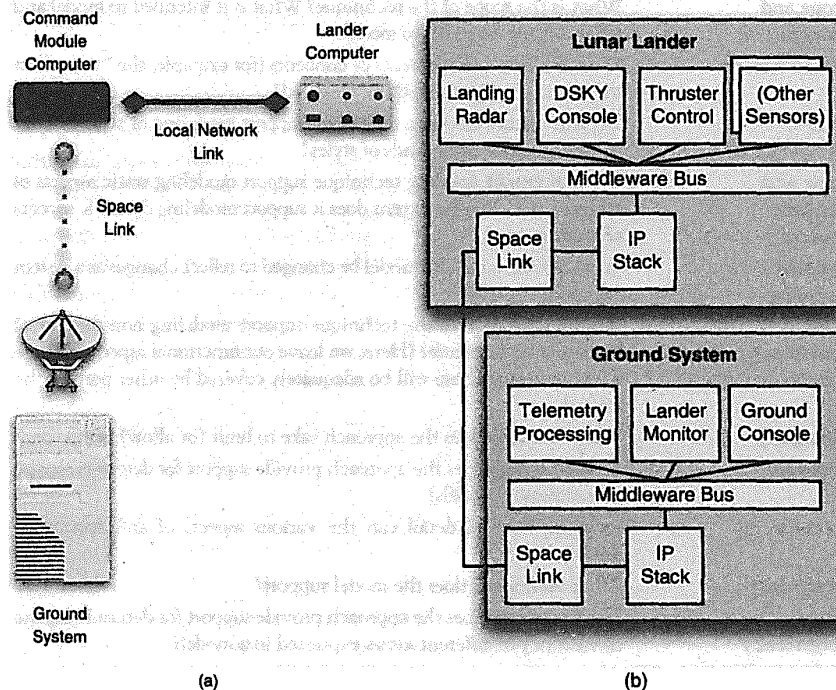


Figure 6-4.
A Lunar Lander
system's physical
view (a) and
logical view (b).

Detecting, identifying, and resolving inconsistencies among views is a difficult problem. First, stakeholders must agree on what “consistency” means for their particular choice of views and modeling notations. Then, strategies must be employed to locate and deal with inconsistencies. These can range from manual approaches such as inspections and checklists to automated approaches such as model checking and simulation. Chapter 8, which covers architectural analysis, discusses in detail specific approaches for dealing with inconsistency.

Inconsistency is generally, but not always, undesirable. Inconsistencies will arise as a natural part of doing exploratory design—as you design an architecture, parts of it may be temporarily inconsistent as the design evolves. Sometimes, inconsistencies are left in an architecture deliberately upon agreement of the stakeholders to deal with a special case, or because repairing the inconsistency would be too costly.

6.4 EVALUATING MODELING TECHNIQUES

In the past sections, we have explored a number of dimensions that can be used to characterize different modeling techniques. These dimensions can be turned into a rubric for critically thinking about and evaluating different modeling techniques. This rubric will be applied to several techniques in the remainder of this chapter.

Modeling Techniques Rubric	Scope and Purpose	What is the scope of the technique? What is it intended to model and what is it <i>not</i> intended to model?
	Basic Elements	What are the basic elements or concepts (for example, the “atoms”) in the technique? How are these modeled?
	Style	To what extent does the technique support modeling of stylistic constraints, and for what kinds of styles?
	Static and Dynamic Aspects	To what extent does the technique support modeling static aspects of architecture? To what extent does it support modeling dynamic aspects of architecture?
	Dynamic Modeling	To what extent can the model be changed to reflect changes as a system executes?
	Non-Functional Aspects	To what extent does the technique support modeling non-functional aspects of architecture? (Here, we leave out functional aspects because we assume that those will be adequately covered by other parts of the rubric).
	Ambiguity	What measures does the approach take to limit (or allow) ambiguity?
	Accuracy	To what extent does the approach provide support for determining the correctness of models?
	Precision	At what level of detail can the various aspects of architecture be modeled?
	Viewpoints	What viewpoints does the model support?
	View Consistency	To what extent does the approach provide support for determining the consistency of different views expressed in a model?

6.5 SPECIFIC MODELING TECHNIQUES

Architects have at their disposal a panoply of notations and techniques for modeling different aspects of architectures. These techniques vary along many dimensions: what they can model, how precisely they can capture architectural semantics, how good their tool support is, and so on. Which methods are used, and for what purposes, will be largely up to a system's architects and stakeholders. While many approaches for modeling architectures are broadly scoped and applicable to many systems in many domains, it is important not to become dogmatically attached to any one approach.

The next sections discuss and evaluate various approaches that can be used for modeling software architectures. A simple version of the Lunar Lander system is used as a running example to illustrate and compare modeling techniques.

6.5.1 Generic Techniques

These techniques are often used to describe a software architecture in whole or in part, although they were not specifically developed or adapted for this purpose. As such, they tend to be flexible but have few semantics.

Natural Language

Natural languages, such as English, are an obvious way to document architectural design decisions. Natural languages are expressive, but tend to be ambiguous, nonrigorous, and nonformal. Natural language cannot be effectively processed and understood by machines, and thus can only be checked and inspected by humans.

Despite these drawbacks, natural-language modeling can be the best way to capture some aspects of an architecture. For example, non-functional requirements are often captured using natural language since they are abstract and qualitative. Natural language is easily accessible to stakeholders and can be manipulated with common tools such as word processors.

An alternative to using pure natural language is to use a restricted form of it. Users can create and consistently employ a dictionary of terms in order to limit certain kinds of problems (for instance, ambiguity that can arise from using different terms for the same concept, or lack of clarity). Statement templates, which are natural language statements or paragraphs with fill-in-the-blanks parameters can be another way to increase rigor in natural language. However, overusing this strategy has a negative effect—essentially, it creates a domain-specific language without any of the flexibility benefits of natural language.¹

¹In the early days of computing, some programming language designers believed that programming languages could be improved by making them more like natural language with added syntactic rigor. Thus, languages like COBOL included statements such as, "MULTIPLY X BY 5 GIVING Z," instead of the terser " $z = x * 5$." Such techniques have fallen out of favor, as they only served to make programs more verbose.

Evaluation Rubric for Natural Language	Scope and Purpose	Describing arbitrary concepts with an extensive vocabulary, but in an informal way.
	Basic Elements	Any concepts required, no special support for any particular concept.
	Style	Stylistic concepts can be described by simply using more general language.
	Static and Dynamic Aspects	Both static and dynamic aspects of systems can be modeled.
	Dynamic Modeling	Models must be changed by humans and can be rewritten with a tool such as a word processor, but there is no feasible way to tie these models to implementation.
	Non-Functional Aspects	Expressive vocabulary available for describing non-functional aspects of systems (although there is no support for verifying them).
	Ambiguity	Plain natural language is perhaps the most ambiguous notation available for expressing architectures; techniques such as statement templates and well-defined dictionaries of terms can be used to reduce ambiguity.
	Accuracy	Correctness must be evaluated through manual reviews and inspections, and cannot be automated.
	Precision	Additional text can be used to describe any aspect of architecture in greater detail.
	Viewpoints	All viewpoints (but no specific support for any viewpoint).
	View Consistency	Correctness must be evaluated through manual reviews and inspections, and cannot be checked automatically.

Lunar Lander in Natural Language

Here, we introduce a simple three-component version of the Lunar Lander application that will be used throughout the remainder of this chapter to illustrate the use of different modeling approaches.

The Lunar Lander application might be described in natural language as shown in Figure 6-5, which provides a fairly significant amount of information about the Lunar Lander system. For example, the structure of the components and their dependencies is explicitly stated, as well as a description of their behaviors, inputs, outputs, and general responsibilities. The description stops short of specifying certain details that are required for implementing the system. For example, it does not explain the algorithm the *calculation* component uses for its computations, the particular formats of the different data values, anything about the connectors between the components, or what the user interface should look like.

Many different implementations could satisfy this architectural description, and they may not all function identically. This is partially due to the nature of architectural models in general—they document *principal* design decisions, not *all* design decisions. It is also partially due to the ambiguity inherent in natural language specifications. This specification is not the tersest or most understandable way of expressing aspects of this system such as its structure—readers of the specification are likely to build up a mental graph of the system's constituent components as they read, since one is not provided.

