

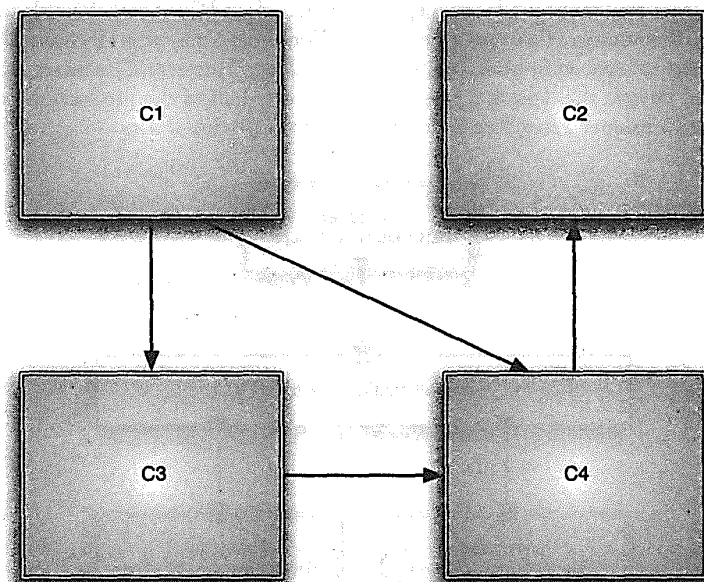
This problem may be mitigated if the Multicast Connector servicing the three components is able to buffer data, but clearly, if the system is long running, the connector's buffer will eventually overflow. The connector will also have to include additional processing logic to store the data temporarily and also route it to the two components in the order received, which will introduce additional overhead in the system. If the two Consumer components require the data to be in different formats from that generated by the Producer, the connector will also have to act as an adapter, introducing further overhead. Likewise, the connector may have to perform additional tasks based on the architectural requirements, such as encrypting the data for security, ensuring that the data is delivered according to the specified routing policy (for example, best-effort or exactly-once), or compressing the data if necessary.

8.2.4 Architectures at Different Abstraction Levels

During the architectural design process, architects frequently address the critical system requirements first, and then both introduce additional elements into the architecture and refine the architecture to include additional details that are necessary for the architecture's realization into the final system. This may involve the addition and further breakdown of architectural elements, as well as the introduction and refinement of existing design decisions.

Consider a simple example. A high-level architectural breakdown of a system may look like the diagram shown in Figure 8-9. For simplicity, the connectors are depicted only as arrows indicating the interacting components as well as the direction of interaction. For example, component C1 initiates the interactions with components C3 and C4. As is the

Figure 8-9.
A high-level
architectural
configuration.



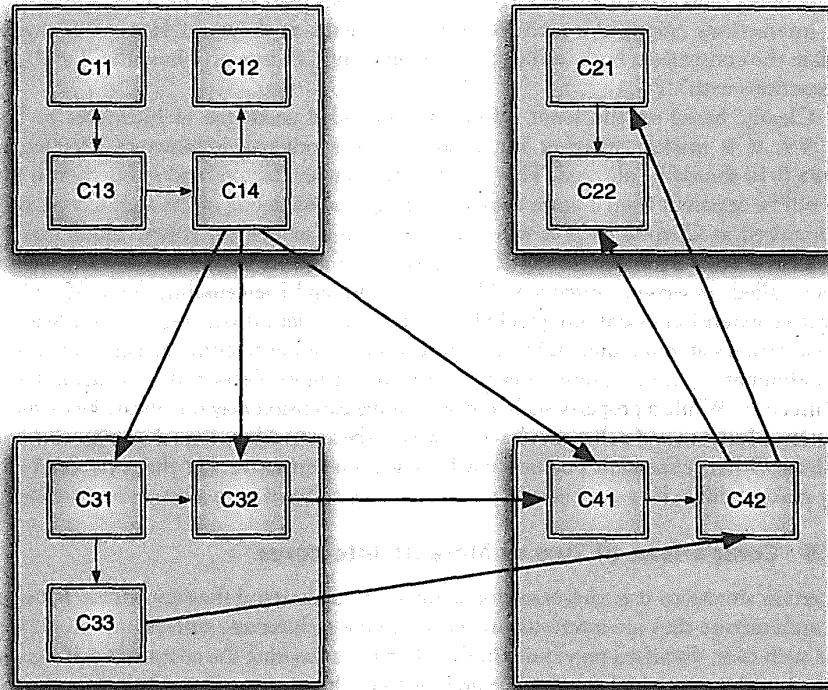


Figure 8-10.
A refined, more
detailed
architectural
configuration.

case with all boxes-and-arrows architectural descriptions, many details of this architecture are missing, including the components' interfaces, details of their behaviors and of the data exchanged, as well as the semantics of their interactions, such as that involving components C1, C2, and C3. While important, such information is not critical to this discussion.

Figure 8-10 depicts another architectural configuration, which is intended to be a refinement of that shown in Figure 8-9. For convenience, groups of components in the refined architecture which are presumed to comprise the original components are highlighted. For example, components C11, C12, C13, and C14 along with their interconnections in Figure 8-10 comprise the component C1 from Figure 8-9. Furthermore, the connectors are refined, such that it is clear that only C1's subcomponent C14 is engaged in interactions with the subcomponents of C3 and C4.

In the process of refinement, several problems can be introduced, as discussed above in the context of refinement incompatibility. The easiest cases to address are those where an architectural design decision has been clearly invalidated. However, even the addition or modification of architectural decisions may cause a refinement inconsistency depending on the context and on the range of architectural changes that are allowed. For example, the architecture depicted in Figure 8-9 and Figure 8-10 may need to abide by an architectural constraint stating that interactions crossing the boundaries of the system's original four

components must have a single target and a single destination. In that case, elaborating the interactions among the architecture's components as shown in Figure 8-10 would violate the constraints, both in the case of component C1's interactions with C3 and C4's interactions with C2.

Clearly, based on the scant information provided about the architectures in Figure 8-9, it is unclear whether the changes to the original architecture depicted in Figure 8-10 should be allowed. This will depend on many factors, and some, though not all, will be apparent from a more detailed and rigorous modeling of the architecture such as discussed in Chapter 6. This will also depend on a refinement policy, which may be explicitly stated. Thus, for example, Mark Moriconi and colleagues define a refinement policy called *conservative extension* (Moriconi, Qian, and Riemenschneider 1995). Conservative extension essentially precludes an architect from introducing any new features into a system's architecture, and instead restricts him or her to either elaborating or possibly eliminating those features already existing in the more abstract, that is, higher-level, architecture. While a property such as conservative extension may not always be useful in practice, Moriconi and colleagues have shown it to be amenable to formal specification and analysis. Other refinement policies may be much less restrictive, but that very flexibility may make it difficult to establish that they are being adhered to in a concrete situation.

8.2.5 Comparison of Two or More Architectures

In certain situations it is important for architects to understand the relationship between the architecture they are interested in and a baseline architecture with known properties. One such case, discussed previously in this chapter, is ensuring the compliance of a given system's architecture with a reference architecture. Other cases may include ensuring the architecture's compliance with the design guidelines captured in an architectural style or with a particular architectural pattern. It is, of course, also possible that the baseline architecture is simply one that the architect knows and understands. The architect may have encountered the architecture in literature, it may be the architecture of a related product he or she and colleagues had developed previously, or it could even be an architecture that has been recovered from a system (possibly even competitor's system) that exhibits desired properties.

Such comparisons of two or more architectures involve comparing the processing and data storage capabilities provided by the components, the interactions as embodied in the connectors, the characteristics of the data exchange, the components' and connectors' compositions into the system's configuration, and the sources of the non-functional properties exhibited by the system. It is possible for the architectures in question to be at different levels of abstraction, in which case the techniques discussed in the previous subsection can be employed.

8.3 ARCHITECTURAL CONCERN BEING ANALYZED

Architectural analysis techniques are directed at different facets of a given architecture. Some techniques strive to ensure primarily the architecture's *structural* properties; others focus on the *behaviors* provided by the architectural elements and their composition; yet others may analyze whether the *interactions* among the architectural elements adhere to

certain requirements and constraints; finally, the *non-functional properties* exhibited by the architecture are frequently considered to be very important and are thus studied carefully.

In practice, a given analysis technique, or suite of techniques, will address more than one architectural concern at a time. In this section, we focus on each concern individually for ease of exposition and also discuss the relevant characteristics of each.

Structural Characteristics

The structural characteristics of a software architecture include concerns such as the connectivity among an architecture's components and connectors, containment of lower-level architectural elements into composite higher-level elements, possible points of network distribution, and a given system's potential deployment architectures. These concerns can help to determine whether the architecture is well formed. Examples include components or subsystems that are disconnected from the rest of the architecture, missing pathways between components and connectors that are intended to interact, existing pathways between components and connectors that are not wanted, encapsulation of components or connectors that must be visible and accessible at a higher level of hierarchical composition, and so on. Structural analysis can also establish adherence to architectural constraints, patterns, and styles. Structural concerns can also help in the analysis of different aspects of system concurrency and distribution, as they tie the system's software elements and their properties with the hardware platforms on which they will execute.

Behavioral Characteristics

A well-structured architecture is of limited utility if the individual components do not provide the behaviors that are expected of them and, further, if they do not combine to provide the expected system-level behaviors. Therefore, analyzing an architecture's behavioral characteristics has two related facets:

1. Considering the internal behaviors of individual components.
2. Considering the architectural structure to assess composite behaviors.

It is possible, especially in systems composed with third-party components obtained off the shelf, that an architect's insight into the internal workings of different system components will be restricted to the components' public interfaces. As indicated earlier, particularly in Section 8.1, the types of behavioral properties that can be inferred at the level of interfaces are quite limited, and many potential problems with the architecture may remain undetected.

Interaction Characteristics

The relevant characteristics of interactions in a given architecture may include the numbers and types of distinct software connectors, and their values for different connector dimensions (recall Chapter 5). Interaction characteristics can help to establish whether the architecture will actually be able to fulfill some of its requirements. For instance, a non-buffering connector in the example from Figure 8-8 would result in a system in which one of the components received at most one-half of all the data.

Analysis of interaction characteristics may also encompass the interaction protocols for different system components (for example, recall Figure 8-3) and internal behaviors specified for different system connectors (for example, recall Figure 8-7). Such details would

aid in the analysis of finer-grain interaction characteristics, such as whether a component interacting through an otherwise appropriate connector will be legally accessed or whether a set of interacting components may deadlock.

Non-Functional Characteristics

Non-functional characteristics form a critical dimension of almost all software systems. These characteristics typically cut across multiple components and connectors, which makes them particularly difficult to assess. Furthermore, non-functional characteristics are often not properly understood, they are qualitative in nature, and their definitions are partial or informal. Therefore, while the non-functional characteristics present an important and formidable challenge to software architects, architectural analysis techniques focusing on these characteristics are scarce.

8.4 LEVEL OF FORMALITY OF ARCHITECTURAL MODELS

The relationship between architectural models and analysis is symbiotic: What the system's stakeholders want to be able to analyze will influence what software architects capture in their architectural models. Conversely, what architects capture in the architectural models directly determines what they will be able to analyze and what analysis methods they will use to do so.

Architectural models have been discussed in detail in the preceding chapters, and particularly in Chapter 6. Here, we will look specifically at the role they play in the context of architectural analysis. For that purpose, architectural models can be classified as informal, semiformal, and formal.

Informal Models

Informal models are typically captured in boxes-and-lines diagrams such as shown in Figure 8-1. Informal models can provide a useful high-level picture of the system. They are amenable to informal and manual analyses, typically by a broad section of stakeholders, including nontechnical stakeholders such as managers and system customers. For example, system managers can use them to determine a project's overall staffing needs. At the same time, informal models should be approached cautiously because of their inherent ambiguity and lack of detail.

Semiformal Models

Most architectural models used in practice are semiformal. A notation that strives to be useful to a large number of system stakeholders, both technical and nontechnical, will typically try to strike a balance between a high degree of precision and formality on the one hand, and expressiveness and understandability on the other. One widely used example is the Unified Modeling Language (UML). Semiformal languages such as the UML are amenable both to manual and automated analysis. Their partial imprecision makes it difficult to perform some more sophisticated analyses, for which formal models are needed.

Formal Models

While semiformal modeling notations typically only have a formally defined syntax, formal notations also have formally defined semantics. An example formal notation is Wright, which was used to specify the Pipe connector in Figure 8-7. Formal models are inherently amenable to formal, automated analysis and are typically intended for the system's technical stakeholders. At the same time, producing complete architectural models using a formal notation can be painstaking. Furthermore, formal models have been frequently shown in practice to suffer from scalability problems.

8.5 TYPE OF ANALYSIS

One useful categorization of architectural analysis techniques is into static, dynamic, or scenario-based techniques. We discuss the three categories and the role architectural models play in each.

Static Analysis

Static analysis involves inferring the properties of a software system from one or more of its models, without actually executing those models. A simple example of static analysis is syntactic analysis: Determining if the system model adheres to the syntactic rules of the modeling notation, whether it be an architectural description language, design diagramming notation, or programming language. Static analysis can be automated (for example, compilation) or manual (for example, inspection). All architectural modeling notations, including the informal boxes-and-lines diagrams, are amenable to static analysis, although the more formal and expressive notations can be harnessed to provide more precise and sophisticated answers. Formal notations used in modeling software systems include:

- *Axiomatic notations*, which model systems via logical assertions; an example is Anna (Luckham and Henke 1985).
- *Algebraic notations*, which model systems via collections of equivalence relations; an example is LARCH (Guttag, Horning, and Wing 1985).
- *Temporal logic notations*, which model systems in terms of order of execution and timing; an example is GIL (Dillon et al. 1992).

It should be noted that the above example notations cannot be considered as architecture description languages, or ADLs, since by themselves they do not provide any explicit architecture modeling constructs. However, it is possible to use them as a basis of an ADL, much in the same way that Wright leverages communicating sequential processes (CSP) as discussed in Chapter 6.

Dynamic Analysis

Dynamic analysis involves actually executing or simulating the execution of a model of the software system. In order to perform dynamic analysis on an architectural model, its semantic underpinning must be executable or amenable to simulation. State-transition diagrams are an example executable formalism with which the reader should be familiar.

Other executable formalisms include discrete events, queuing networks (Lazowska et al. 1984), and Petri nets.

Scenario-Based Analysis

For large and complex software systems, it is often infeasible to assert a given property for the entire system over the entire space of its possible states or executions. For such systems, specific-use cases are identified that represent the most important or most frequently occurring system usage scenarios, and the analysis is focused on those. Scenario-based analysis can be an instance of both static analysis—as a tool for reducing a modeled system's state space, as discussed later in this chapter—and dynamic analysis—as a tool for reducing the system's execution space. At the same time, scenario-based analysis requires that architects be very careful about the inferences they make from their inherently limited evidence.

8.6 LEVEL OF AUTOMATION

Different architectural analysis techniques are amenable to different levels of automation. The level of automation depends on several factors, including the formality and completeness of the architectural model and the property being assessed. In general, an architectural model provided in a more formal notation will be more amenable to automated analysis than a model provided in a more informal notation. Likewise, a model that captures a greater number of the architectural design decisions for the given system will be more amenable to rigorous, automated analysis than a model that is missing many such design decisions. Finally, a well-understood property that is quantifiable and can itself be defined formally will be easier to assess automatically than a qualitative property that may not be as well understood. This last point is particularly important in software engineering: As will be demonstrated in Chapter 12, many non-functional properties, which are critical to the success of most all software systems, are understood at the level of intuition, anecdote, and informal guideline.