The Lunar Lander application consists of three components: a **data store component**, a **calculation component**, and a **user interface component**.

The job of the **data store component** is to store and allow other components access to the height, velocity, and fuel of the lander, as well as the current simulator time.

The job of the **calculation component** is to, upon receipt of a burn rate quantity, retrieve current values of height, velocity, and fuel from the **data store component**, update them with respect to the input burn rate, and store the new values back. It also retrieves, increments, and stores back the simulator time. It is also responsible for notifying the calling component of whether the simulator has terminated, and with what state (landed safely, crashed, and so on).

The job of the **user interface component** is to display the current status of the lander using information from both the **calculation** and the **data store** components. While the simulator is running, it retrieves the new burn rate value from the user, and invokes the **calculation component**.

Takeaways. Natural language should be used as an adjunct to more rigorous and formal languages, for aspects of architecture where formalism is infeasible or unnecessary. It is particularly good for specifying non-functional properties in a way that almost no other language can match.

Informal Graphical PowerPoint-style Modeling

Tools such as Microsoft PowerPoint (Microsoft Corporation 2007) and OmniGraffle (The Omni Group 2007) provide users with the ability to create decorative diagrams of interconnected shapes through a point-and-click graphical interface. These are ubiquitous and it is easy to use them to create aesthetically pleasing diagrams. The size limitations of such diagrams (one slide, one sheet of paper) help to ensure that they remain suitably abstract.

Informal graphical notations have a lot in common with natural languages—they provide informal, unconstrained ways of expressing ideas (although they primarily use symbols rather than words). Informal graphical diagrams often capture few (if any) semantics. This makes it difficult to reliably interpret the meaning of the diagrams, undermining the key advantages of using an architecture in the first place (communication, analysis, and so on). These diagrams are good for early prototyping and exploration, capturing ideas at an abstract, conceptual level. However, they lack the rigor required for more concrete models.

Figure 6-5.
Lunar Lander
model in natural
language
(American
English).

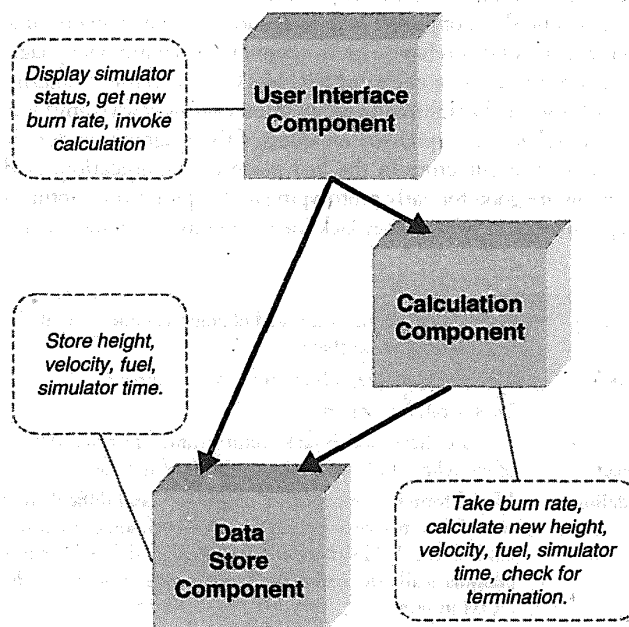
Scope and Purpose	Arbitrary diagrams composed of graphical and textual elements, with few restrictions on them.	Evaluation Rubric for Informal Graphical Modeling
Basic Elements	Geometric shapes, splines, text strings, clip art.	
Style	In general, no support.	
Static and Dynamic Aspects	Because these models lack architectural semantics, there is no notion of modeling static or dynamic aspects of systems.	
Dynamic Modeling	Model formats are generally proprietary and difficult to change outside of their native environment. However, some environments expose externally callable interfaces that allow models to be manipulated programmatically (for example, PowerPoint exposes its models through a COM interface).	

Non-Functional Aspects	Non-functional aspects can be modeled using natural-language decorations on diagrams.
Ambiguity	As with natural language, ambiguity can be controlled through the use of a controlled symbolic vocabulary or dictionary.
Accuracy	In general, correctness is determined through manual inspection and cannot be checked automatically.
Precision	Modelers can choose an appropriate level of detail; however, they are generally limited by the amount of information that can fit on one diagram or slide.
Viewpoints	Because these models lack architectural semantics, there is no direct support for multiple viewpoints. In theory, a model can show any viewpoint.
View Consistency	In general, consistency is determined through manual inspection and cannot be checked automatically.

Lunar Lander in PowerPoint. There are many ways to express the Lunar Lander architecture in a tool like Microsoft PowerPoint, limited only by the symbol palette available in the tool.

A Lunar Lander architectural model in PowerPoint is shown in Figure 6-6. As with most PowerPoint architecture diagrams, the particular symbology used is not directly explained—there is no underlying semantic model (other than the reader's expectations)

Figure 6-6.
Lunar Lander
architectural
model in
PowerPoint.



through which to interpret this diagram. The meanings of the three-dimensional boxes, the arrows, and the rounded rectangles are not provided. More serious modeling efforts might provide a symbolic dictionary to accompany such a diagram. However, this can be dangerous: Even when users employ consistent or well-understood symbols, informal diagramming tools cannot provide automated consistency checks or analysis.

The intended interpretation of this particular diagram is that the three-dimensional boxes are software components, the arrows indicate invocation dependencies, and the rounded rectangles are commentary on the intended behavior and responsibilities of the components. Assuming that the reader correctly interprets the meanings of the symbols, this model is in certain ways easier to understand than the natural language model. For example, the componentization of the system and the component dependencies are immediately obvious, and the behaviors are visually connected to the components.

Takeaways. PowerPoint and similar tools are seductive because they make it very easy to create graphical diagrams. However, great care should be taken in using informal graphical diagrams as part of any official architecture description—they can be valuable for early back-of-the-napkin architecture descriptions, but as soon as basic decisions are made, more rigorous notations should be employed for real architecture capture.

The Unified Modeling Language (and Its Cousins)

UML, the Unified Modeling Language (Booch, Rumbaugh, and Jacobson 2005), is the most popular notation for writing down software designs. It is unified in that it combines concepts from earlier notations such as Booch diagrams (Booch, 1986), James Rumbaugh's OMT (Rumbaugh et al. 1990), Ivar Jacobson's OOSE (Jacobson 1992), and David Harel's Statecharts (Harel 1987). UML's main strengths are its large variety of modeling constructs and viewpoints, extensive tool support, and widespread adoption.

UML is a massive notation about which entire books can be and have been written. We summarize its role as an architectural modeling language here. Further information about UML as a standard for software and system modeling is presented in Chapter 16.

UML is an inherently graphical notation, with views consisting of textually annotated graphical symbols. It provides its users with an extremely wide variety of modeling constructs and concepts. UML 2.0 includes thirteen different viewpoints, called “diagrams” in UML parlance. These viewpoints cover many of the dimensions identified earlier in this chapter, including basic architectural elements, stylistic constraints, and static and dynamic aspects of systems.

Much debate has ensued over the years as to UML's suitability for modeling software systems at the architecture level. Early versions of UML (1.0 and 1.1) (Booch, Rumbaugh, and Jacobson 1998) had a strong focus on detailed design—below the level of common architectural constructs such as components and connectors. They were biased toward the design of object-oriented systems that communicate primarily through procedure calls. UML 2.0 significantly expanded UML to provide much better support for higher-level architectural constructs. Existing viewpoints were extended with new elements and entirely new viewpoints were added.

UML's broad range of diagrams makes it an attractive option for modeling all kinds of software systems. However, it is vitally important for architects not to get locked in to



Figure 6-7. A UML component diagram showing a dependency between two components.

UML as their one and only modeling solution for architectures, or to overestimate the benefits conferred by using UML.

UML's viewpoints remain mostly focused on design. In many ways, these viewpoints still retain an inherent bias toward object-oriented systems. Some of the viewpoints extend into other life cycle activities, capturing some requirements- and implementation-related aspects. Explicit support is minimal for activities such as testing and maintenance and meta-activities such as management. Architectural decisions can affect any of these aspects of development; this is why additional notations should be used to capture a complete architecture.

Understanding the semantics of UML diagrams is key to making them useful in describing architectural design decisions. UML is more precise than arbitrary diagrams that would be produced in, say, PowerPoint. However, it has been designed to be purposefully ambiguous in many respects so as to increase its generality.

For example, the dashed open-headed arrow is a "dependency" arrow. It indicates that the element at the tail of the arrow has some dependency on the element at the head of the arrow. Figure 6-7 shows a simple UML component diagram with two components. The dashed arrow indicates that the Calculation component is dependent on the Data Store component. However, this could mean any number of things, including:

- Some element of Calculation calls Data Store.
- Instances of Calculation contain a pointer or reference to an instance of Data Store.
- Calculation requires Data Store to compile.
- Calculation's implementation has a method that takes an instance of Data Store's implementation as a parameter.
- Calculation can send messages to Data Store.

And so on, and so on. Most constructs in UML are similarly semantically ambiguous. (It is even unclear what the components in Figure 6-7 refer to: They could be classes in an object-oriented programming language, C modules, Web services, or something else.)

This approach gives UML great flexibility but limits its semantic precision. Fortunately, UML includes facilities that allow its users to define new attributes (called *stereotypes* and *tagged values*) and constraints that can be applied to existing elements to specialize them. Constraints in UML can be specified in any language (including natural language), but a companion language called the Object Constraint Language (OCL) provides a convenient way to write rigorous constraints for UML. A collection of these additional attributes and constraints is known as a UML *profile*. Profiles are generally designed for specific applications, product lines, or domains, and serve to increase the semantic precision of UML diagrams within that scope.

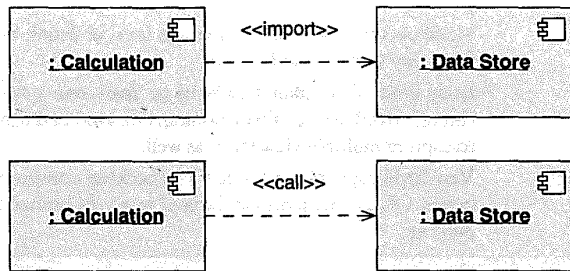


Figure 6-8.
UML component
diagrams with
more specific
semantics
provided by using
stereotypes.

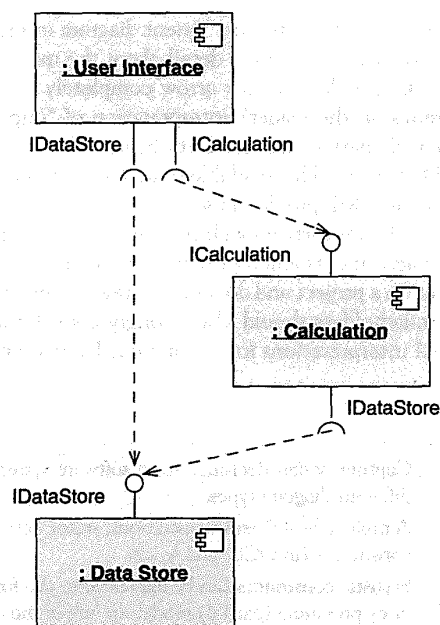
Figure 6-8 shows two variants of the component diagram in Figure 6-7. Each of these uses a UML *stereotype* to provide additional detail about the meaning of the dependency arrow. This does not make the dependency arrow completely unambiguous, however—understanding it still relies on the reader's interpretation of “import” or “call.” The top diagram indicates that Calculation imports Data Store. The bottom diagram indicates that Calculation calls Data Store. The available selection of stereotypes and their specific meanings are defined by the UML profile in use.

UML diagrams alone do not carry enough information to completely interpret them. Stakeholders may make agreements among themselves about how to interpret particular aspects of UML diagrams on a project and document these agreements in external natural-language documents. Stakeholders should also strongly consider defining or selecting a profile with documented interpretations for the included stereotypes, tagged values, and constraints.

Scope and Purpose	Capture design decisions for a software system using up to thirteen different diagram types.	Evaluation Rubric for UML
Basic Elements	A multitude—classes, associations, states, activities, composite nodes, constraints (in OCL) and so on.	
Style	Stylistic constraints can be expressed in the form of OCL constraints or by providing (partial) models in one of the many viewpoints.	
Static and Dynamic Aspects	Includes a number of diagrams for modeling both static (for example, class diagram, object diagram, package diagram) and dynamic aspects (for example, state diagram, activity diagram) of systems.	
Dynamic Modeling	Depends on the modeling environment; in practice very few systems are tied directly to a UML model such that the UML model can be updated as they run.	
Non-Functional Aspects	No direct support, except perhaps in textual annotations.	
Ambiguity	In general, UML elements can mean different things in different contexts. Ambiguity can be reduced through the principled use of UML profiles, including stereotypes, tagged values, and constraints.	
Accuracy	Aside from OCL constraint checking and basic well-formedness checks (that is, no broken links, every element has a name, and so forth), there is no standard for assessing the accuracy of a UML model.	

Precision	Modelers can choose an appropriate level of detail; UML offers wide flexibility in this regard.
Viewpoints	Each kind of diagram represents at least one possible viewpoint; through overloading or partitioning, one kind of diagram can be used to capture multiple viewpoints as well.
View Consistency	Very little support is provided for checking consistency among diagrams; OCL constraints can be used in a very limited fashion for this purpose.

Figure 6-9.
Lunar Lander
component
diagram in
UML.



Lunar Lander in UML. UML provides many possible viewpoints from which to model the Lunar Lander architecture. Stakeholders are responsible for choosing which viewpoints are used and how precise each view will be. Here, we will use the component, statechart, and sequence viewpoints to describe the Lunar Lander at roughly the same level of detail as has been shown in the natural language and informal graphical examples.

The component diagram for Lunar Lander might look like Figure 6-9.² This diagram looks very similar to the informal graphical diagram, because it largely depicts the same

²This diagram uses the UML 2 ball-and-socket icons to represent provided and required interfaces, respectively. In earlier versions of UML 2, the socket and ball could be directly graphically joined (without an intervening dependency arrow) to indicate connection, but this diagrammatic style has been deprecated by the UML committee except in composite structure diagrams.