Manual

Manual analysis of software architectures requires significant human involvement, and is thus expensive. However, manual analysis can be performed on models of varying levels of detail, rigor, formality, and completeness. It has the added advantage that architectural rationale, which is often tacit, can be taken into account. This type of analysis may also be required when multiple, potentially clashing properties must be ensured in tandem.

A number of architectural analysis techniques fall in this category. These are inspection-based techniques and will be further elaborated upon later in this chapter. One well-known example is the architecture trade-off analysis method, or ATAM (Clements, Kazman, and Klein 2002). The analysis results emerging from manual analysis are typically qualitative. Since it is not always possible to quantify important properties of a software system—such as scalability, adaptability, or heterogeneity—any analysis of the extent to which the system exhibits those properties will not be quantifiable. The analysis results are also frequently qualified by a particular context in which a system may exhibit a given property. Scenario-based techniques fall in this category.

Given the human-intensive nature of this category of architecture analysis techniques, a critical concern must be to make the analysis reliable and repeatable. Since the architectural models as well as the properties of interest may be less than formally captured, the focus of many manual analysis techniques has been on specifying a detailed process that must be followed by the system architects and other stakeholders participating in the analysis.

Partially Automated

Growing levels of rigor in architectural models, and in understanding of the software systems' key properties, present opportunities for automating different facets of architectural analysis. In fact, most architectural analyses can be at least partially automated, involving both software tools and human intervention. In that sense, architectural analysis techniques can be thought of as covering a spectrum of automation, with manual and fully automated analysis being the ends of that spectrum.

Most architecture modeling notations presented in Chapter 6 are amenable to ensuring a given architectural description's syntactic correctness, as well as different degrees of semantic correctness. For example, an xADL model can be analyzed for style-specific component interconnectivity rules, while Wright allows one to analyze a given composition of components communicating through a connector for deadlocks. At the same time, neither model can be analyzed automatically for other properties, such as reliability, availability, dependability, or latency. This is at least in part because system parameters that are relevant to assessing these properties are not captured—to the necessary degree, or at all—by these architectural modeling notations.

Fully Automated

It can be argued that the specific analyses mentioned above, such as ensuring the syntactic correctness or deadlock freedom in an architectural description, can be considered fully automatable since it is possible to complete them without human involvement. At the same time, the results of automated analyses are typically partial: The fact that an architectural description provided in a given ADL fully adheres to that ADL's syntax, or that a partial system description is deadlock free, still leaves a large number of questions about the respective models unanswered. This means that, in practice, fully automated architectural analysis techniques must be combined with other techniques, which themselves may need human intervention, in order to get more comprehensive answers.

8.7 SYSTEM STAKEHOLDERS

The stakeholders in a software project will often have different objectives. For example, customers may be interested in getting the most functionality as quickly as possible, for the lowest amount of money possible. A project manager may be interested in ensuring that the project is staffed appropriately and that the rate of expenditure does not exceed some target. The architects' primary objective may be to deliver a technically sound system that will be easily adaptable in the future. Finally, a developer may be interested primarily in ensuring that the modules he or she is responsible for are implemented on time and bug-free. Therefore, the different stakeholders will not necessarily have identical architecture

analysis needs. The remainder of this section highlights the role architectural analysis plays in the case of each stakeholder type.

Architects

Software architects must take a global view of the architecture and are interested in establishing all four Cs in the architecture: completeness, consistency, compatibility, and correctness. Depending on the project's context and objectives, architects may need to rely on all types of architectural models at all levels of scope and formality. While they may prefer to use automated analysis techniques, architects will frequently have to rely on manual and semi-automated techniques.

Developers

Software developers often take a more limited view of the architecture—namely, the modules or subsystems for which they are directly responsible. As such, developers are interested primarily in establishing the consistency of their modules with other parts of the system with which these modules will interact, as well as compatibility with the required architectural styles, reference architectures, and standards. They need not worry about the architecture's completeness, and can at best assess its partial correctness. The models that developers will likely find most useful are formal, with all necessary details specified and ready for implementation. However, these likely would be models of individual elements for which a given developer is directly responsible, rather than models of the entire architecture.

Managers

Project managers are typically primarily interested in an architecture's completeness—Are all the requirements satisfied?—and correctness—Are the requirements appropriately realized in the architecture and thus eventually will be realized in the implementation? Managers may also be interested in the architecture's compatibility if the architecture and eventually the implemented system must adhere to a reference architecture or a set of standards.

Consistency is a system's internal property, and managers typically do not concern themselves with it, that is, they naturally delegate such responsibilities to architects and developers. There are exceptions, however. For example, architectural defects may become a major issue that begins affecting the project's schedule or budget. Alternatively, customers or project contracts may explicitly mandate certain consistency properties, in which case managers would need to explicitly consider them.

The types of architectural models that are useful to managers are usually less formal models of the entire system. A manager's focus will frequently be on cross-cutting non-functional system properties, as well as the system's structural and dynamic characteristics.

Customers

Customers are interested primarily in the commissioned system's completeness and correctness. Their concerns can be summarized with two key questions:

1. Is the development organization building the right system?
2. Is the development organization building the system right?

A customer may also be interested in the system's compatibility with certain standards, and possibly reference architectures in which the customer has a vested interest. Consistency is not of critical importance unless it is reflected in externally visible system defects.

In terms of architectural models, customers typically favor understandability over formality of models. They are interested in overall models (the "big picture") and the system's key properties. They are often interested in scenario-driven assessment of a system's structural, behavioral, and dynamic characteristics.

Vendors

Software vendors typically sell technology, such as individual components and connectors, rather than architecture. As such they are interested primarily in composability of those components and connectors as well as their compatibility with certain standards and widely used reference architectures. Like a given system's customers, vendors may value the understandability of architectural models, but their customers are software developers who may demand formal models of the software they purchase from the vendor. The vendors' primary focus is on the analysis of the individual elements and their properties. The structural characteristics of the overall architecture are not as important, although dynamic characteristics may be since they may have implications on the composability of the individual elements in future systems.

8.8 ANALYSIS TECHNIQUES

A large number of analysis techniques are available to software architects. Some of them are variations on techniques applied to other software development artifacts—primarily formal specifications and code—while others have been developed specifically with software architectures in mind. In this section we discuss a cross-section of architectural analysis techniques. Although it is not intended to provide a complete overview of existing techniques, the cross-section is broadly representative.

We divide architectural analysis techniques into three categories:

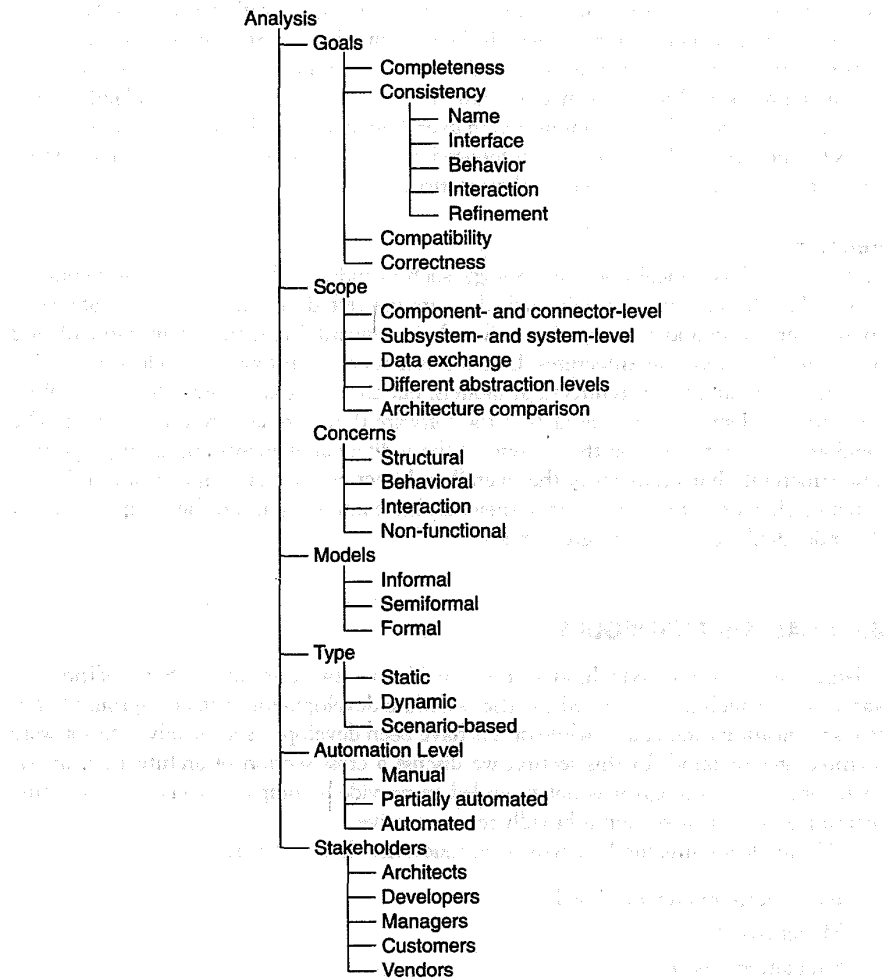
- Inspection- and review-based.
- Model-based.
- Simulation-based.

The discussion of the techniques within these categories will focus on the architectural analysis dimensions outlined above and summarized in Figure 8-11.

8.8.1 Inspections and Reviews

Software inspections and reviews are widely used code-analysis techniques. If unfamiliar with these techniques, the reader is encouraged to consult an introductory software engineering text. *Architectural* inspections and reviews are conducted by different stakeholders to ensure a variety of properties in an architecture. They involve a set of activities conducted by system stakeholders in which different architectural models are studied for specific properties. These activities often take place in architecture review boards, where several stakeholders define the objective of the analysis—such as ensuring that the architecture

Figure 8-11.
Architectural
analysis
dimensions
discussed in this
chapter.



satisfies a given non-functional property—and then, as a group, carefully study and critique the architecture or some of its parts.

Inspections and reviews are manual analysis techniques, and as such can be expensive. On the other hand, they have the advantage of being useful in the case of informal or partial architectural descriptions. They can also be employed effectively in the case of “soft” architectural properties, such as scalability or adaptability, which are not precisely understood and amenable to formal definition. Another advantage of inspections and reviews is that they can simultaneously take into account the objectives of multiple system stakeholders and consider multiple desired architectural properties.

Depending on the context, inspections and reviews can have any of the four architectural analysis goals: consistency, correctness, completeness, and compatibility. In terms of consistency, they will typically be well suited to name and interface consistency analysis. Behavior, interaction, and refinement consistency analysis may be conducted by the technical stakeholders—architects and developers—although doing so manually may be a difficult and error-prone task. For example, recall the interaction characteristics embodied in the single Wright Pipe connector from Figure 8-7, and even the comparatively simpler single state-transition interaction protocol of the QueueServer component from Figure 8-3. Dealing with a large number of such models manually would cognitively overload even the most capable architects. Thus, if analyzing for any of the three latter types of consistency is undertaken during an architectural inspection or review, it may be advisable to restrict the analysis to carefully confined subsets of the architecture.

The scope of inspections and reviews can vary. The stakeholders may be interested in individual components and connectors, or their compositions in a specific subsystem or the entire system. The stakeholders may also center on the data exchanged among the specific components and connectors or, globally, across the entire architecture. They may try to assess the compliance of the architecture to a higher-level architecture that served as its starting point. They may also try to assess the architecture's similarity to an existing architecture with known properties.

Similarly, the specific concern of the analysis can vary. The stakeholders may focus on the structural, behavioral, or interaction properties, although as mentioned above, the latter two may be difficult to assess manually. Inspections and reviews may be particularly well suited to establishing certain non-functional properties, especially those that require some interpretation and consensus reaching by the human stakeholders.

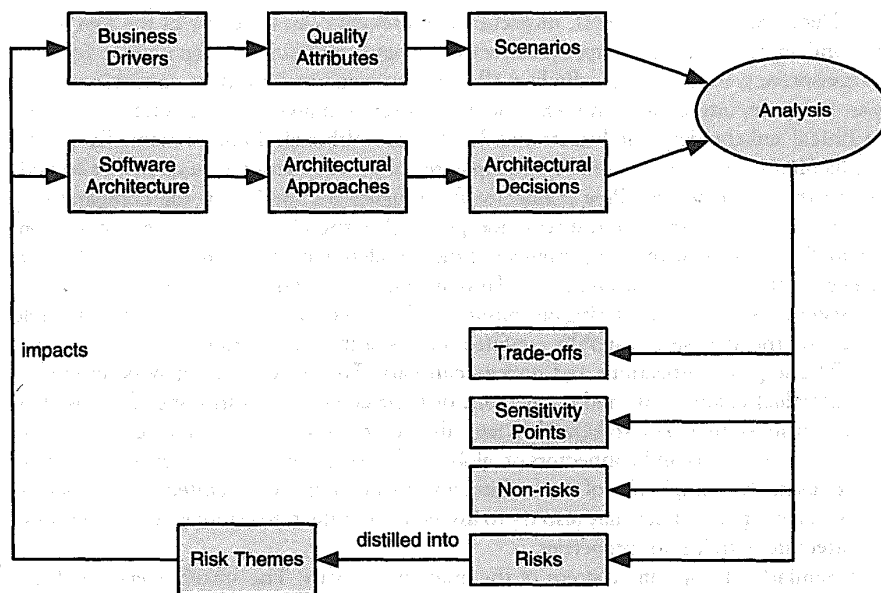
In terms of the types of models particularly amenable to inspections and reviews, any level of formality may be suitable in principle. However, highly formal models will not be useful to the nontechnical stakeholders, and even the technical stakeholders may find them difficult to read and understand. At the other end of the spectrum, informal models may be useful if, for example, the objective of the inspection is to develop a common understanding of the architecture's general characteristics. On the other hand, it may not be very meaningful to rely on informal models when inspecting the architecture for concrete properties of interest.

By their nature, the types of analysis for which inspections and reviews are geared are static and scenario-based. Since the stakeholders manually assess the architectural models, they have to focus on the architecture's static properties, such as proper connectivity, interface conformance between interacting components, adherence to desired architectural patterns, and so on. Furthermore, as discussed below in the case of the ATAM analysis technique, the stakeholders may manually run through some critical scenarios to ensure that the architecture will behave as expected.

As already mentioned, in terms of automation level inspections and reviews are manual and very human intensive.

Finally, all system stakeholders, save for perhaps component vendors, may participate in inspections and reviews. Architects and developers will conduct inspections and reviews most frequently, and will periodically be joined by project managers and possibly by customers.

Figure 8-12. The high-level view of the process steps followed in ATAM. The diagram has been adopted from Carnegie Mellon University's Software Engineering Institute.



ATAM

The Architectural Trade-Off Analysis Method (Clements, Kazman, and Klein 2002), or ATAM, developed at Carnegie Mellon's Software Engineering Institute, is a human-centric process for identifying risks in a software design early in the development life cycle. ATAM specifically focuses on the quality attributes, or non-functional properties (NFPs), of modifiability, security, performance, and reliability. Its objective is to reveal not only how well an architecture satisfies given quality goals, but also how those goals trade off against each other.