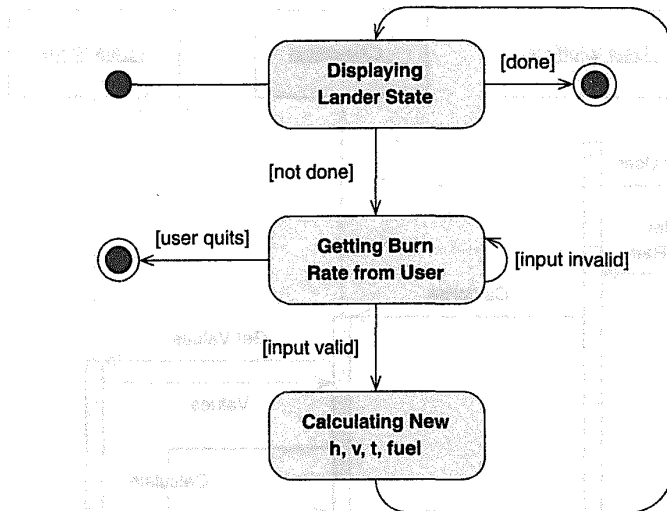


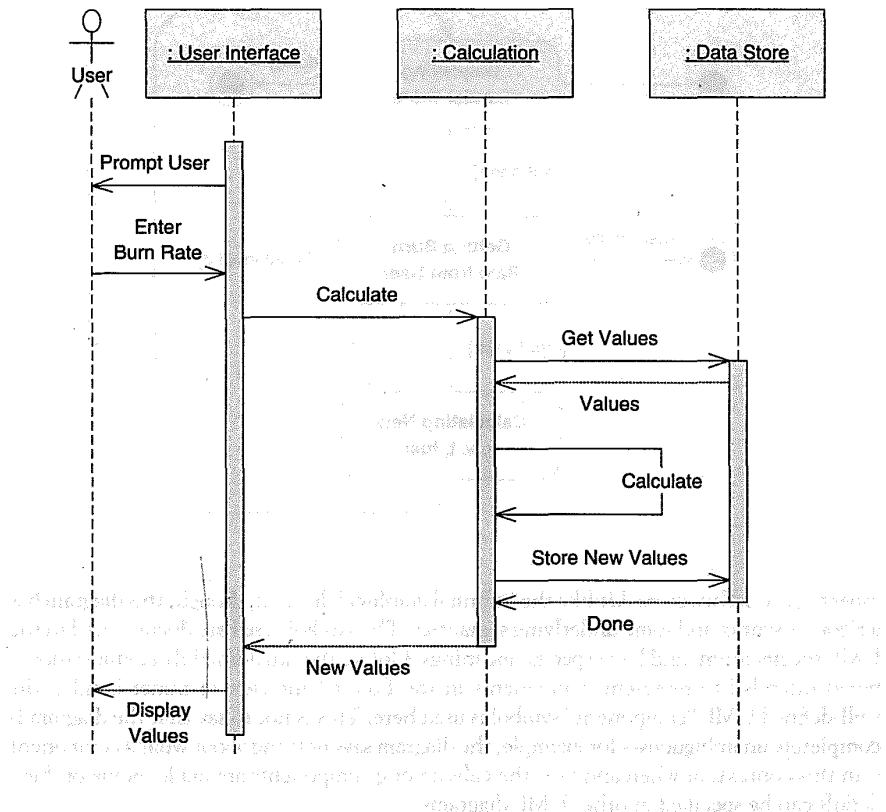
Figure 6-10.
Lunar Lander
statechart
diagram in
UML.

aspect of the architecture. Unlike the informal graphical diagram, though, this diagram has a rigorous syntax and some underlying semantics. The symbols used are documented in the UML specification, and have specific meanings. Unlike the unadorned three-dimensional boxes intended to represent components in the PowerPoint view of Lunar Lander, the well-defined UML “component” symbol is used here. This is not to say that the diagram is completely unambiguous—for example, the diagram says nothing about what a component is in this context, or when and how the calls among components are made. Some of these details can be specified in other UML diagrams.

The behavior of the system might be specified in a UML statechart diagram, as shown in Figure 6-10. The start state is indicated by the plain dark circle, and the end state is indicated by the outlined dark circle. Each rounded rectangle represents a state of the system, and arrows represent transitions between the states. The conditions in square brackets indicate guards that constrain when state transitions may occur. This statechart indicates that the Lunar Lander system begins by displaying the lander state. If the simulation is done, the simulator will stop. Otherwise, the system will request a burn rate from the user. Here, the user may choose to end the program early. Otherwise, if the burn rate is valid, the program will then calculate the new simulator state and display it. This control loop will repeat until the simulation is done. While this statechart diagram provides a more rigorous and formal description of the system behavior than either the natural language or informal graphical architecture description, it leaves out some important details contained in both those descriptions—namely, which components perform the specified actions. This information can be captured in another UML diagram, such as a UML sequence diagram.

A sequence diagram for the Lunar Lander application might look like Figure 6-11. This diagram depicts a particular sequence of operations that can be performed by the three Lunar Lander components. Sequence diagrams such as this one are not intended to

Figure 6-11.
Lunar Lander
sequence diagram
in UML.



capture all the behavior of a system; rather, they are used to document particular scenarios or use cases. The above diagram depicts one nominal scenario—User Interface gets a burn rate from the user, Calculation retrieves the state of the lander from the Data Store and updates it, and then returns the termination state of the lander to User Interface. Figure 6-9, Figure 6-10, and Figure 6-11 represent three different views of the Lunar Lander architecture. These views capture both static (structural) and dynamic (behavioral) aspects of the system. A natural question to ask is whether these views are consistent with one another. There is no standard way to answer this question, as there is no universal notion of consistency in UML. Instead, we have to establish our own criteria for consistency and check the views against these criteria. For example, we could check whether each component in the component diagram is represented in the sequence diagram, and whether the calls in the sequence diagram are allowed by the dependency arrows in the component diagram.

Checking the consistency of the statechart and sequence diagrams is a more difficult task. These two diagrams model behavioral aspects of the architecture at substantially different levels. The statechart describes the overall functioning of the system without regard to its structure, while the sequence diagram shows a particular scenario in a different

context: the interactions between components. In this case, inspections and stakeholder agreement are probably the best way to determine the consistency of the diagrams.

Takeaways. UML is a syntactically rich notation with extensive tool support. As a way of expressing architectural design decisions, it is superior to a symbols-only notation such as PowerPoint's. However, UML's purposeful ambiguity about the meaning of most of its symbols leaves it open for abuse. Even well-intentioned development groups can interpret the same UML diagrams in different ways, especially when stakeholders are geographically separated and do not have face-to-face opportunities to come to a mutual understanding. When UML is employed, profiles must be developed and used to ensure consistent modeling, although this does not necessarily guarantee unambiguous interpretations. Profiles are not a panacea.

6.5.2 Early Architecture Description Languages

The 1990s spawned nearly a decade of research on how to best capture software architectures. The result of this research was a proliferation of architecture description languages (ADLs), notations developed specifically for software architecture modeling.

During this period, there was substantial debate over what does and does not constitute software architecture. This debate continues today. A parallel debate naturally emerged: Which notations or modeling approaches can be called "architecture description languages?" There is no established litmus test to determine whether a particular notation can or cannot be called an ADL, and there is significant variability in the spectrum of languages that are identified by their creators as ADLs. Nenad Medvidović and Richard Taylor (Medvidović and Taylor 2000) surveyed a wide variety of early ADLs and found that the common denominator among them was explicit support for modeling:

- Components.
- Connectors.
- Interfaces.
- Configurations.

Additionally, they found that languages identified as ADLs tended to be semantically precise but lacked breadth and flexibility. This largely mirrored the prevailing sentiment about software architecture; namely, that it was a high-level structural view of a system extended with rich semantics.

This book takes a much broader view on software architecture. While high-level component-and-connector views are still important when describing a software system's architecture, our definition of architecture as the set of principal design decisions about a system encompasses a much broader range of concepts.

During the 1990s, there was significant debate about whether design notations like UML could be called "architecture description languages." Early versions of UML lacked explicit support for component-and-connector style modeling as well as precise semantics, leading many researchers to conclude that UML was not, in fact, an architecture description language. In our broader conception of architecture, any language used to capture principal design decisions is effectively an architecture description language, including UML.

The early “first-generation” architecture description languages described in this chapter are, for the most part, research projects. None of these is still used actively in practice. We present them here because of their unique contributions to the field of architecture modeling, but their use in practice may prove difficult for extrinsic reasons, such as lack of current tool support.

Darwin

Darwin (Magee et al. 1995) is a general-purpose architecture description language for specifying the structure of systems composed from components that communicate through explicit interfaces. Darwin has a canonical textual representation in which components and their interconnections are described. There is an associated graphical visualization that depicts Darwin configurations in a more accessible but less precise way.

In Darwin, systems are modeled as a set of interconnected components. There is no notion of explicit software connectors in Darwin, but a component that facilitates interactions could be interpreted as a connector. Darwin components expose a set of provided and required services, sometimes called *ports*. Services in Darwin correspond to our notion of provided and required architectural interfaces. Configurations are specified by a set of bindings between interfaces. Darwin also has support for hierarchical composition—that is, components that have internal structures also consisting of components, services, and bindings.

Evaluation Rubric for Darwin	Scope and Purpose	Structures of distributed systems that communicate through well-defined interfaces.
	Basic Elements	Components, interfaces (required and provided), links (called bindings), hierarchical composition.
	Style	Limited support through the use of parameterizable constructs.
	Static and Dynamic Aspects	Support for static structural views, additional support for dynamic architectures (for example, those that change at run time) through lazy and direct dynamic instantiation and binding.
	Dynamic Modeling	Not available.
	Non-Functional Aspects	Not available.
	Ambiguity	How Darwin’s constructs (for instance, components, interfaces) can be composed and used within a Darwin model is well defined. Their external meaning (for instance, what it means to be a component or an interface) is subject to stakeholder interpretation.
	Accuracy	Darwin can be modeled in the formalism known as the <i>pi</i> -calculus, which allows the models to be checked for internal consistency.
	Precision	Detail limited to structural elements and their interconnections.
	Viewpoints	Structural viewpoints, deployment viewpoints through the use of hierarchical composition.
	View Consistency	Not available.