The ATAM process requires the gathering of the software architects designing the system, other important stakeholders of that system, and a separate independent architecture evaluation team. An evaluation using ATAM typically takes three to four days. The set of activities followed are depicted in Figure 8-12.

Two key inputs into the ATAM process are the *business drivers* and the system's *software architecture*. A project's decision maker, who is usually the project's manager or customer, first presents the system's major business drivers. These include:

- The system's critical functionality.
- Any technical, managerial, economic, or political constraints.
- The project's business goals and context.
- The major stakeholders.
- The principal *quality attribute* (NFP) goals that impact and shape the architecture.

The quality attributes become a basis of eliciting a set of representative scenarios that will help ensure the system's satisfaction of those attributes. There are three scenario categories in ATAM.

- *Use-case scenarios*, which describe how the system is envisioned by the stakeholders to be used.
- *Growth scenarios*, which describe planned and envisioned modifications to the architecture.
- *Exploratory scenarios*, which try to establish the limits of architecture's adaptability by postulating major changes to the system's functionality, operational profiles, and underlying execution platforms.

Once the scenarios are identified, they are prioritized in terms of importance by the system's stakeholders.

Another thread of activity in ATAM involves the project's architects presenting the key facets of the architecture. This includes:

- *Technical constraints*, such as the required hardware platforms, operating systems, middleware, programming languages, and off-the-shelf functionality.
- *Any other systems* with which the system under development must interact.
- *Architectural approaches* that have been used to meet the quality requirements.

An architectural approach in ATAM refers to any set of architectural design decisions made to solve the problem at hand. Architectural approaches are typically architectural patterns and styles. The architectural approaches are used to elaborate the architectural design decisions made for the system.

The key step in ATAM is the *analysis* of the architectural approaches in the context of the identified scenarios, with the primary goal of establishing the relationship between the approaches—that is, architectural design decisions—and quality attributes. The analysis is not rigorous, but rather is intended to observe general, coarse-grained characteristics of the architecture. To this end, for each architectural approach a set of analysis questions are formulated that are specific to the quality attributes and architectural approach under consideration. The system's architects engage with the ATAM evaluation team in answering these questions. The answers particularly focus on the architectural approach's known *risks* (weaknesses), *non-risks* (strengths), *sensitivity points*, and *quality trade-off points*.

Depending on the architects' responses, any of the answers may be used as a starting point for further analysis. For example, let us assume that an architect is unable to answer questions about a given subsystem's event processing priorities or about the overall system's deployment—that is, the allocation of software components to hardware nodes, further discussed in Chapter 10. In that case, there will be no point in investing further resources to perform rigorous model-based or simulation-based analyses such as the construction of queueing networks or rate-monotonic performance analysis.

Finally, the risks identified in the given iteration of ATAM are distilled into *risk themes* and are fed back to the ATAM inputs of software architecture and business driver. The objective is to repeat the above process until all major architectural risks are properly mitigated.

Regardless of its actual effectiveness, any inspection- and review-based architectural analysis process such as ATAM will sensitize a system's stakeholders to the important facets of their architecture. In ATAM's case, because of its focus and objectives, as well as the prolonged participation of system stakeholders, it has a high potential of resulting in clarified quality attribute requirements for the system, a better documented basis for architectural decisions, identification of risks early in the life cycle, and increased communication among stakeholders.

ATAM: Analysis Summary	Goals	Completeness Consistency Compatibility Correctness
	Scope	Subsystem- and system-level Data exchange
	Concern Models	Non-functional Informal Semiformal
	Type	Scenario-driven
	Automation Level	Manual
	Stakeholders	Architects Developers Managers Customers

8.8.2 Model-Based Analysis

Model-based architectural analysis techniques rely solely on a system's architectural description and manipulate that description to discover properties of the architecture. Model-based techniques involve analysis tools of different levels of sophistication. These tools are frequently guided by architects, who may have to interpret the intermediate analysis results and guide the tool in further analysis.

Because of their tool-driven nature, model-based techniques are much less human intensive, and hence usually less costly, than inspections and reviews. On the other hand, they can only be used to establish "hard" properties of a system's architecture, that is, those properties that can be encoded in the architectural model. They cannot easily account for implicit properties—those that a human might readily infer from the existing information and thus chooses not to model explicitly. Additionally, model-driven analysis techniques cannot typically assess "soft," but very important, aspects of an architecture, such as design intent and rationale. Model-based techniques usually also focus on a single, specific facet of a system's architecture, such as syntactic correctness, deadlock freedom, adherence to a given style, and so forth.

Another concern with model-driven analysis techniques is their scalability: More sophisticated techniques are required to keep track of a very large number of the modeled system's elements and properties. The usual trade-off encountered in many static analysis tools, of which model-based architectural analysis tools are an instance, is between scalability on the one hand and precision or confidence on the other. In other words, architects

may be able to arrive at highly precise analysis results with a high degree of confidence in those results for smaller systems, but would have to sacrifice that precision and confidence for larger systems. Because of all these reasons, the results of model-driven analyses are usually partial, and multiple such techniques are used in tandem in any given architecture-driven software development project. Even then, model-based analysis usually does not provide all the needed answers to an architect, and is coupled with techniques from the other two categories—inspections and reviews and simulation-based analysis.

The goals of model-based architectural analysis techniques are usually consistency, compatibility, and internal completeness. The goals can also include certain aspects of external completeness and correctness. Recall that both external completeness and correctness assess the adherence of a system's architectural model to the requirements, and of the system's implementation to the architectural model. Certain facets of external completeness and correctness, such as structural completeness or correctness, can be established from the architectural model. Furthermore, the output of the analysis technique may be automatic generation of (partial) system implementation from the architectural model, which would ensure external completeness and correctness by construction. If the analysis involves system requirements, then to some extent the requirements will need to be formalized.

The scope of model-based architectural analysis can span individual components and connectors, their compositions in a specific subsystem or the entire system, as well as the data exchanged among the specific components and connectors or, globally, across the entire architecture. Model-based techniques may also try to assess the compliance of the model to a higher-level model from which it has been derived. Analysis techniques in this category may also try to assess the architecture's similarity to an existing architecture with known properties. This presumes that both architectures are modeled in one of the notations supported by the analysis technique.

Similarly, the specific concern of model-based analysis can vary. The techniques may focus on the structural, behavioral, or interaction properties. Behavioral and interaction properties may be difficult to assess completely using model-based techniques alone, and these techniques are usually coupled with simulation-based approaches. Model-based techniques can be used to analyze an architecture's non-functional properties, but usually require the use of specific formalisms.

In terms of the types of models particularly amenable to this type of analysis, the general rule of thumb is that more formality yields more meaningful and precise results. Thus, for example, a Rapide specification, such as that shown in Figure 8-2, will be much more amenable to manipulation by an analysis tool than an informal diagram such as that shown in Figure 8-1. Usually the architectural models to which sophisticated analysis tools are applied have formally specified syntax as well as semantics.

By their nature, the type of analysis for which model-based techniques are well suited is static analysis. These techniques are well suited to assessing properties such as proper connectivity, type conformance, definition-use analysis of architectural services, interface and behavioral conformance between interacting components, structural adherence to desired architectural patterns, deadlock freedom, and so on.

As already mentioned, with regard to *automation level* model-based techniques are typically at least partially automated, and are often fully automated, requiring no human intervention.

Finally, model-driven architecture analysis techniques are usually targeted at the technical *stakeholders*, namely architects and developers. Other stakeholders may be interested in summaries of analysis results, but typically these techniques require a certain degree of mastery and provide low-level insights into the architecture.

The remainder of this section will survey the spectrum of model-based analysis techniques developed for different architecture description languages and discuss how architectural models can be leveraged to enable analysis of a representative non-functional property—reliability.

Model Checking

A particular, widely used model-based analysis technique in computer-based systems (software as well as hardware) is model checking. Model checking is a method for algorithmically verifying formal systems. This is achieved by verifying whether the model derived from a hardware or software design satisfies a formal specification expressed as a set of logic formulas. Several software model-checking techniques have emerged over the past decade; interested readers should consult a recent survey of this area by Dwyer and colleagues (Dwyer et al. 2007).

The model of a system is usually expressed as a finite state machine, in other words, a directed graph consisting of vertices and edges. A set of atomic propositions is associated with each vertex, that is, state. The edges represent possible executions that alter the system's state. Finally, the atomic propositions represent the properties that hold at the given point of execution.

The model-checking problem can be stated as follows: Given a desired property, expressed as a temporal logic formula p , and a model M with initial state s , decide if the model satisfies the logic formula, or formally

$$M, s \models p$$

If model M is finite, model checking is reduced to a graph search. Unfortunately, this is rarely the case with software systems, and the critical challenge faced by model-checking tools is *state explosion*, an exponential growth in the state space. Each model-checking technique must address state explosion in order to be able to solve real-world problems.

Researchers have developed several techniques that can help alleviate state explosion, including symbolic algorithms, partial-order reduction, binary decision diagrams, and abstraction of uninteresting or noncritical system characteristics. Another strategy for combating the exponential growth in the state space, successfully adopted by the Alloy technique (Jackson 2002), is to explicitly bound the size of the state space, thus ensuring the technique's feasibility. Of course, doing so introduces optimistic inaccuracy into the analysis process: While no defects may have been found during analysis, there is no guarantee that the model is in fact defect-free.

When using analysis tools such as model checkers, a software system's architects will have to trade off the risks of undiscovered critical defects against the practical limitations of applying a model checker on a very large architectural model.

Model-Based Analysis Enabled by ADLs

The types of analyses for which an ADL is well suited depend on its underlying semantic model and, to a lesser extent, its specification features. For example,

- Wright (Allen and Garlan 1997) uses communicating sequential processes, or CSP, to analyze a system's connectors and the components attached to them for deadlocks. The objective of the analysis is to identify any conditions under which a component

requires access to a currently unavailable system resource in order to continue its processing, but the system is unable to make that resource available. For example, if component C_i needs to access a record in a database, but that record is locked by another component C_j , and C_j is unable to release the lock until some chain of conditions ending with component C_i is met (for example, C_i may need to free up some other system resource so that C_j can complete its current task), then the system will deadlock.

- Aesop (Garlan, Allen, and Ockerbloom 1994) ensures style-specific topological constraints and type conformance among architectural elements. For example, Aesop will disallow a server to make requests in a client-server architecture.
- MetaH (Honeywell 1998) and UniCon (Shaw et al. 1995) support schedulability analysis by specifying non-functional properties, such as the criticality and priority of components. Then techniques such as rate-monotonic analysis (Liu and Layland 1973) are applied to ensure that the architecture as-modeled can, in fact, accomplish its required functionality. This is particularly important in the case of systems with stringent performance requirements, such as real-time requirements.

Language parsers and compilers are another kind of analysis tools. Parsers analyze architectures for syntactic correctness, while compilers establish semantic correctness. All architecture modeling notations used in practice have parsers. Several—for example, Darwin (Magee and Kramer 1996), MetaH, and UniCon—also have compilers of sorts, which enable them to generate executable systems from architectural descriptions, provided that component implementations already exist. Rapide's compiler generates executable simulations of Rapide architectures (Luckham et al. 1995).

Another aspect of analysis is enforcement of constraints. Parsers and compilers enforce constraints implicit in type information, non-functional attributes, component and connector interfaces, and semantic models. Rapide also supports explicit specification of other types of constraints, and provides means for their checking and enforcement. Its Constraint Checker analyzes the conformance of a Rapide simulation to the formal constraints defined in the architecture. An architecture constraint checking tool, Armani (Monroe 1998), which is based on the ACME ADL (Garlan, Monroe, and Wile 2000), allows specification and enforcement of arbitrary architectural constraints. UML allows the specification of architectural constraints in its Object Constraint Language, or OCL, but does not natively enforce those constraints; additional tool support must be obtained or built to do so.

Languages such as SADL (Moriconi, Qian, and Riemenschneider 1995) and Rapide provide support for refining an architecture across multiple levels of detail. SADL requires manual proofs of the correctness of mappings of constructs between an abstract and a more concrete architectural style. For example, an abstract architectural style can be the dataflow style, a more concrete style would be the pipe-and-filter style, and an even more concrete style would be the pipeline style. The proof in SADL is performed only once; thereafter, SADL provides a tool that automatically checks whether any two architectures described in the two styles adhere to the mapping. Rapide, on the other hand, supports event maps between individual architectures. The maps are compiled by Rapide's Simulator, so that its Constraint Checker can verify that the events generated during simulation of the concrete architecture satisfy the constraints in the abstract architecture.

A summary of the different ADLs' combined analysis foci is given in the table below.

ADLs: Analysis Summary	Goals	Consistency Compatibility Completeness (internal)
	Scope	Component- and connector-level Subsystem- and system-level Data exchange Different abstraction levels Architecture comparison
	Concern	Structural Behavioral Interaction Non-functional
	Models	Semiformal Formal
	Type	Static
	Automation Level	Partially automated Automated
	Stakeholders	Architects Developers Managers Customers

Reliability Analysis

A software system's reliability is the probability that the system will perform its intended functionality under specified design limits, without failure. A *failure* is the occurrence of an incorrect output as a result of an input value that is received, with respect to the specification. An *error* is a mental mistake made by the designer or programmer. A *fault* or a *defect* is the manifestation of that error in the system. In other words, a defect is an abnormal condition that may cause a reduction in, or loss of, the capability of a component to perform a required function; it is a requirements, design, or implementation flaw or deviation from a desired or intended state.

Failure in the above definition denotes that a particular system component is not operational. Reliability can be assessed using several metrics, such as:

- *Time to failure*, for example, mean time until a system fails after its last restoration.
- *Time to repair*, for example, mean time until a system is repaired after its last failure.
- *Time between failures*, for example, mean time between two system failures.

Modeling, estimating, and analyzing software reliability has been an active discipline for several decades. During most of that time, techniques for reliability analysis have been developed and applied at the level of system implementation artifacts. The reason is that several key system parameters are known during or after implementation, including the system's operational profile and possibly its failure and recovery history. Those parameters typically are used in the creation of a state-based model of the system, in which the probabilities of transitions between the states are derived from the system's operational profile

and failure data. Then, this model is fed into a stochastic model such as a discrete-time Markov chain, or DTMC (Stewart 1994), which is solved using standard techniques to determine the reliability of the system in question.

Engineers need not, and should not, wait to estimate the reliability of a system until the system has been implemented. A software architectural model can be analyzed for reliability in a manner similar to that described above. At the same time, there are several sources of uncertainty inherent in an architectural model that must be addressed:

1. Software developers may work within different development scenarios. Examples of development scenarios may include implementing entirely new systems, reusing components and/or architectures from previous projects, purchasing software from a vendor, and so forth. Each scenario introduces different reliability challenges.
2. The granularity of the architectural models may vary significantly. A system may be accompanied by coarse-grained models for very large components, partial models for commercial off-the-shelf components, very detailed models for safety-critical components, and so forth.
3. Finally, different sources of information about the system's likely usage may be available. Developers may have at their disposal little or no information about an unprecedented system, a functionally similar system whose usage will also likely be similar, access to experts or extensive domain knowledge, and so forth.