Lunar Lander in Darwin. Darwin is primarily a structural description notation, and we can use it to describe the three-component Lunar Lander’s structure. The textual visualization of

```

component DataStore{
  provide landerValues;
}

component Calculation{
  require landerValues;
  provide calculationService;
}

component UserInterface{
  require calculationService;
  require landerValues;
}

component LunarLander{
  inst
    U: UserInterface;
    C: Calculation;
    D: DataStore;
  bind
    C.landerValues -- D.landerValues;
    U.landerValues -- D.landerValues;
    U.calculationService -- C.calculationService;
}

```

Figure 6-12.
Lunar Lander
architecture in
Darwin's textual
visualization.

Lunar Lander in Darwin might be expressed as shown in Figure 6-12. Here, each component is described with explicit provided and required interfaces. The overall application structure is defined using a top-level component with an internal structure; that is, the Lunar Lander application itself is a component containing the `UserInterface`, `Calculation`, and `DataStore` components.

As stated above, Darwin also has a canonical graphical visualization; the model expressed in that visualization might look like Figure 6-13. The use of multiple visualizations to depict the same architectural model will be discussed extensively in the next chapter.

One of the most interesting aspects of Darwin is the set of constructs with which configurations of components can be specified. Many declarative ADLs simply enumerate all the components and bindings in an architecture one by one (and Darwin supports this—the above model of Lunar Lander uses Darwin in this way). However, Darwin also supports the creation of configurations using programming-language-like constructs such as loops. For example, consider a Web application where a number of identical clients are all connecting to the same Web server. We might model this architecture in Darwin shown in Figure 6-14.

Graphically, this model might look like Figure 6-15. In this Darwin model, the actual number of clients is parameterizable. Using these facilities, relatively terse models can be constructed that describe a wide variety of architectures without a great deal of redundancy in the model.

Takeaways. Darwin represents a more rigorous and formal way of capturing an architecture's structure than any notation we have seen thus far in the chapter. It provides

Figure 6-13.
Lunar Lander
architecture in
Darwin's
graphical
representation.

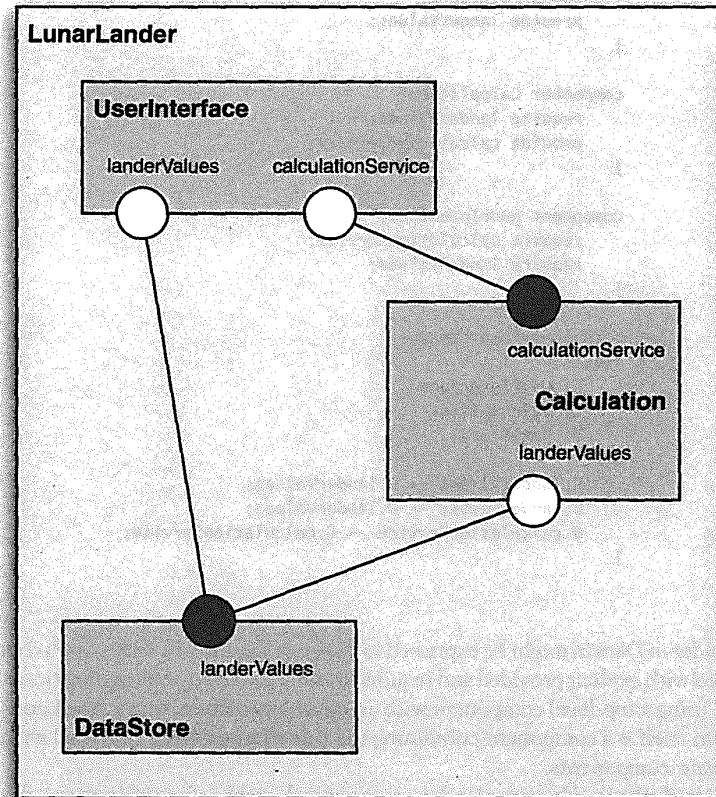


Figure 6-14.
Model of a
multiclient Web
application in
Darwin's textual
visualization.

```

component WebServer{
  provide httpService;
}

component WebClient{
  require httpService;
}

component WebApplication(int numClients){
  inst S: WebServer;
  array C[numClients]: WebClient;
  forall k:0..numClients-1{
    inst C[k] @ k;
    bind C[k].httpService -- S.httpService;
  }
}

```

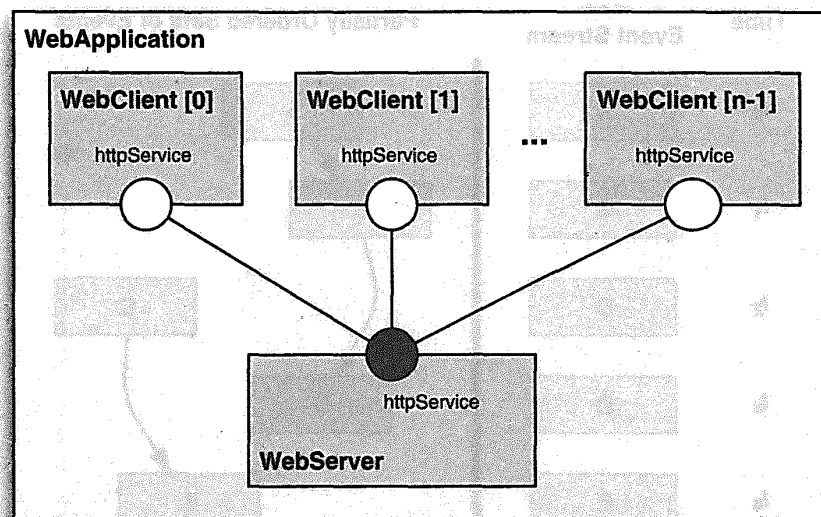


Figure 6-15.
Model of a
multiclient Web
application in
Darwin's
graphical
visualization.

a well-defined textual syntax with an associated graphical visualization, along with specific, well-defined semantics. It is not overly complex and can be understood by straightforward reading. For structural modeling, Darwin is an excellent choice. For modeling other aspects of architecture, however, other notations should be used.

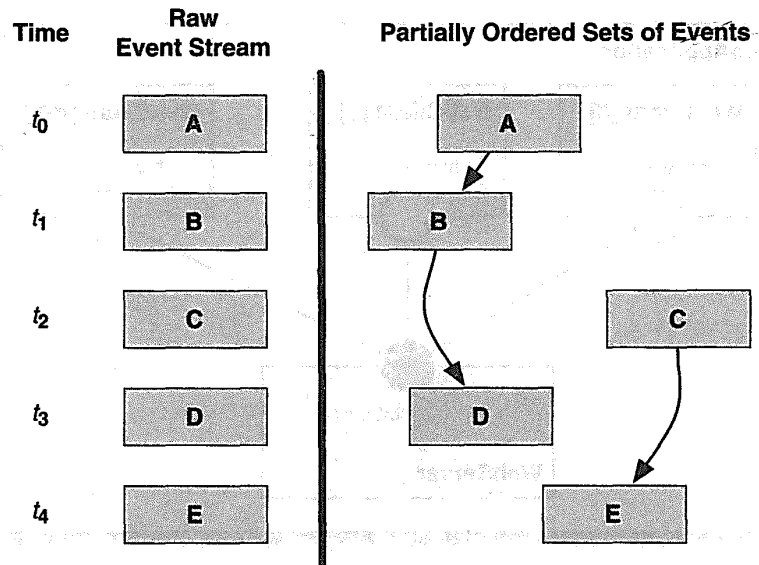
Rapide

Rapide (Luckham et al. 1995) is an architecture description language developed to specify and explore dynamic properties of systems composed of components that communicate using events. In Rapide, events are simply the result of an activity occurring in the architecture. Traditional procedure calls are expressed as a pair of events: one for the call and one for the return value.

The power of Rapide comes from its organization of events into partially ordered sets, called POSETs (Luckham 2002). Rapide components work concurrently, emitting and responding to events. There are causal relationships between some events: for example, if a component receives event A and responds by emitting event B, then there is a causal relationship from $A \rightarrow B$. Causal relationships between events A and B in Rapide exist when any of the following are true (from the Rapide documentation):

- A and B are generated by the same process.
- A process is triggered by A and then generates B.
- A process generated A and then assigns to a variable v , another process reads v and then generates B.
- A triggers a connection that generates B.
- A precedes C which precedes B (transitive closure).

Figure 6-16.
Partially ordered
sets of events.



As a program runs, its various components will generate a stream of events over time. Some of these events will be causally related (by one of the above relationships). Some of them may occur around the same time as other events, but be causally unrelated.

Figure 6-16 depicts this situation. The left portion of the figure shows a raw event stream over time: Components in the architecture send events A, B, C, D, and E at times t_0 through t_4 , respectively. These events are temporally ordered, but not causally ordered. The right portion of the figure shows the causal ordering of the events. Here, there are two partial order sequences: one consisting of events A, B, and D, and one consisting of events C and E. This is called a *partial* order because not all the events are causally ordered with respect to one another. The fact that B occurred earlier in time than C is only a coincidence—an accident of the scheduler: C could just as easily have occurred before B.

Rapide focuses on capturing dynamic aspects of software architectures. Architecture specifications in Rapide are interesting because they are *executable*. Rapide is accompanied by a tool that allows users to execute the architecture description, in effect simulating the operation of the described system. The result is a graph of events similar to the one depicted above in Figure 6-16, showing events as nodes and causal orderings as directed edges.

**Evaluation
Rubric for
Rapide**

Scope and Purpose

Interactions between components in terms of partially ordered sets of events.

Basic Elements

Architectures (structures), interfaces (components), actions (messages/events), and operations describing how the actions are related to one another.

Style

Not available.

Static and Dynamic Aspects	Interconnections among components capture static structure of the architecture, actions and behaviors capture dynamic aspects of the architecture.
Dynamic Modeling	Architectural models do not change during run time, although some tools provided limited animation capabilities.
Non-Functional Aspects	Not available.
Ambiguity	The semantics of the behaviors are well defined.
Accuracy	An automatic simulator produces results that can be examined to determine if the architecture's behavior matches expectations. Rapide constraints can also be checked automatically for violations.
Precision	Interconnections and behaviors in terms of event exchanges can be modeled in detail.
Viewpoints	A single structural/behavioral viewpoint.
View Consistency	There is no way to automatically check view consistency (for example, between multiple models describing the same architecture). Inspections can be used to check for consistency across simulator outputs.

Lunar Lander in Rapide. The description of Lunar Lander in Rapide looks, on the surface, similar to a Darwin description: A textual notation resembling a programming language is used to define the components and their interface operations. However, the Rapide description also includes behavioral information. This information is used, along with the application structure, to generate event graphs. Since Rapide is optimized for operating on concurrent applications where multiple threads of control interact, we will use it to evaluate a cooperative two-player version of Lunar Lander.

Such an architecture might be specified as shown in Figure 6-17. This specification begins by defining the types of available components primarily in terms of their interfaces. Each interface has a number of events it can receive and send out ("in" and "out" actions, respectively). Each component has a behavioral specification as well, defining how it reacts to different events. This simple example takes advantage of only a small number of Rapide language features; it has support for defining and manipulating the data that makes up the events as well. A more complex Lunar Lander specification might actually be able to simulate the entire Lunar Lander game, including data, new state calculations, and so on.

At the end of the specification, the system's structure is defined: first components that implement the various interface types, and then links between the component interfaces. This Rapide architecture is relatively straightforward: The players start off by sending an updated burn rate, making their first move of the game. Then, they wait for the display to be updated with new status before making another move. Players are limited to three moves in this system so the game does not go into an infinite loop (as there is no other end-game condition). The Calculation component waits for a `SetBurnRate` event. When it receives one, it will fire an internal event, `CalcNewState`, and then fire a `DoSetValues` message to the `DataStore` component to update the game state. When the game state is updated, the `DataStore` fires a `NotifyNewValues` event,

Figure 6-17.
Cooperative
two-player Lunar
Lander
architecture in
Rapide.

```

type DataStore is interface
  action in SetValues();
  out NotifyNewValues();
  behavior
  begin
    SetValues => NotifyNewValues();;
  end DataStore;

type Calculation is interface
  action in SetBurnRate();
  out DoSetValues();
  behavior
  action CalcNewState();
  begin
    SetBurnRate => CalcNewState(); DoSetValues();;
  end Calculation;

type Player is interface
  action out DoSetBurnRate();
  in NotifyNewValues();
  behavior
  TurnsRemaining : var integer := 1;
  action UpdateStatusDisplay();
  action Done();
  begin
    (start or UpdateStatusDisplay) where \
      ($TurnsRemaining > 0) => \
      if ( $TurnsRemaining > 0 ) then \
        TurnsRemaining := $TurnsRemaining - 1; \
        DoSetBurnRate(); \
      end if;;
    NotifyNewValues => UpdateStatusDisplay();;
    UpdateStatusDisplay where $TurnsRemaining == 0 \
      => Done();;
  end UserInterface;

architecture Lander() is
  P1, P2 : Player;
  C : Calculation;
  D : DataStore;
  connect
    P1.DoSetBurnRate to C.SetBurnRate;
    P2.DoSetBurnRate to C.SetBurnRate;
    C.DoSetValues to D.SetValues;
    D.NotifyNewValues to P1.NotifyNewValues();
    D.NotifyNewValues to P2.NotifyNewValues();
  end LunarLander;

```

which causes the players' displays to be updated, thus prompting them to make their next moves.

In prose, as described here, this implementation strategy might sound perfectly reasonable. But is it? Rapide's analysis and simulation capabilities can help to make that determination. If we remove the second player from the specification to create only a

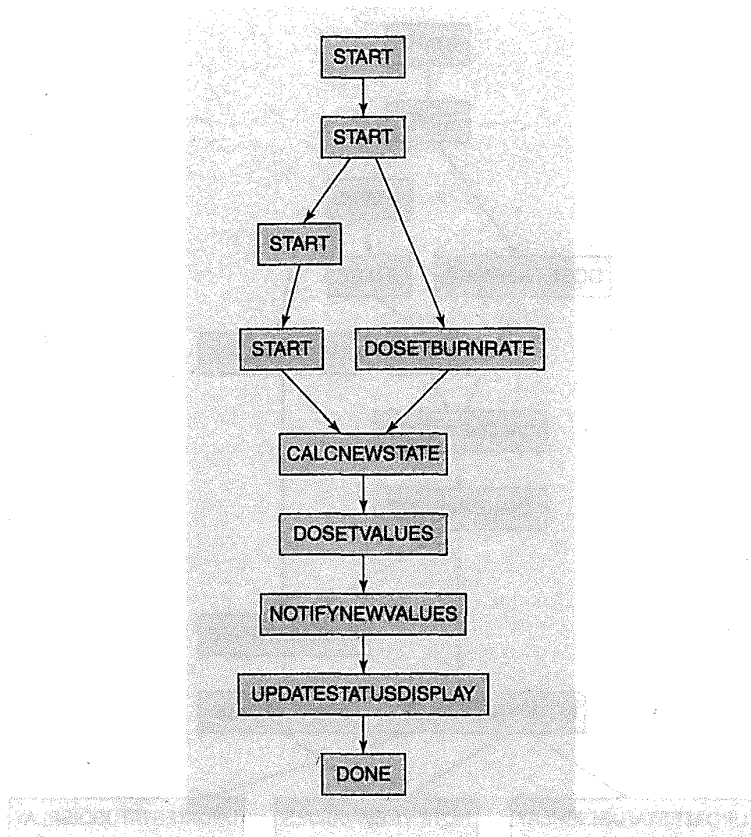
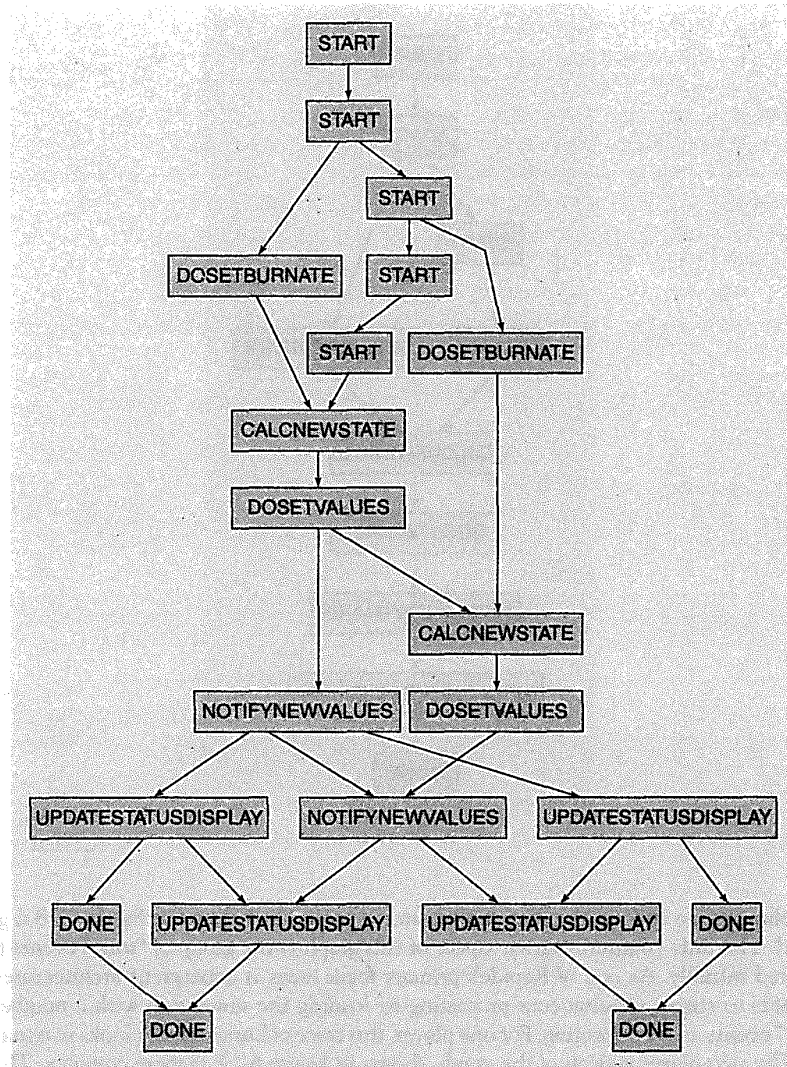