These types of variation must be accounted for in architecture-level reliability modeling and analysis. As in the case of implemented systems, a state-based model corresponding to the architecture can be constructed. However, certain information such as the operational profile and system failure frequencies cannot be obtained, but must be estimated. For this reason, the reliability values obtained at the level of architecture should not be treated as absolute values, but should instead be qualified by the assumptions, such as the presumed operational profile, made about the system. Furthermore, the uncertainties inherent in the architecture-level reliability model suggest that it may be more meaningful to leverage stochastic models other than DTMCs. One such model, specifically intended to deal with modeling uncertainties, is the hidden Markov model, or HMM (Rabiner 1989).

Goals	Consistency Compatibility Correctness	Facets of Reliability Analysis
Scope	Component-and connector-level Subsystem- and system-level	
Concern	Non-functional	
Models	Formal	
Type	Static Scenario-based	
Automation Level	Partially automated	
Stakeholders	Architects Managers Customers Vendors	

8.8.3 Simulation-Based Analysis

Simulation requires producing a dynamic, executable model of a given system, or of a part of the system that is of particular interest, possibly from a source model that is otherwise not executable. For example, the model of QueueServer component's behavior from Section 8.1, specified in terms of its operations' pre- and post-conditions, is not executable. On the other hand, its interaction protocol from Figure 8-3 can be simulated by selecting a possible sequence of component invocations, which can also be referred to as a sequence of system events. This event sequence would be used to execute QueueServer's interaction protocol state machine. For instance, if we assume that the Empty state is the start state in the state machine, the following will be valid sequences of events

```
<enqueue, enqueue>
<enqueue, dequeue, enqueue>
<enqueue, enqueue, dequeue, enqueue, dequeue>
```

while the single-event sequence

```
<dequeue>
```

is invalid, as is any sequence that starts with the dequeue event or has more dequeue than enqueue events. Note that the validity of a sequence such as

```
<enqueue, enqueue, enqueue>
```

cannot be established without introducing additional information about the system, namely, the size of the queue.

Simulation need not produce identical results to the system's execution: The source model, such as an architectural model, may elide many details of the system. For example, the QueueServer component's model says nothing about the frequency with which the component will be invoked, the sizes of elements that will be placed on the queue, the processing time required to store or access and return the queue elements, and so on. Because of this, the output of simulation might be observed only for event sequences, general trends, or ranges of values rather than specific results. Note that a system's implementation can be thought of as a very faithful executable model. Likewise, running that implementation model can be thought of as a highly precise simulation.

Clearly not all architectural models will be amenable to simulation—recall as a simple example the informal model from Figure 8-1. Even those architectural models that are amenable to simulation may need to be augmented with an external formalism in order to enable their execution. For example, models of event-based architectures may not provide event generation, processing, and response frequencies. To include that information, the architectural model may be mapped, for example, to a discrete event system simulation formalism or a queueing network, with possible ranges of such frequencies specified, and then simulated.

Of course, every time additional information such as event frequency is introduced, architects run the risk of injecting imprecision into the architectural model, and hence into the analysis results. On the other hand, since simulation is tool supported, architects can supply many different values for the model parameters, representing different possible system usage scenarios. They would thereby obtain more precise results than would

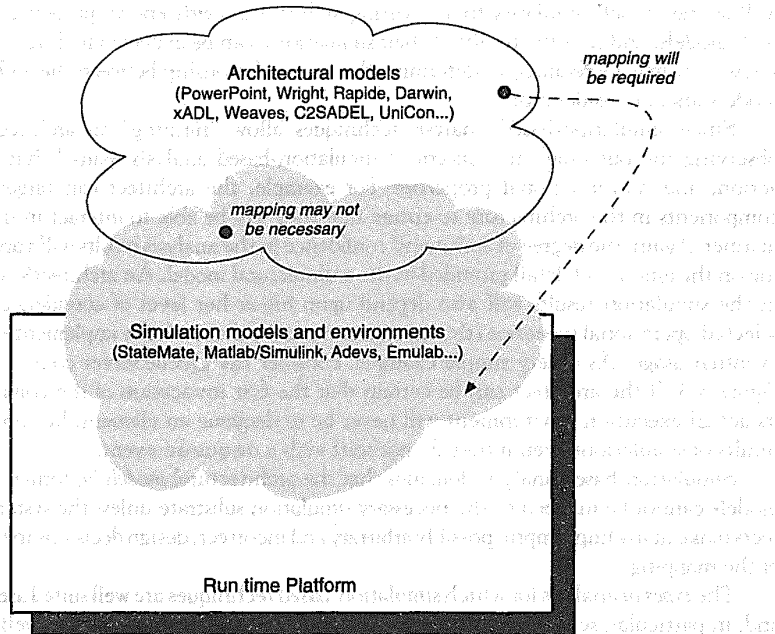


Figure 8-13.
The conceptual
relationship
between
architectural and
simulation
models.

be possible with one-size-fits-all model-based analysis, and do so much more easily and inexpensively than via inspections and reviews.

Several software simulation platforms exist, and different ADLs discussed in Chapter 6 and in the previous section have made use of them. The conceptual relationship between ADLs and simulation models and platforms is depicted in Figure 8-13: Certain architectural models (such as UML's state-transition-based models or Rapide's partially ordered event-based models) can be simulated directly; other models may require a mapping to the simulation substrate. The required mapping from an architectural model to a simulation model may be partial or complete and of various degrees of complexity, depending on the semantic closeness between the ADL and the simulation platform. In turn, the simulation platform can be executed on top of a specific run time platform.

The goals of simulation-based architectural analysis techniques can be any of the four Cs: completeness, consistency, compatibility, and correctness. However, similarly to software testing, because of the nature of simulation these properties can be established only with a limited degree of confidence and possibly only for a particular subsystem or system property.

The scope of simulation-based architectural analysis usually is the entire system or a particular subsystem, as well as the dataflow in the system. However, it is possible to isolate individual components and connectors and simulate their behaviors as well. Simulation-based techniques can be particularly effective in assessing the compliance of the given architectural model to a higher-level model from which it has been derived, as

well as the model's similarity to an existing architecture with known properties. Because such models, and thus the results of their simulations, can be expected to differ, additional know-how may be required to determine the exact relationship between the architectural models under consideration.

Since simulation-based analysis techniques allow "running" an architecture and observing the outcome, the *concern* of simulation-based analysis spans behavior, interaction, and non-functional properties. For example, the architect can target a set of components in the architecture to ensure that they will be able to interact in the desired manner. Again, the degree of architects' confidence in the analysis results will vary depending on the amount of detail provided in the architectural model. An architect's confidence in the simulation results will also depend upon his or her level of confidence that the selected operational scenarios (that is, system inputs) will match the implemented system's eventual usage. As a very simple example, consider the QueueServer component from Figure 8-3: If the architect can be certain that the first invocation of the component in its actual execution environment will never be to dequeue an element, he can trust the results of simulations even if they do not start with a dequeue event.

Simulation-based analysis demands that the architectural *models* be formal. Informal models cannot be mapped to the necessary simulation substrate unless the system's architects make many impromptu, possibly arbitrary and incorrect, design decisions in the course of the mapping.

The *types* of analysis for which simulation-based techniques are well suited are dynamic and, in particular, scenario-based analysis. Simulation-based techniques are well suited to assessing the system's run time behavior, interaction characteristics, and non-functional qualities. If known system usage scenarios are postulated, the architects no longer need to be concerned with the completeness of the analysis results and whether the results will be representative of the system's actual use.

As already discussed, with regard to *automation level* simulation-based techniques are typically fully automated, requiring no human intervention other than supplying system inputs. However, the process of mapping an architectural model to a simulation model may require significant human involvement. This, in turn, can render the process error-prone.

Finally, simulation-driven architecture analysis techniques can be useful to all system *stakeholders*. Setting up and running simulations may require a high degree of technical expertise and familiarity with the architectural model, the architecture modeling notation, as well as the simulation substrate.

XTEAM

The eXtensible Tool-chain for Evaluation of Architectural Models (XTEAM) is a model-driven architectural description and simulation environment for mobile and resource-constrained software systems (Edwards, Malek, and Medvidović 2007). XTEAM composes two existing, general-purpose ADLs, enhancing them to capture important features of mobile systems. It implements model generators that take advantage of the ADL extensions to produce simulation models.

For modeling architectural concerns, XTEAM builds an architectural meta-model. This meta-model leverages xADL's Core, which, as the reader will recall from Chapter 6, defines architectural structures and types common to most other ADLs. This meta-model is augmented with finite state processes, or FSP, which allows the specification of

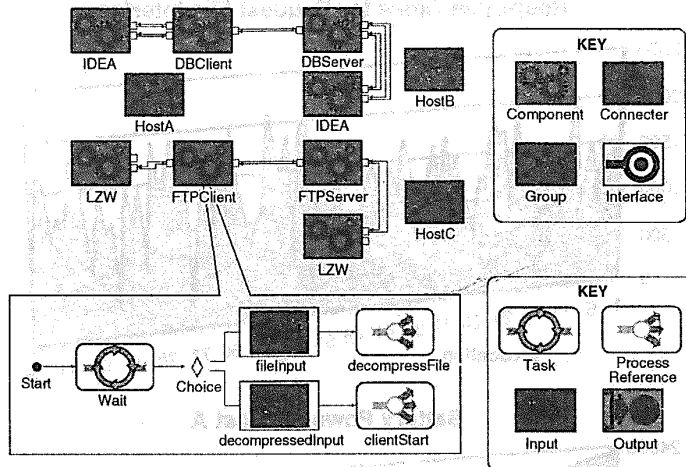


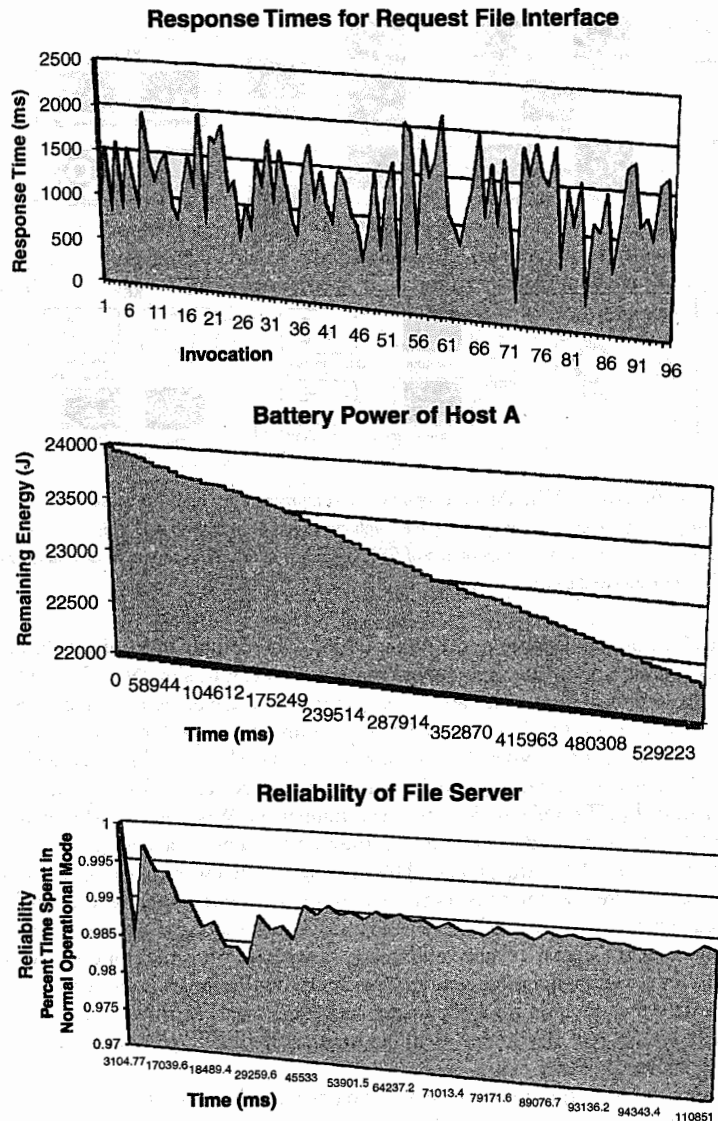
Figure 8-14. An example XTEAM model of a system comprising three hardware hosts and eight software components. The behavior of one of the components has been highlighted. This behavior represents XTEAM's syntactic overlay on top of FSP. XTEAM allows behaviors to be abstracted in the form of the *ProcessReference* construct.

component behaviors (Magee and Kramer 2006). The combination of xADL and FSP allows the creation of architectural representations that can be simulated. XTEAM masks many of the details of the two notations via a visual language depicted in Figure 8-14.

Models conformant to the composed ADL contain sufficient information to implement a semantic mapping into low-level simulation constructs, which can be executed by an off-the-shelf discrete event simulation engine, such as adevs (Nutaro). This semantic mapping is implemented by XTEAM's model interpreter framework. When invoked by an architect, the XTEAM interpreter framework traverses the architectural model, building up a discrete event simulation model in the process. The interpreter framework maps components and connectors to discrete event constructs, such as atomic models and static digraphs. The FSP-based behavioral specifications encoded in XTEAM, such as the model shown in the bottom diagram of Figure 8-14, are translated into the state-transition functions employed by the discrete event simulation engine. The interpreter framework also creates discrete event entities that represent various system resources, such as threads. Hook methods provided by the framework allow an architect to generate the code needed to realize a wide variety of dynamic analyses.

XTEAM implements simulation-based analysis capabilities for end-to-end latency, memory utilization, component reliability, and energy consumption. Each analysis type requires the implementation of the needed XTEAM ADL extensions and a model interpreter. For example, XTEAM implements the component reliability modeling extension and analysis technique using the HMM-based approach discussed in Section 8.8.2. This reliability estimation approach relies on the definition of component failure types, the probabilities of those failures at different times during component execution, and the probability of, and time required for, recovery from failure. In order to support this type of analysis,

Figure 8-15. End-to-end latency, energy consumption, and reliability profiles of different elements of the architecture from Figure 8-14, obtained from adevs simulations. The simulations were generated automatically from the XTEAM model.



XTEAM's FSP-based behavior language had to be extended to include failure and recovery events and probabilities. XTEAM was then augmented with an analysis technique that determines if and when failures occur as the components in the system progress through different tasks and states.

Another analysis supported by XTEAM is that of the modeled mobile system's energy consumption rate. The selected energy consumption model by Seo (Seo, Malek, and Medvidović 2007) was implemented as another XTEAM ADL extension and model interpreter, which incorporated the necessary elements of the energy cost model. Then, another adevs simulation was generated by XTEAM to show the system's likely energy consumption over time, under certain usage assumptions. The results of three different simulations for the architecture depicted in Figure 8-14 are shown in Figure 8-15.

Goals	Consistency Compatibility Correctness	XTEAM: Analysis Summary
Scope	Component- and connector-level Subsystem- and system-level Data exchange	
Concern	Structural Behavioral Interaction Non-functional	
Models Type	Formal Dynamic Scenario-based	
Automation Level	Automated	
Stakeholders	Architects Developers Managers Customers Vendors	

8.9 END MATTER

A major part of what makes good architectural practice is deciding what one wants out of the architecture. An individual or organization cannot and should not focus on architecture just because it is widely accepted that doing so is the right thing to do. Instead, system stakeholders should first decide on the specific benefits they want to gain from the explicit architectural focus. One clear benefit of this is the establishment of a system's properties of interest before the system has been implemented and deployed. It is critical to make sure that the system possesses the key required properties and does not have or is reasonably unlikely to have any undesirable characteristics. Architectural analysis provides system stakeholders with a powerful suite of techniques and tools for doing that.

However, architectural analysis is neither easy nor cheap (see "The Business Case" sidebar). It should be planned for and applied carefully. The system's stakeholders must be clear about their project objectives and must map those objectives to specific architectural analyses. This chapter has provided a structure that can be used to determine the architectural analysis techniques that are best suited for specific system development scenarios. In many cases, multiple analysis techniques may need to be used in concert. Again, deciding

which techniques to use is a critical part of an architect's job: using too many will expend the project's resources unnecessarily; using too few will run the risk of propagating defects into the final system; using wrong analyses will have both of these drawbacks.

Once a system's architecture is properly designed and modeled, and it is established through analysis that it possesses the desired properties, the architecture is ready to be implemented. Chapter 9 presents a detailed overview of architectural implementation challenges, techniques, and tools. Chapter 10 then discusses the software architectural implications in an implemented system's deployment and run time mobility. Chapter 14 will return to these issues in the context of architectural adaptation.

The Business Case

The up-front costs in an architecture-driven software development project can be significant. Even if we consider architectural analysis alone,

1. Techniques for architectural inspections and reviews typically involve the participation of many stakeholders over several days. Recall ATAM for example.
2. Model-based analysis requires the development or acquisition of multiple tools as well as the possible introduction of formalisms, each with its own learning curve, required by those tools. For example, recall the adoption of Markov models for reliability analysis or Wright's incorporation of CSP to ensure deadlock freedom.
3. Finally, simulation-based analysis most often requires augmenting the architectural model or recasting it into the form required by the simulation substrate. Recall XTEAM's reliance on xADL, FSP, and adevs.

Clearly, architectural analysis can be costly. These are the very costs of which many managers and organizations are leery since the planned revenue from the system under development is still very far off. Therefore, the costs must be offset by the benefits gained from architectural analysis. The benefits of detecting and removing software system defects early on are well known: Many sources, starting with Barry Boehm's seminal work *Software Engineering Economics* (Boehm 1981), have documented that discovering defects earlier rather than later can reduce the cost of mitigating those defects by an order of magnitude or more.

This does not mean that all costs of architectural analysis are automatically justified. There is no particular need to do analysis for the sake of demonstrating that one can do it or to show off a technique or tool. Analysis must have a clear objective and take place within a particular, well-defined context. Put another way, the undertaken analysis activity must recoup its own costs via the benefits it yields. Nobody should pay more for something than the value it gives back. For example, a system's architects should not spend more to perform deadlock analysis than the projected risk and associated cost of deadlock for the system in question. The bottom line for any project with a tight budget—which, in practice, means for the great majority of projects—is that under no circumstances should the cost of architectural analysis impinge upon the organization's ability to actually implement the system.

8.10 REVIEW QUESTIONS

1. What are the architectural analysis criteria?
2. What are the four Cs of architectural analysis?
3. What is the difference between internal and external consistency?
4. What is interface inconsistency? Provide an example.
5. What is interaction inconsistency? Provide an example.
6. What are the advantages and disadvantages of informal architectural models as pertaining to analysis?

7. What are the advantages and disadvantages of formal architectural models as pertaining to analysis?
8. Why is system-level analysis important if you have already completed component- and connector-level analysis?
9. Why is the analysis of the data exchanged in an architecture important?
10. Why is it important to analyze an architecture for non-functional concerns?
11. Who are the possible stakeholders in architectural analysis?
12. Is scenario-driven analysis a special case of static analysis or dynamic analysis? Justify your answer.
13. Name some advantages and disadvantages of architectural inspections and reviews.
14. What is model checking? How does it pertain to architectural analysis?
15. How does one analyze an architecture for deadlocks?
16. What are discrete-time Markov chains used for?
17. What is simulation?
18. If an architectural model is expressed in a notation that does not have executable semantics, can it still be simulated? Justify your answer.

8.11 EXERCISES

1. Select one of the models of Lunar Lander from Chapter 4 or Chapter 6. Determine its components. For each individual component, model the services it provides and requires in terms of their names and interfaces. After you have completed the model, analyze the model manually for name and interface inconsistencies.
2. Model the behaviors of the Lunar Lander components used in the previous exercise, by specifying each component operation's pre- and post-conditions, as discussed in this chapter. After you have completed the model, analyze the model manually for behavioral inconsistencies.
3. As an alternative, or in addition to the pre- and post-condition based behavior model from the preceding exercise, model the Lunar Lander component behaviors with state-transition diagrams, such as those used by UML. Discuss the specific advantages and difficulties in using this model in ensuring behavioral consistency. Research whether any available techniques can be adopted or adapted for ensuring the consistency of two state-transition diagrams.
4. Select one of the models of Lunar Lander from Chapter 4 or Chapter 6. Determine its components. Construct each component's interaction model. Analyze these models for interaction inconsistencies. What, if any, are the differences between this task and the task described in the preceding exercise?
5. Download and study the Alloy modeling language and analysis tool suite (alloy.mit.edu). Discuss the possible ways in which Alloy can be used to aid architecture modeling and analysis. Demonstrate your arguments with specific examples of simple architectures, architectural properties, architectural styles, and so on.
6. Select an existing application with a documented architecture. This can be an application you have worked on or an open-source system. Study the architecture if you are unfamiliar with it. Follow the ATAM process to determine the quality attributes important to the system, as well as the architectural decisions employed. Develop a set of scenarios and key decisions underlying the architecture. Use the scenarios to determine whether the architectural decisions result in the required quality attributes.
7. Discuss the potential shortcomings of applying the ATAM process without all the prescribed stakeholders and a separate architecture evaluation team.
8. Select one of the models of Lunar Lander from Chapter 4 or Chapter 6. Provide its state-based reliability model as discussed in this chapter. Study an existing off-the-shelf technique for reliability estimation (such as DTMC or HMM). Apply this technique on your reliability model. Devise strategies for improving your system's reliability.
9. Select an important property of a software system, such as reliability, availability, or fault tolerance. State the assumptions required in order to be able to assess this property at the architectural level. Discuss the missing knowledge about the system that necessitates the assumptions. Then discuss the implications and reasonableness of each assumption.

10. Select two existing ADLs of your choice. Use each ADL to model the Lunar Lander architecture. Use the two ADLs' respective analysis tools to analyze each model for the properties on which the given ADL focuses. Discuss the extent to which the two ADLs' modeling features impact the available analyses versus the extent to which the selected analyses appear to impact the available modeling features.
11. Download and study an available simulation engine such as adevs (www.omni.gov/~lqn/adevs/index.html). Provide a mapping from an ADL of your choice to the simulation engine. Discuss the challenges in doing so and the range of architectural properties for which the simulations can be used. Practically demonstrate the use of simulation for at least one such property.

8.12 FURTHER READING

A significant amount of literature on software architecture has focused on architecture-based analysis. The early work on ADLs, such as Wright and Rapide, focused as much on the enabled analysis techniques as it did on the modeling features. Likewise, when UML first came onto the software development scene in the mid- to late-1990s as a de facto modeling standard, it was almost immediately accompanied by a continuing stream of publications on how UML models can be effectively analyzed for many different types of properties of interest. A widely cited overview of ADLs and their associated analysis capabilities was provided by Medvidović and Taylor in 2000 (Medvidović and Taylor 2000). That work has been recently revisited and the perspective expanded by Medvidović, Dashofy, and Taylor (Medvidović, Dashofy, and Taylor 2007).

Carnegie Mellon University's Software Engineering Institute has, over time, proposed a number of architectural analysis approaches that entail inspections and reviews. Examples are ATAM, studied in this chapter;

its precursor, Software Architecture Analysis Method, (SAAM); and an evaluation method for partial architectures, called ARID. These techniques are described in a book by Paul Clements, Rick Kazman, and Mark Klein (Clements, Kazman, and Klein 2002). Recently, Liliana Dobrica and Eila Niemela provided a broader survey of inspection and review-based architectural analysis approaches (Dobrica and Niemela 2002).

Leveraging explicit architectural models for determining a system's key non-functional properties has gained a lot of interest over time. This has resulted in a cross-pollination of ideas from established disciplines such as reliability, dependability, and fault-tolerant computing on the one hand, and software architecture on the other hand. In 2007 alone, Anne Immonen and Niemela (Immonen and Niemela 2007) have provided a survey of architecture-based methods for availability and reliability estimation, while Swapna Gokhale conducted a separate survey of architecture-based reliability analysis approaches (Gokhale 2007).

Implementation

Implementation is the one phase of software development that is not optional. Regardless of how well the requirements have been captured or how carefully the architecture has been designed and reviewed, software systems must be implemented in code to achieve their ends. Architecture is used to capture important design decisions about a system, using sound engineering principles and knowledge to increase confidence that the target system's qualities match the expectations of its stakeholders. To imbue these qualities in the target system, the implementation *must* be derived from its architecture.

In terms of implementation, architecture is both prescriptive and restrictive. It is prescriptive in the sense that it gives implementers direction on what to produce—how to structure code modules, how they should be interconnected, and how they should behave. It is restrictive in the sense that its guidelines—particularly those specified in the architectural style—tell developers what they may *not* do: the forms of communication that are prohibited, the kinds of behaviors or system states that are not allowed, and so on.

The problem of relating architecture to implementation is one of *mapping*. Concepts defined at the architecture level should be directly connected to artifacts at the implementation level. This correspondence is not necessarily one-to-one. For example, in general, a software component or connector will be implemented using many code and resource artifacts. Likewise, a single software library may be shared among several component implementations (as long as its manner of use does not violate style rules). When this mapping is not maintained, architectural degradation can occur—specifically, the gradual deviation of a system's implementation from its architecture. This generally makes software harder to understand and maintain, and makes it difficult to achieve or retain the qualities embodied by the architecture.

Properly implementing a system so that the architecture survives in the implementation requires a well-formed understanding of how concepts in the architecture (and the architectural style) map onto implementation technologies such as programming

languages, development environments, reusable libraries and components, middleware, and component models. Implementation technologies such as middleware and component libraries can help, but they can also hinder: They nearly always come with their own assumptions that can influence or get in the way of architectural decisions made earlier. Even concepts we take for granted, such as object-oriented programming, can have architectural influences. In this chapter, we discuss techniques and technologies for creating and maintaining mappings between architecture and implementation.

We also introduce and focus on a concept known as an *architecture implementation framework*, which is software that bridges the gap between architectures and implementation technologies. We discuss how to identify, evaluate, and create new frameworks. We also relate frameworks to other, similar technologies such as middleware and component frameworks.

Outline of Chapter 9

9. Implementation

9.1 Concepts

9.1.1 The Mapping Problem

9.1.2 Architecture Implementation Frameworks

9.1.3 Evaluating Frameworks

9.1.4 Middleware, Component Models, and Application Frameworks

9.1.5 Building a New Framework

9.1.6 Concurrency

9.1.7 Generative Technologies

9.1.8 Ensuring Architecture-to-Implementation Consistency

9.2 Existing Frameworks

9.2.1 Frameworks for the Pipe-and-Filter Architectural Style

9.2.2 Frameworks for the C2 Architectural Style

9.3 Examples

9.3.1 Implementing Lunar Lander in the Pipe-and-Filter Style Using the java.io Framework

9.3.2 Implementing Lunar Lander in the C2 Style Using the Lightweight C2 Framework

9.4 End Matter

9.5 Review Questions

9.6 Exercises

9.7 Further Reading

9.1 CONCEPTS

Here, we will discuss concepts and issues related to architecture-based implementation.

9.1.1 The Mapping Problem

Implementing an architecture is a problem of mapping—specifically, mapping design decisions to specific implementation constructs that realize those decisions. From a software-quality perspective, this mapping is a form of *traceability*. In general, the term

traceability can refer to any mechanism for connecting different software artifacts; in this context we specifically mean traceability from architecture to implementation. Choosing how to create and maintain this mapping is critical in architecture-based development. Here, we discuss implementation mappings for several kinds of design decisions discussed in Chapter 6.

Components and Connectors. Design decisions about components and connectors partition the application's functionality into discrete elements of computation and communication. In general, programming environments provide mechanisms such as packages, libraries, or classes that are used to partition functionality in implementations. Here, the challenge is to maintain a mapping between the partitions established by the architecture-level components and connectors and the partitions established by the implementation-level packages, libraries, classes, and so on. If implementations are not partitioned according to the component and connector boundaries specified in the architecture, then component boundaries may break down and cause architectural drift and erosion.

Interfaces. At the architectural level, interfaces can be specified in many different ways. If interfaces are specified in terms of method or function signatures similar to those in the target programming language, mapping is a straightforward process of translating the method signatures into code. However, if the architecture-level interface definition is more complex—specifying a protocol or set of state transitions, then greater effort will be required to create an appropriate implementation.

Configurations. At the architectural level, configurations are often specified as linked graphs of components and connectors. These graphs specify and constrain how the components and connectors interact through their interfaces. The same interactions and topologies must be preserved in the implementation. Many programming languages include features that allow one module to refer to another module by way of its interface, rather than its implementation (for example, though explicitly defined interfaces as in Java, or function pointer tables in C). Additionally, some programming languages and middleware systems allow the use of reflection or dynamic discovery to connect and disconnect components at runtime. When these constructs are available, it is often possible for the implementation-level links between components and connectors to be specified independent of the components and connectors themselves, or even generated from the architecture description.

Design Rationale. Design rationale is a construct that often has no specific mapping to implementation, since it is not something that directly influences the functionality of the application. Often, the best way to retain design rationale during implementation is by writing it down in source-code comments or external documentation.

Dynamic Properties (Behavior). Depending on how they are modeled, architecture-level behavioral specifications can ease or facilitate implementation. Some behavioral specifications can be translated directly into implementation skeletons or even complete implementations. However, this is not always the case; formal behavioral specifications often lack bindings to programming-language-level constructs and therefore it is difficult to determine whether a behavioral specification is actually implemented correctly. Some

behavioral specifications are more useful for generating analysis or testing plans than for implementations.

Non-Functional Properties. Implementing non-functional properties is perhaps one of the most difficult propositions in software engineering. The best way to accomplish this is through a combination of techniques—documenting rationale, inspections, testing, user studies, and so on. This difficulty is why refining non-functional properties into functional design decisions (when possible) is so important.

One-Way and Round-Trip Mapping

Architectures and implementations must inevitably co-evolve. Architectures can evolve because of changing requirements or increased understanding, and these changes must be propagated to the implementation. Likewise, discoveries and changes made during implementation will affect the architecture. Keeping architecture and implementation in sync is a challenging problem, involving issues of process, tool support, and organizational culture. Aspects of the mapping that lack strong traceability are often the first to diverge.

Maintaining the architecture-implementation mapping in the face of change depends on the process of change. One option is to mandate that all changes begin from the architecture—the architecture is changed first, the mappings to implementation are used, and then the implementation is updated through the use of automated tools or manual processes. This is effectively a one-way mapping. Another option is to allow changes to be initiated in either the architecture or the implementation. In this case, automated tools or manual processes are still used to update the other artifact. This is a two-way mapping.

Two-way mappings are better for detecting and resolving architectural drift and erosion, but they are also more complex and expensive to create and maintain. This is sometimes known as a round-tripping problem because changes have to be mapped from architecture to implementation and back again. Various strategies for maintaining both one-way and round-trip mappings are discussed in this chapter.