Figure 6-18.
Event graph of a
one-player Lunar
Lander
architecture.

one-player game and run the Rapide simulator, a graph like the one in Figure 6-18 is generated. The only counterintuitive aspect of this graph is the group of “start” events that are fired initially. As one of Rapide’s primary focus areas is concurrent architectures, it attempts to trigger simultaneous processing by loading the simulation with a number of “start” events at the beginning. For one player, this trace of Lunar Lander looks reasonable.

The two-player version of the graph, shown in Figure 6-19, is more complex. This is to be expected. However, by examining the causality arrows, we can see that something is not quite right. The two pathways are intertwined. Requests are getting intermingled, since there is no locking or transaction support in this design. Furthermore, we can see a fan-out of display updates at the bottom. Each user’s display is getting updated twice—once for their own move and once for their partner’s. For one turn, this is fine, but recall that the specification calls for the player to make a move each time the display is updated. With more players and more moves come more display updates, and each one will cause a player to try to move, compounding the effect of the interleaving. This is a non-obvious bug in the specification: All players should not reactively move after each screen update. Without

Figure 6-19.
Event graph of a
two-player Lunar
Lander
architecture.



Rapide's graphs, however, it is not easy to see this. This is the kind of bug that can remain undetected until it is discovered late in development during implementation or testing, when it is costly to fix.

Takeaways. Rapide addresses dynamic aspects of architecture directly, and provides tool support for the simulation of those aspects. Stakeholders can directly see how the components in a software system are intended to interact and are dependent upon each other by looking at the results of these simulations. Rapide has several significant drawbacks,

however: The notation is arcane, with a steep learning curve, and it lacks support for implementing architectures in a way that is consistent with their specifications.

Wright

Wright (Allen and Garlan 1997) is focused on checking component interactions through their interfaces. Interfaces in Wright are specified using a notation derived from the Communicating Sequential Processes (CSP) formalism (Hoare 1978). Wright specifications can be translated into CSP, and then analyzed with a CSP analysis tool like FDR [Formal Systems (Europe) Ltd. 2005]. These analyses can determine, for example, if connected interfaces are compatible, and whether the system will deadlock. Additionally, Wright has the ability to specify stylistic constraints as predicates over instances.

Scope and Purpose	Structures, behaviors, and styles of systems that are composed of communicating components and connectors.	Evaluation Rubric for Wright
Basic Elements	Components, connectors, ports and roles (equivalent to interfaces), attachments, styles.	
Style	Supported through predicates over instance models.	
Static and Dynamic Aspects	Static structural models are annotated with behavioral specifications that describe how components and connectors should both behave and interact.	
Dynamic Modeling	Not available.	
Non-Functional Aspects	Not available.	
Ambiguity	Wright specifications can be translated into the CSP formalism and thus have a formal semantic basis. However, external meaning (for example, specifically what it means to be a component or a connector) is not specified.	
Accuracy	Wright can be modeled in the formalism known as CSP, which allows the models to be automatically checked for, for example, structural consistency and deadlock freedom.	
Precision	Can describe architecture structures and their behavior in great detail (in fact, formal behavioral specifications are required for analysis).	
Viewpoints	Structural viewpoints, behavioral viewpoints, style viewpoints.	
View Consistency	Structural and behavioral aspects are intertwined because behavior is specified as a decoration on structural elements. Consistency between style and instances can be automatically determined with a CSP evaluator.	

Lunar Lander in Wright. The structural aspects of Lunar Lander, modeled in Wright, look similar to depictions we have seen earlier (for example, in a UML component diagram or Darwin). Wright's distinguishing feature is the CSP-based formal specifications of component/connector interfaces and behavior. These specifications can be specified in a notation that resembles mathematical equations, or an equivalent (if less decorative) ASCII notation. Lunar Lander, specified in Wright, might look like that in Figure 6-20.

Figure 6-20.
Lunar Lander
modeled in
Wright. Some
behavioral
specifications are
omitted for
simplicity but the
Call connector is
fully specified.

```

Component DataStore
  Port getValues (behavior specification)
  Port storeValues (behavior specification)
  Computation (behavior specification)

Component Calculation
  Port getValues (behavior specification)
  Port storeValues (behavior specification)
  Port calculate (behavior specification)
  Computation (behavior specification)

Component UserInterface
  Port getValues (behavior specification)
  Port calculate (behavior specification)
  Computation (behavior specification)

Connector Call
  Role Caller = call → return → Caller[] $
  Role Callee = call → return → Callee[] $
  Caller.call → Callee.call → Glue
  Glue = []Callee.return → Caller.return → Glue
  []$

Configuration LunarLander
  Instances
    DS : DataStore
    C : Calculation
    UI : UserInterface
    CtoUIgetValues, CtoUIstoreValues, UItoC, UItoDS : Call

  Attachments
    C.getValues as CtoUIgetValues.Caller
    DS.getValues as CtoUIgetValues.Callee

    C.storeValues as CtoUIstoreValues.Caller
    DS.storeValues as CtoUIstoreValues.Callee

    UI.calculate as UItoC.Caller
    C.calculate as UItoC.Callee

    UI.getValues as UItoDS.Caller
    DS.getValues as UItoDS.Callee
End LunarLander.

```

The main thing to notice about this specification is the detailed set of formal specifications of behavior. The CSP formalism, while semantically sound and well-documented, is somewhat arcane and has a steep learning curve that is quite different from learning, for example, a new programming language. The value of these specifications is that properties such as freedom from deadlock can be analyzed, which are difficult or impossible to detect in systems implemented in traditional programming languages.

Takeaways. Wright's formal specifications have an extremely high learning curve and cognitive overhead even for the simplest of systems. The analysis capabilities are powerful, but limited to a small set of properties (such as freedom from deadlock). However, the

extensive formal modeling might be worth the effort in, for example, safety-critical systems. Support for refining architectural specifications into implementations is lacking. Like other early ADLs, Wright is not actively used and supported today.

6.5.3 Domain- and Style-Specific ADLs

The ADLs surveyed above are suitable for describing a wide variety of software systems in many domains and architectural styles. However, some ADLs are domain-specific or style-specific, or at least optimized for describing architectures in a particular domain or style.