9.1.2 Architecture Implementation Frameworks

When developing a system, an ideal approach would be to define the architecture first and then select implementation technologies (programming languages, software libraries, operating systems, and so on) that most closely match its needs. This ideal is difficult to achieve—programming languages rarely have explicit support for architecture-level constructs. Moreover, selection of implementation technologies will often be driven by extrinsic or accidental factors such as cost, maturity, platform support, organizational culture, and even externally imposed or wrongheaded requirements specifications or standards.

An important strategy for bridging the gap between concepts in an architecture and a system's implementation technologies is to use (or develop) an architecture implementation framework (Malek, Mikic-Rakic, and Medvidović 2005).

Definition. An *architecture implementation framework* is a piece of software that acts as a bridge between a particular architectural style and a set of implementation technologies. It provides key elements of the architectural style in *code*, in a way that assists developers in implementing systems that conform to the prescriptions and constraints of the style.

By far, the most common example of an architecture framework in use today is the Standard I/O library in UNIX (University of California 1986) and similar operating systems. Although few developers may recognize it as such, it is actually a bridge between the pipe-and-filter style (which is character-stream oriented and concurrent) and procedural, nonconcurrent programming languages such as C. It provides architectural concepts such as access to interfaces via readable and writable character streams in a way that fits the target environment (for example, procedure calls). A fuller discussion of how the Standard I/O library serves as an architectural framework appears later in this chapter.

Architecture frameworks are effectively technologies that assist developers in conforming to a particular architectural style. However, most frameworks will not prevent developers from wandering outside the constraints of the style. For example, just because a UNIX program imports the Standard I/O library does not mean that the program will work in a pipe-and-filter style; it may read and write all its data from named disk files and ignore the standard input and output streams completely.

It is possible to develop applications in almost any architectural style without the use of an architecture framework. However, this usually means weaving the architectural concepts throughout the implementation and makes it difficult to develop and maintain them. In the cases where no framework exists for a particular programming language/operating system combination, developers will usually end up implementing a set of software libraries and tools that amount to an architecture framework anyway.

A natural question to ask is: How are frameworks represented in architectural models? From an architectural perspective, frameworks are often considered to be a substrate underlying all components and connectors. Therefore, it is unusual to see a framework modeled as a component or connector in the architecture itself. However, frameworks often include implementations for common components and connectors (such as those defined by the style—a pipe connector or an event-bus, for example) that serve as implementations for components and connectors that are specified in the architecture.

Same Style, Different Frameworks

A single architectural style can be supported by a number of different, alternative frameworks. This can happen for a number of reasons. First, different programming languages and implementation platforms usually require different frameworks. For example, Java applications use classes in the `java.io` package (Harold 2006) to perform stream input and output; these classes are how Java implements functions similar to the C standard I/O library. C++ programmers can either use the object-oriented `iostream` library or the procedural `stdio` library for the same purpose. Each of these architecture frameworks bridges the same architectural style (pipe-and-filter) to different implementation technologies (Java, C++, or C).

Sometimes, multiple frameworks for the same combination of style, programming language, and operating system will be developed. Usually, these frameworks distinguish themselves based on different qualities or capabilities. A good example is the New I/O (`java.nio`) package in Java (Hitchens 2002). Like the older `java.io` package, `java.nio` allows programs to read and write data streams from various sources. However, the New I/O package provides enhanced capabilities such as native support for buffering, better control over synchronization, and the ability to use fast data transfer techniques such as memory mapping. Users can choose the appropriate framework for their application based on the quality needs of those applications.

9.1.3 Evaluating Frameworks

Frameworks, like any software system, can vary widely along nearly any quality dimension. This is why many frameworks are often developed to support the same architectural style in the same environment. Evaluating a framework, then, is similar to evaluating any important software component.

Platform Support

An architecture framework brings together three key elements: an architectural style, a programming language, and an operating system. One of the most basic criteria for evaluating a framework, then, is platform support. Once an architectural style has been identified, the availability of architecture frameworks for a target programming language/operating system combination can be determined. If the project has the freedom to select the implementation platform based on the architecture (which is becoming increasingly rare as software systems are more and more likely to run on existing platforms already in the field) then the availability of suitable architecture frameworks should be a criterion for platform selection.

Fidelity

One quality that is particularly important in architecture implementation frameworks is fidelity, specifically fidelity to the target architectural style. To be useful, a framework need not provide direct implementation support for every single design decision in its target style; for example, it may provide communication mechanisms but leave the concurrency of the architecture up to the individual component implementers. Furthermore, frameworks often provide support for following stylistic constraints, but not enforcement; that is, the framework will make it easy for implementers to follow the constraints of the style, but will not explicitly prevent them from breaking the constraints. More faithful frameworks are generally better at preventing or avoiding architectural drift and erosion. However, this generally comes at a cost because the frameworks are more complicated, bigger, or less efficient.

Matching Assumptions

Architectural styles induce certain design decisions and constraints on applications. Frameworks can do the same thing—ideally, the decisions and constraints induced by the framework are the same as those induced by the target style. However, styles often leave many aspects of a system unconstrained, and frameworks have the additional responsibility of supporting the concrete implementation activity. For these reasons, frameworks may induce additional constraints on applications. For example, a framework might assume that the system will be instantiated and configured only by the framework, or that individual components and connectors will not start their own threads of control, or that each software component in the architecture can be associated with a module in the target programming language.

Problems can occur when the assumptions of a framework conflict with the assumptions of other implementation technologies used on a project. Consider an architecture framework for an object-oriented programming language. This framework might require that every component have a main class that is derived from a base class provided by the framework. However, the project might include several GUI elements, and the GUI toolkit may require that GUI element classes extend base classes provided by the toolkit.

If the programming language is one that does not support multiple inheritance (like Java) there is a serious mismatch between the GUI toolkit and the implementation framework. This situation may not even be an architectural mismatch; it is a mismatch of particular implementation decisions. Nonetheless, strategies must be identified to alleviate this situation.

Thus, when evaluating an architecture framework, it is important to enumerate the assumptions it makes and compare those with the assumptions made by other components, toolkits, libraries, and environments with which the application will interact. Sometimes, workarounds can be developed, especially if the mismatch is a low-level implementation detail. However, if the mismatch is architectural, this might call into question the compatibility of the architectural style itself with the choice of implementation technologies for the implemented application.

Efficiency

In general, architecture frameworks add a layer of functionality between the application and the hardware it runs on. One of the primary dangers of introducing new layers is a decrease in application efficiency. This concern is especially important when dealing with architecture frameworks, since they tend to pervade the application. An architecture framework may mediate all communication between components in a system, for example, or dictate the concurrency policy for the entire application. When this is the case, efficiency should be a primary selection criterion for a framework.

Before committing to a framework, it is a useful exercise to run benchmarks on the framework with parameters derived from the target application to get a feel for the upper bound of application performance using the framework. For example, if a framework can exchange 10,000 messages per minute in a dummy application whose sole purpose is to exchange messages as quickly as possible, it is not realistic to build an application with that framework that will exchange 20,000 messages per minute.

Other Considerations

As noted above, the issues involved in selecting an architecture framework are very similar to those involved in selecting any software component. It could be argued that because frameworks have such a pervasive effect on applications, they are the most critical element of all to select. Qualities such as size, cost, ease of use, availability of source code, reliability, robustness, portability, and many others are all important when choosing a framework.

9.1.4 Middleware, Component Models, and Application Frameworks

A spectrum of technologies exists to integrate software components and provide services above and beyond those provided by a given programming language/operating system combination. These technologies go by a number of different names: middleware, component models (or component frameworks), and application frameworks. We will refer to these systems collectively as "middleware." Popular examples include CORBA (Object Management Group 2001), JavaBeans (JavaSoft 1996), COM/DCOM/COM+ (Sessions 1997), .NET, Java Message Service (JavaSoft 2001), various Web Services technologies, and so on.

There are many similarities between architecture frameworks and middleware. Both of them provide developers with implementation services that are not natively available

in the underlying programming language or operating system. For example, CORBA middleware provides services such as remote procedure calls (RPCs) and the ability to dynamically discover the interfaces of objects. The JavaBeans component model introduces a new concept to Java: the bean, an object that follows certain interface guidelines that make it possible to compose beans more easily.

Architecture implementation frameworks are a form of middleware. The difference between traditional middleware and architecture frameworks is the focus on architectural style. Architecture implementation frameworks are implemented specifically to support development in one or more architectural styles. Here, the *style* is the primary artifact driving the implementation technology. Middleware is created based on the services that are provided, generally without regard to the style of the application being developed.

How Middleware and Component Frameworks May Induce a Style

Middleware often constrains applications in ways that are similar to architecture frameworks. Middleware often influences how an application's functionality is broken up into components, how those components interact (often through middleware-provided services akin to connectors) and also the application's topology. These are generally architectural concerns. In this sense, middleware can induce an architecture or architectural style on an application (Di Nitto and Rosenblum 1999).

CORBA [and CORBA-like technologies such as COM and RMI (Grosso 2001)] are a good example of how middleware can influence application architectures. CORBA breaks up an application into objects that may reside on different hosts. Objects that participate in the application expose their own services through provided interfaces whose method signatures are specified in an interface definition language (IDL). Objects look up other objects through services such as naming services or trading services, and then call each other using a request-response pattern, passing only serializable parameters across the interface boundaries. Together, these constraints comprise an architectural style that might be referred to as the "distributed objects" style.

If system stakeholders have chosen the distributed objects style for their application, then CORBA-like middleware might serve as an ideal architecture framework. However, things are rarely this simple. Presented with an application to design, software architects have to make hundreds of design decisions. Choosing the application's architectural style is one of the most important decisions they will make. However, experienced architects are also familiar with many different middleware technologies and the advantages of those technologies. The services provided by many middleware technologies can be seductive, and often the capabilities of a particular middleware system will influence an architect's decision-making process. Architects must be especially careful to avoid having a middleware technology overly influence their designs.

Resolving Mismatches between Architectural Styles and Middleware

Two major conflicts can arise between architectural styles and middleware:

1. The architectural style chosen for the application does not match that induced by the middleware chosen.
2. The application's designers chose a middleware first based on services provided and let this have an undue influence over the architectural style of the application.

When selecting implementation technologies, it is critical to understand that the quality benefits provided by that technology often come with architectural implications, which may not be compatible with your architecture's design. For example, CORBA provides the benefits of distribution and reflection, but in a form that induces systems that are based on objects that communicate by request-response procedure calls. This may cause certain architectural drawbacks, such as increased latency and synchronization. Allowing the choice of middleware to influence the architecture is backward: It is the tail wagging the dog. Architecture should influence your choice of middleware.

When there is an architectural mismatch between middleware and the target architectural style, several options are available:

Change the Style. The architectural style can be changed to better fit the middleware. This should be done only when the benefits of using the middleware outweigh the costs of adapting it to work with the target style.

Change the Middleware. The middleware can be adapted to better fit the architectural style. This strategy can be difficult because middleware packages are often large, complex, or proprietary.

Develop Glue Code. Architecture frameworks can be built on top of middleware, leveraging the parts of middleware that match, and working around the parts that do not. This way, neither the style nor the middleware itself has to be adapted.

Ignore Unneeded Middleware Services. Some middleware packages or component frameworks might provide a host of services that cut across many aspects of application development. However, it may be possible to use a subset of these services selectively, and ignore the services that are not compatible with (or relevant to) the target architectural style.

Hide the Middleware. Developers use middleware because it provides certain services. If those services are not necessarily cross-cutting, and can be applied at specific points in the architecture, then it may be possible to hide the middleware inside individual components or connectors. For example, if CORBA is being used only to facilitate communication between heterogeneous components running on different hosts, all CORBA-related code can be isolated within individual connectors that need cross-host communication. Other CORBA services such as lookup and dynamic interface discovery might be used entirely within the context of the connectors or simply ignored.

Using Middleware to Implement Connectors

Many middleware packages provide services that are effectively communication-centric: They provide different mechanisms for heterogeneous components to communicate. If improving communication in an architecture is a goal, then using middleware as the basis for implementing connectors, rather than the whole application, can allow a system to avoid having the middleware's assumptions bleed into and corrupt the architecture's design decisions.

In this scenario, architects first should define and identify the capabilities required for a connector, ideally without regard to how that connector will be implemented. Then, middleware should be selected that can provide all (or most) of those capabilities and also fit with the other project goals. If capabilities are not provided directly by the middleware,

they should be implemented as part of the connector's implementation. The result is a connector that fulfills the architectural need, rather than one that bows to assumptions made by middleware developers.

For example, a connector might be needed that provides message-passing support between a C++ component running on Linux and a Java component running on Windows. Two message-oriented middleware packages may be available: a commercial package that has C++ and Java support for both platforms, but is proprietary and expensive, and an open-source solution that supports both platforms but only C++ components. If budgets are tight, the open-source solution can be selected, and a Java Native Interface (JNI) adapter (Liang 1999) can be written to allow the Java component to communicate with the middleware.

9.1.5 Building a New Framework

Occasionally, circumstances motivate the development of a new architecture implementation framework. Good reasons to develop a new framework include:

- The architectural style in use is novel.
- The architectural style is not novel but it is being implemented on a platform for which no framework exists.
- The architectural style is not novel and frameworks exist for the target platform, but the existing frameworks are inadequate.

Developing an architecture framework is a task that should not be approached lightly. These frameworks will impact almost every part of the applications built atop them and can be a make-or-break factor for the success of those applications, so great care should be undertaken in their design. Developing an architecture framework is, in many respects, like developing any other application—it requires the development of requirements, a design, input from many stakeholders, quality evaluation, and so on. As such, almost everything we have said in this book about developing applications in general can be applied to architecture frameworks. There are, however, some additional guidelines that can be applied specifically to framework development:

Have a Good Understanding of the Style First. Developing an architecture framework with an incomplete understanding of the target architectural style is a recipe for disaster. There will be no standard by which to measure the framework for fidelity or completeness. A clear, concise set of the rules and constraints of the architectural style should be developed before framework design begins.

Limit the Framework to Issues Addressed in the Architectural Style. To the greatest extent possible, an architecture implementation framework should be independent from any specific target application. Including application-specific features (that are not part of the style) in a framework limits the reusability of the framework and blurs the line between what is part of the application and what is part of its framework.

Choose the Scope of the Framework. Well-implemented architecture frameworks are valuable reusable assets for the organizations that develop them. Developers of a new framework must decide how the framework will be reused in the future to properly scope

its capabilities. For example, a particular architectural style may be amenable to dynamic architectures—those that change their structure on the fly. However, the initial target applications built in the style may not take advantage of this. Whether or not to implement dynamism in the framework depends on how likely it is that dynamism will be needed in a future project (or a future version of the current project). This leads to the following related piece of advice:

Avoid Over-engineering. When building new frameworks, it is tempting to include all sorts of clever or useful capabilities, regardless of whether the target applications will actually use them. This is especially true because frameworks are often (and should be) developed separately from specific applications. These additional capabilities can involve additional layers and levels of abstraction and have significant effects on the framework, particularly on its usability and performance.

Limit Overhead for Application Developers. Every framework puts some additional burden on application implementers—to include boilerplate code in components, to implement a standard set of behaviors that the framework can call upon, and so on. As burdens on application developers increase, frameworks become more cumbersome and less palatable. Limiting their additional obligations (either through framework design or tool support) can mitigate this.