Domain- and style-specific ADLs are important for several reasons. First, their scope is better tailored to stakeholder needs since they target a particular group of stakeholders, rather than software and systems developers in general. Second, they are able to leave out unnecessary details and excessively verbose constructs because there is little need for genericity. Assumptions about the domain or style can be directly encoded into the ADL's semantics instead of being repeated in every model. For example, if a particular style mandates the use of a single kind of connector between each pair of linked components, there is no need to include the notion of a connector in the ADL—the ADL's users and tool set can just assume that such connectors exist on each link implicitly.

Examples of domain- and style-specific ADLs include Koala, Weaves, and AADL. We have discussed Koala previously; it is an ADL used to model families of consumer electronics devices. Weaves is both an architectural style and an associated ADL for modeling systems composed of components that interact through streams of objects. AADL is an industrial ADL tailored for modeling embedded, real-time systems (both hardware and software).

Koala

Koala (van Ommering et al. 2000) was developed by Philips Electronics to capture the architecture of consumer electronics devices such as televisions, VCRs, and DVD players. It is, to a large extent, a domain-specific ADL—it was developed for the specific use of one company for modeling software in a single domain to address specific issues within that domain. The software systems that Koala models are composed of interconnected software components that communicate via explicit provided and required interfaces.

Semantically and syntactically, Koala is a descendant of Darwin. It uses Darwin's structural concepts of input and output ports, but expands on them through the addition of constructs to support product-line architectures. Product lines were introduced earlier, in Chapter 1. Product lines are prevalent in the consumer electronics domain, where a single product like a television may have a host of feature variations: diagonal size, audio/video inputs and outputs, and optional components such as integrated VCRs, DVD players, or both. Separately documenting the software architectures of each of these products would be redundant and error-prone, since substantially the same elements would be repeated over and over again in each product. Koala addresses this by having specific constructs in the language for explicitly defining points of variation. In this way, multiple products can be described with a single model, with differences between the products encoded as variation points.

The product-line concepts in Koala will be discussed in detail later in this book, specifically in Chapter 15. We defer the discussion of the specifics of the notation until

then; interested readers should skip ahead. The evaluation rubric for Koala is presented here for completeness.

Evaluation Rubric for Koala	Scope and Purpose	Capturing the structure, configuration, and interfaces of components in the domain of embedded consumer electronics devices.
	Basic Elements	Components, interfaces, and constructs for specifying explicit points of variation: diversity interfaces, switches, and multiplexers.
	Style	Koala descriptions capture the architectures of related, variant products simultaneously, and this might be seen as defining a narrow architectural style.
	Static and Dynamic Aspects	Only static structure and interfaces are modeled.
	Dynamic Modeling	Although points of variation are defined statically in the architecture, the selection of variants could change at run time.
	Non-Functional Aspects	Not explicitly modeled.
	Ambiguity	Koala elements are concrete and closely mapped to implementations.
	Accuracy	Koala models have well-defined rules and patterns of interconnection. Koala elements are also closely mapped to implementations. Errors in models should be relatively easy to identify by looking for pattern violations or implementation problems such as compiler errors.
	Precision	The configuration of a system is well defined in a Koala model, but other aspects of a system are not specified.
	Viewpoints	Structural viewpoint with explicit points of variation.
	View Consistency	In general, Koala specifications do not include multiple views.

Weaves

Weaves (Gorlick and Razouk 1991) is both an architectural style and an accompanying notation. Weaves is used to model systems of communicating small-grain “tool fragments” that process objects of data. Weaves can be seen as a variant of the pipe-and-filter style with three significant differences. First, Weaves tool fragments process object streams instead of pipe-and-filter’s byte streams. Second, Weaves connectors are explicitly sized object queues, whereas pipe-and-filter connectors are implicit pipes. Weaves connectors serve to facilitate the transfer of both data and control among components. They receive objects from input ports, return control to the calling tool fragment, and then pass the object to tool fragments on output ports in a separate thread of control. Third, Weaves tools can have multiple inputs and outputs, whereas pipe-and-filter components have one input and one output.

Figure 6-21 shows a basic architecture expressed in the Weaves notation. The component Tool Fragment 1 outputs a stream of objects to an explicit queue connector Q1, which forks the stream and forwards the objects to both Tool Fragment 2 and Tool Fragment 3. As is obvious from this diagram, the Weaves notation is graphical and minimalist: Components are represented by shadowed boxes and queue connectors are represented by plain boxes; configurations are expressed using directed arrows connecting components and connectors.

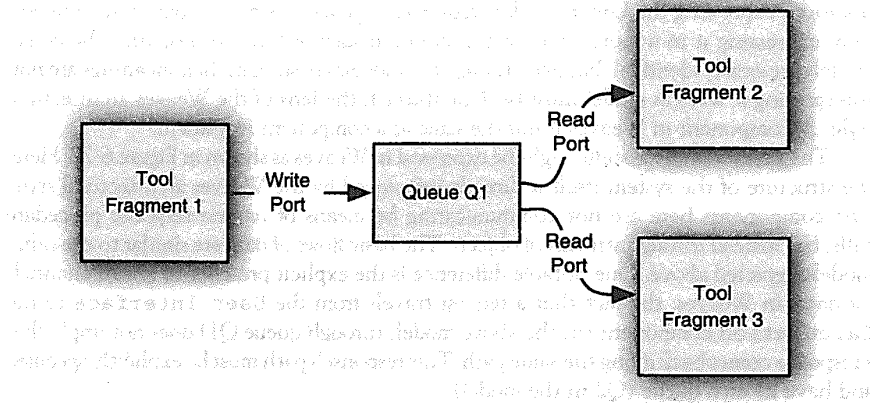


Figure 6-21.
A basic weave in
the Weaves
notation.

Although the Weaves notation is minimal, it is adequate to serve as a description notation for the Weaves style. Here, additional complexity is unnecessary: Weaves' extreme simplicity and straightforwardness makes it easy to understand and interpret, and the semantic interpretations of the few elements are provided by the style itself.

Scope and Purpose	The structure and configuration of components in architectures that conform to the Weaves architectural style.	Evaluation Rubric for Weaves
Basic Elements	Components, connectors (queues), and directed interconnections.	
Style	Weaves models conform to the Weaves architectural style; the constraints of that particular style are implicit in the model.	
Static and Dynamic Aspects	Only static structure is modeled.	
Dynamic Modeling	Although there is no direct support for dynamic modeling, the Weaves style establishes a close correspondence between components in the architectural model and components in the implementation; changes tracked in one can be straightforwardly applied to the other.	
Non-Functional Aspects	The Weaves style induces certain non-functional properties on systems but these are not explicitly modeled.	
Ambiguity	The meaning of Weaves elements is well defined and not ambiguous.	
Accuracy	Errors in models are limited to broken links and interconnection problems, which are easy to identify. It should be easily possible to determine whether implementations correspond to Weaves models since there is a close correspondence established by the style.	
Precision	The configuration of components and connectors in a system is well defined in a Weaves model, but other aspects of a system are not specified.	
Viewpoints	Structural viewpoint.	
View Consistency	In general, Weaves specifications do not include multiple views.	

Lunar Lander in Weaves. Because Weaves is both a style and an architecture description notation, expressing the Lunar Lander architecture in Weaves means something different from expressing it in a more style-neutral notation such as UML or Darwin. The actual models are nearly identical, but even though the models are similar, their meanings are not. Interpreting a Weaves model must be done through the lens of the Weaves architectural style: A component in Weaves is not the same as a component in Darwin.

The Lunar Lander system might be expressed in Weaves as shown in Figure 6-22. Here, the structure of the system itself is directly influenced by the Weaves architectural style. The components here are not communicating by means of request-response procedure calls, but instead through streams of objects. The basic flows of data are similar to the other models depicted above. One notable difference is the explicit presence of return channels for data: in Weaves, the fact that a request travels from the *User Interface* to the *Calculation* component (in the above model, through queue Q1) does not imply that a response comes back along the same path. This response's path must be explicitly specified and have its own queue (Q2 in the model).

Figure 6-22.
Model of the
Lunar Lander
application in
Weaves.