Develop a Strategy for Legacy and Off-the-Shelf Resources. Almost any application is bound to include elements (components, connectors, middleware, and so on) that were not developed with the framework in mind. Without a documented or tool-supported strategy for integrating these external resources, developers will be forced to come up with their own mechanisms on an ad hoc basis. This can cause problems as developers reinvent the wheel (or worse, reinvent different wheels). Framework developers should strongly consider the kinds of external resources that might be incorporated in applications and establish strategies for integrating these resources to distribute with the framework.

9.1.6 Concurrency

In the past, many software-intensive systems could be designed to run on a single computer with a single processor. Today, even individual computers have multicore or multithreaded processors that can perform multiple tasks simultaneously. Furthermore, many modern applications include some form of distribution over a network, where each network host will have one or more processors. In the very near future, systems will have to be designed for concurrency: multiple tasks in the system executing simultaneously (Magee and Kramer 2006). Concurrency is generally implemented with a variety of strategies; on a single host, multiple threads or operating system processes are generally employed. On a network, different hosts necessarily run independent processes that must work together and implement the behavior of the system as a whole.

Most architectural styles have some notion of concurrency, whether it is simple synchronization or complex multiprocessing. Pipe-and-filter, one of the simplest styles, was developed to take advantage of concurrency to process partial results in parallel; it was an improvement over batch processing systems that were unable to do so. Many of the architectural styles identified in Chapter 4 have specific provisions for which elements can (or should) run concurrently.

Many architecture implementation frameworks and middleware packages have concurrency management as one of their primary features. Later in this chapter, two example implementations of Lunar Lander in both the pipe-and-filter and C2 architectural styles are presented. In both cases, concurrency is handled entirely by the underlying framework or operating system. In the case of the pipe-and-filter system, each filter runs in a concurrent operating system process; in the case of the C2 system, each component and connector runs in its own thread of control.

An increasing amount of research is going into new programming models to support concurrency. However, concurrent programs are still difficult to write. If the architectural style has a concurrency policy that is well-matched to the target application, support for concurrency can be implemented primarily in the architecture framework. Concurrency bugs can lead to race conditions and deadlock—two of the most difficult faults to reproduce and track down. Encapsulating the implementation of concurrency in well-tested framework or middleware code can help to mitigate the risks of deadlock and race conditions (although it cannot eliminate them).

9.1.7 Generative Technologies

One proposed “silver bullet” that has received quite a bit of attention over the years is the idea that software system implementations can be made much more efficient and effective by generating (parts of) those implementations directly from their designs. Indeed, this is the focus of the Object Management Group’s (OMG) Model Driven Architecture initiative (Mukerji and Miller 2003), described in Chapter 16, as well as many generative technologies that have been developed over the years.

Because generation can derive (partial) implementations directly from designs, generation is an attractive strategy for maintaining the mapping from architecture to code. However, it is generally not a comprehensive (or easy) solution to implement properly. Some generative strategies that can be employed in architecture-centric development are described in the following text.

Generation of Complete Implementations of Systems or Elements. Given a sufficient architectural specification, including structural, interface, and complete behavioral specifications, it is possible to generate a complete implementation for a component, connector, or even an entire system. When this strategy is employed, architectural drift and erosion can be effectively eliminated, since implementations are simply transformations of the architecture. In practice, however, this is extremely difficult, due to the extensive amount of detail needed to generate implementations—the behavioral specifications for a component, for example, are usually of equal complexity to code implementing the component.

Generation of Skeletons or Interfaces. It is also possible to generate partial implementations of elements or systems from architectural models. For example, if interfaces are well described, it is possible to generate code skeletons for each service or method in the interface, and allow implementers to fill in the behavior. Likewise, if partial behavioral specifications are available (in the form of statecharts, for example), finite-state automata can be generated in code with the behavior for each state left up to coders.

Generation of Compositions. In situations where a library of reusable component and connector implementations is already available and systems are simply composed from

this library, architectural models can be used to generate the configurations and glue code needed to connect the elements into a complete system. This strategy is generally most effective in the context of domain-specific software engineering (see Chapter 15).

In any generative effort, the round-tripping problem becomes paramount. In the context of generation, one-way approaches allow one artifact to be generated from another—for example, for code to be generated from architectural models. Round-trip approaches allow changes in the target artifact to be reflected back in the source artifact automatically. For example, in a one-way approach, a component in an architectural model might result in the creation of a new Java package containing class files. In a round-trip approach, the creation of a new Java package might result in the generation of a new component in the architectural model, as well. In general, this requires maintaining some meta-data in the generated code—usually in specially formatted comments. While round-trip approaches are preferable to one-way approaches, they are generally tricky to implement correctly, especially when architectural modeling and code development tools are not well integrated (as is often the case).

9.1.8 Ensuring Architecture-to-Implementation Consistency

Even with the use of an architectural framework, it is rarely obvious whether an implementation actually conforms to its prescribed architecture. Determining whether this is the case will generally require a combination of techniques, including manual inspection and review. There are several strategies that can make this task easier.

Create and Maintain Traceability Links. The existence of links, or explicit mappings, from architectural elements to implementation elements can assist developers in determining whether each architectural element has a corresponding implementation and vice versa. Having these links makes it easier to determine whether something has been inadvertently ignored. If these links are to concrete parts of an architecture model and/or concrete implementation artifacts, then automated link checking can be used to determine whether any links have broken due to changes in either the model or the implementation. This strategy works well for concrete artifacts, but mapping across different levels of abstraction or elements that do not have a direct architecture-to-implementation link can be tricky.

Include the Architectural Model. An architectural model may contain information that can be used directly in a system's implementation. For example, a description of a system's structure in a model (indicating how components are to be instantiated and connected) can be used as an implementation artifact. A tool can be used to extract information about components, connectors, and their topology directly from the architecture description and wire the system up in this way automatically. This can be done at build time or during system startup. In either case, one form of architecture-implementation-correspondence is guaranteed, because the structure of the implemented application is derived directly from the architectural model.

Generate Implementation from the Architecture. Depending on the form and contents of an architectural model, it may be possible to generate portions of an implementation directly from the model using automated tools. If the set of components in a system is specified and the architectural style of the application is known, it is possible to generate

component skeletons for a target architecture implementation framework. If behavioral information is also available in the model, it may be possible to generate some or all of the implementations of those components from the model.

9.2 EXISTING FRAMEWORKS

This section presents examples of architecture implementation frameworks that have been implemented for various architectural styles, shows how they satisfy our definition of a framework, and evaluates their strengths and weaknesses.

9.2.1 Frameworks for the Pipe-and-Filter Architectural Style

Many programmers have used an architecture framework without necessarily being aware of it, specifically an architecture framework for the pipe-and-filter style. Nearly every programming language implemented on every major operating system is bundled with a library or module that serves as an architecture framework for the pipe-and-filter style.

The Standard I/O Framework

The C programming language is single-threaded, uses call-return control flow for procedures and functions, and generally stores and retrieves all data from memory by address. How, then, is the C language made compatible with the pipe-and-filter style, where filters can run in parallel, are generally activated when data becomes available, and where filters retrieve and send data through byte streams? The answer is an architecture framework called the standard input-output (I/O) library, also known by its abbreviated name, `stdio`.

Recall that an architecture framework serves as a bridge between the needs of an architectural style and the services provided by the programming language and the operating system. Here, the C programming language provides generic services: control constructs, the ability to read from and write to memory, and so on. The operating system, on the other hand, provides a number of useful services: concurrency at the process level¹ as well as at least two distinguished data streams for each process ("standard input" and "standard output").

The `stdio` library provides C programs access to the operating system's provided standard input/output streams through a procedural API that treats the streams in the same way as files on a sequential-access storage device like a hard drive. Low-level routines such as `getchar(...)` and `putchar(...)` allow programs to read and write a single byte at a time; more complex routines such as `scanf(...)` and `printf(...)` allow the reading and writing of larger quantities of formatted data. Depending on how streams are implemented in the underlying operating system, different `stdio` implementations may employ techniques such as buffering to improve performance. Different implementations may also have different abilities with respect to blocking: If bytes are written to an output

¹When implementations of C are available on an operating system that supports process concurrency such as UNIX or Windows, the operating system's process scheduler will generally be used to assign CPU time to each filter. However, pipe-and-filter has also been supported on single-process operating systems like DOS: In this case, the filters run entirely in sequence and the output of each filter is stored in a temporary file on the hard disk. This effectively turns all pipe-and-filter applications into batch-sequential applications.

stream, and the next filter in the pipeline is not ready to consume those bytes, then `stdio` may cause the write operation to block (make the caller wait) until the receiver is ready, or it may buffer the data and allow the caller to continue.

This is certainly not the only way that such a framework could be implemented. For example, one could imagine a framework where bytes arriving on the standard input triggered the program to begin execution, rather than the operating system invoking the program's `main(...)` method. With this understanding, we can now evaluate the `stdio` framework in terms of the qualities we discussed earlier.

Platform Support. The `stdio` interface is constructed specifically for the C programming language, and its implementation ships with every implementation of the language. Similar libraries may exist in other languages. Implementations of the framework on platforms with little or no operating system support for streams as an interprocess communication mechanism may be more complicated, or may not support pipe-and-filter applications at all.

Fidelity. The `stdio` library's support for streams is good, but a program that uses it is not constrained to working as a filter. Programs are free to ignore both the standard input and output streams, and do input/output through other mechanisms (for example, interfacing directly with the keyboard or using GUI libraries for data output).

Matching Assumptions. The default assumptions of the `stdio` library with respect to pipe-and-filter systems is that each filter will be a separate operating system process and the operating-system-provided streams (standard input and standard output) will be used for communication. If the application wants to use pipe-and-filter differently (for instance, with filters as portions of a C application running in a single process, or perhaps using a disk file as intermediate storage), then the application has to be modified somewhat. Because the `stdio` library provides practically identical interfaces for reading and writing to different kinds of streams (file streams, in-memory streams, interprocess streams and so on) this widens the kinds of pipe-and-filter applications that can be built with it.

Efficiency. Whether filters run concurrently (one of the key efficiency benefits of pipe-and-filter over batch-sequential architectures) is largely dependent on how the underlying operating system schedules processes. For single-process operating systems, output has to be stored in shared memory or secondary storage as each filter runs sequentially. Largely, the `stdio` library itself has no control over how this is handled.

The java.io Framework

The Java programming language is multithreaded, object-oriented, uses call-return method calls for transfer of control, and bundles code and data within objects. The object classes that are used for constructing pipe-and-filter applications in Java are found in the package `java.io`. Although these classes share a purpose with C's `stdio` library, their design is different. The `java.io` class library defines two primary base classes: `InputStream`, which allows callers to read a sequence of bytes, and `OutputStream`, which allows callers to write a sequence of bytes. Each of these provides a small set of methods for reading and writing single bytes or groups of bytes, as well as a few auxiliary methods for rewinding within a stream during reading and flushing writes.

These low-level base classes define minimal functionality for readable and writable byte streams. These classes are not used directly; instead, subclasses are used. In `java.io`, two kinds of subclasses are provided. One set provides access to concrete data sources and sinks: files, network sockets, in-memory byte arrays, and so on. In addition, three distinguished objects are provided by the runtime environment: `System.in`, `System.out`, and `System.err`, which are used for reading from and writing to the standard input, output, and error streams of the operating system process and are used for creating multiprocess pipe-and-filter applications. Another set of subclasses adds functionality to the basic input and output streams by wrapping these low-level streams. For example, `BufferedInputStream` and `BufferedOutputStream` add buffers to improve performance. `DataInputStream` and `DataOutputStream` add additional interface methods to wrapped streams that allow the reading and writing of basic Java data types (integers, floats, and so on). With this in mind, we can now evaluate `java.io` and contrast it with the `stdio` library.

Platform Support. The `java.io` library is part of the standard set of Java packages, and so is available on any platform that can run Java. Platform-specific features such as how the operating system's standard input and output streams are accessed are abstracted away.

Fidelity. The library's support for streams is comprehensive, but as with `stdio`, programs that use this library do not have to work as a filter. On the other hand, pipe-and-filter architectures running within a single program are easier to construct with `java.io` due to the existence of streams that read from and write to memory, and in-process pipe classes that allow in-process streams to be connected.

Matching Assumptions. The `java.io` library matches the assumptions of the pipe-and-filter style well. In-process pipe-and-filter structures can be constructed with relative ease due to Java's innate support for threading, and the ability to run multiple internal filters concurrently.

Efficiency. Java gives programmers fine-grained control over efficiency mechanisms: Buffers can be used by wrapping a stream in a buffered stream, and threads can be explicitly allocated to separate I/O operations from computationally intensive operations (which can increase performance on multiprocessor machines). However, with increased cooperation from the operating system, it is often possible to achieve even higher efficiency. This motivated the construction of the later `java.nio` (New I/O) package, which can take advantage of faster mechanisms.

9.2.2 Frameworks for the C2 Architectural Style

Constructing applications in the C2 architectural style [(Taylor et al. 1996) (see Chapter 4)] differs markedly from traditional procedure-call and object-oriented programming. It imposes strict rules on how applications are constructed internally and how components within an application communicate. It governs both transfers of control and data within an application, and makes assumptions about concurrency and threading. Because of these differences, frameworks are essential for effective C2 development.

Services provided by a C2 framework arise from the various C2 architectural style constraints. For example, C2 requires that application functionality be partitioned into discrete components (for computation) and connectors (for communication). Therefore,

C2 frameworks provide support, at the programming-language level, for application developers to partition their functionality into modules. C2 components and connectors communicate via asynchronous messages and should operate as if they run in separate threads of control. Frameworks supporting this constraint must provide a concept of an asynchronous message, and must allocate (from the operating system) or simulate (via a technique such as round-robin scheduling of a single thread's activities) multiple threads of control.

C2 frameworks have been developed for many platform/language combinations. C2 Frameworks have been developed for C++, Ada, Java, and other languages running on Windows, UNIX, the Java Virtual Machine, and so on. Several different Java C2 frameworks have been developed, each with different characteristics. We will compare and contrast a basic framework called the Lightweight C2 Framework and a larger but more configurable framework called the Flexible C2 Framework.

The Lightweight C2 Framework

The first C2 framework implemented in Java is known as the Lightweight C2 framework, implemented in only sixteen classes (about 3000 lines) of Java code.

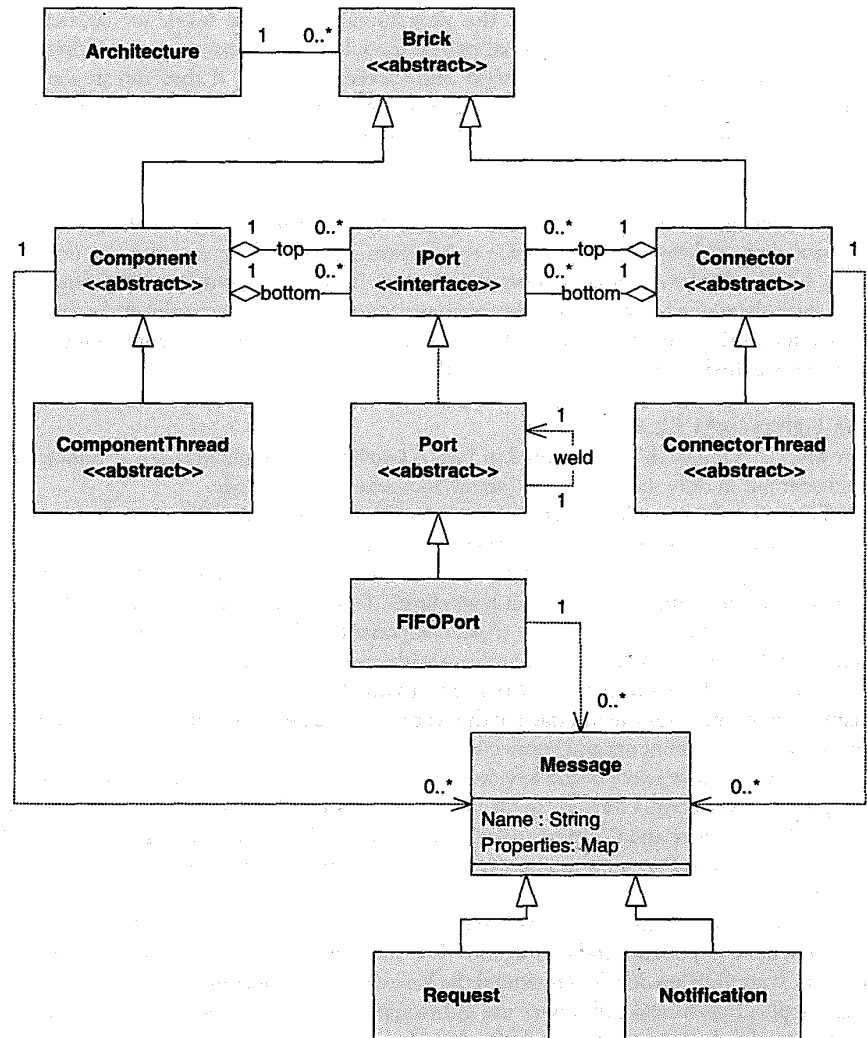
Figure 9-1 shows a selected set of classes from the Lightweight C2 Framework, as well as their relationships, as a UML class diagram. To implement an application using this framework, developers create component and connector implementations as subclasses of the `Component` or `Connector` abstract base classes. Developers may also create additional classes as needed that are called by these component and connector classes. The component and connector classes communicate with each other using only messages, which are instances of the `Request` and `Notification` classes. The developer then creates a main program that uses the interface of the `Architecture` class to instantiate and hook up the various components and connectors.

Figure 9-1 shows some of the key design choices made by the framework authors. C2 components and connectors are implemented as Java classes that extend abstract base classes (`Component` and `Connector`) provided by the framework. Messages are encoded as objects with a string name and a string-to-object property map containing the message contents.

Certain aspects of the framework—such as threading and message queuing—are left up to individual components and connectors. Two threading policies are available to implementers: If application developers extend the base classes `Component` and `Connector`, a single application thread will service the entire application. Developers using this strategy must be careful that they do not block this thread and inadvertently hang their applications. If developers extend the base classes `ComponentThread` and `ConnectorThread`, then each component or connector gets an independent thread of control. This takes up more resources, but also reduces the possibility of inadvertent deadlock. Message queuing is handled through `Port` objects, which are objects that are capable of receiving incoming messages for a component or connector. Note that ports are a concept introduced entirely by this framework—they are not part of the C2 style itself. The framework provides only one kind of port: a first-in-first-out (FIFO) queue. With FIFO ports, messages are processed in the order received; no message is given priority over any other. Developers are free to implement their own non-FIFO ports as long as those ports extend the abstract `Port` class.

It is also interesting to note that not every constraint of the C2 style is reflected in the framework. For example, nothing in the framework enforces the rule that components

Figure 9-1.
Selected classes
and relationships
from the
Lightweight C2
Framework.

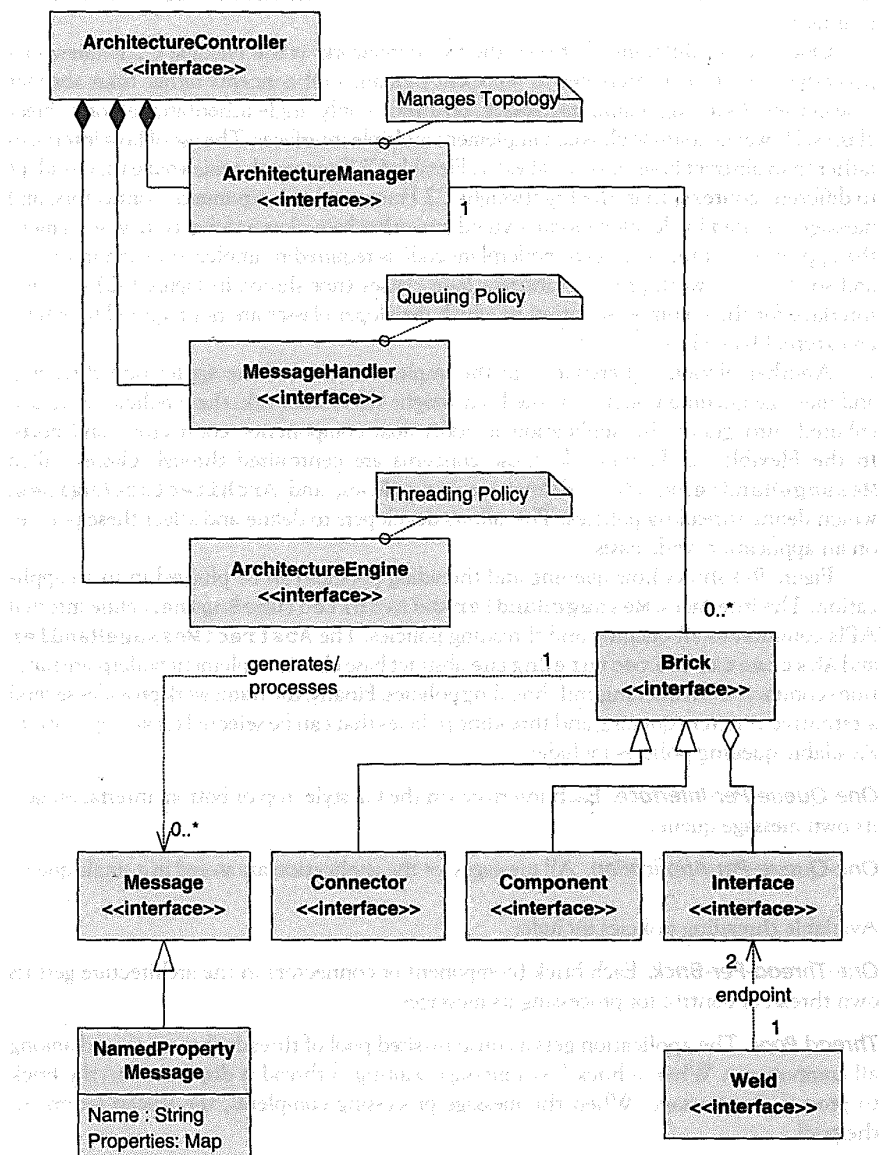


and connectors must act as if they run in separate memory spaces. In a single-process application, if a component or connector inserts a Java object reference into a message object, other components or connectors could modify this object directly without sending any messages. This would constitute illegal communication in the C2 style. However, Java does not have any support for the concept of separate in-process memory spaces, so enforcing this constraint in a framework would be prohibitively expensive. This represents a situation in which fidelity is traded for efficiency and where a framework is used to aid, but not enforce, implementations that conform to the target style.

The Flexible C2 Framework

The Flexible C2 Framework was developed later, and incorporates more aspects of the architectural style directly into the framework. As such, it is larger with seventy-three classes (approximately 8500 lines of code). Figure 9-2 shows a similar set of selected

Figure 9-2.
Selected Flexible
C2 Framework
classes and their
relationships.



classes in the Flexible C2 framework. Application developers using this framework follow a similar course of action as those using the Lightweight C2 Framework. They first implement application components and connectors as classes that implement the `Component` and `Connector` interfaces. These classes exchange data through objects that implement the `Message` interface. They then write a main class that instantiates, connects, and starts these components and connectors using the `ArchitectureController` interface.

One obvious difference between the two frameworks is the Flexible C2 Framework's pervasive use of Java interfaces to represent fundamental concepts rather than abstract base classes. As a programming language, Java allows only single inheritance among object classes. However, a single class can implement multiple interfaces. The use of Java interfaces rather than abstract base classes makes the Flexible C2 Framework somewhat easier to adapt to different contexts than the Lightweight C2 Framework. Components, connectors, and messages created by developers can extend any other base class as long as they implement the appropriate interface. Some boilerplate code is required to implement each interface, and so the framework provides abstract base classes (not shown in Figure 9-2) for each interface for the common situation in which developer classes are not required to extend an external base class.

Another obvious difference is in the implementation of the application threading and message queuing policies. In the Lightweight C2 Framework, these policies were distributed throughout the application in individual components, connectors, and ports. In the Flexible C2 Framework, these concerns are centralized through classes called `MessageHandlers`, which define queuing policies, and `ArchitectureEngines`, which define threading policies. This allows developers to define and select these policies on an application-wide basis.

Figure 9-3 shows how queuing and threading policies can be plugged in to an application. The interfaces, `MessageHandler` and `ArchitectureEngine`, define internal APIs common to all queuing and threading policies. The `AbstractMessageHandler` and `AbstractArchitectureEngine` abstract base classes implement boilerplate functions common to all queuing and threading policies. Finally, the framework provides several alternative concrete queuing and threading policies that can be selected for an application. Available queuing policies include:

One-Queue-Per-Interface. Each interface (in the C2 style, top or bottom interfaces) gets its own message queue.

One-Queue-Per-Application. All messages for the application are stored in a single queue.

Available threading policies include:

One-Thread-Per-Brick. Each brick (component or connector) in the architecture gets its own thread of control for processing its messages.

Thread Pool. The application gets a constant-sized pool of threads that are shared among all components. When a brick has a message waiting, a thread is dispatched to the brick to process the message. When the message processing completes, the thread returns to the pool.

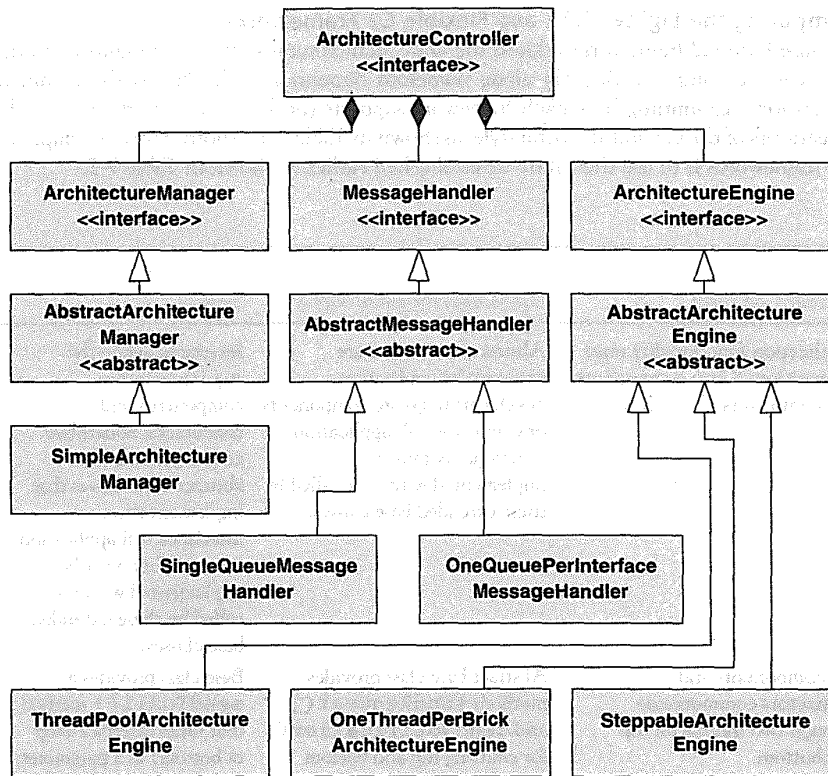


Figure 9-3.
The
implementation
of pluggable
message queuing
and threading
policies in the
Flexible C2
Framework.

Steppable Engine. A special case of the thread pool policy, the application gets one thread that is controlled by a GUI. When the user presses a ‘step’ button, the thread is dispatched to process a single message. In this way, applications can be more easily debugged.

All of these policies are allowable within the C2 style, but dramatically affect how applications are developed and how much component developers have to consider when writing code. For example, it would be relatively difficult to change a Lightweight C2 Framework application from a one-thread-per-brick to a steppable threading policy, since this would require changes to each component’s and connector’s code. In the Flexible C2 Framework, this change can be made in one line of code in the application’s main class. Still, centralized, uniform policies can make it more difficult to implement applications in which different individual components and connectors behave differently with respect to queuing and threading.

Both C2 frameworks address similar subsets of the C2-style constraints—for example, neither explicitly addresses the requirement that components and connectors are not allowed to communicate through shared memory. As explained above, this requirement is simply too expensive to implement in Java.

Comparing the Lightweight and Flexible C2 Frameworks

Because both C2 frameworks address the same architectural style and the same platform, they can be compared directly along numerous dimensions. The first is the technical dimension—examining how each framework supports (or does not support) each of the constraints of the C2 architectural style, as shown in Table 9-1. Another way of comparing the frameworks is to use the rubric we established earlier, as shown in Table 9-2.

Table 9-1
Technical
Support of C2
Architectural
Style.

C2 Style Constraint	Lightweight Framework Support	Flexible Framework Support
Application functionality must be partitioned into components and connectors.	Abstract base classes are extended by application developers to create components or connectors; all application functionality must be implemented within or called by these extended base classes.	Interfaces define the requirements for components and connectors; boilerplate code is provided in abstract base classes that implement these interfaces. All application functionality must be implemented within or called by these extended base classes.
All components and connectors communicate through two interfaces: top and bottom.	Abstract base class provides methods <code>sendRequest()</code> and <code>sendNotification()</code> for emitting top and bottom messages, respectively, and <code>handleNotification()</code> and <code>handleRequest()</code> for receiving top and bottom messages.	Base class provides a <code>sendToAll()</code> method that takes interface (top or bottom) as a parameter, and <code>handle()</code> method that is called when a message is received on any interface. Messages are tagged with the interface they arrived on.
Components and connectors should operate as if they run in their own threads of control.	Threads are created by component and connector base classes.	Threads are controlled by a central threading policy object called an ArchitectureEngine . Different engines can provide alternative threading policies.
Messages sent to the top interface of a component or connector should be received on the attached component or connector bottom(s) and vice versa.	Methods <code>sendRequest()</code> and <code>sendNotification()</code> look up connected components and connectors and directly enqueue messages in queues belonging to those elements.	Message queuing policy is centralized in an object called a MessageHandler . MessageHandlers ensure that messages get deposited in appropriate queues for processing.

C2 Style Constraint	Lightweight Framework Support	Flexible Framework Support
Components and connectors should operate as though they do not share memory; all messages exchanged must be serializable.	Message objects are structured as sets of name-value pair properties, where both names and values are strings.	All messages must implement the java.io.Serializable interface, but are not constrained to any particular format. An implementation of name-value pair set messages is also provided.
Components may be connected to at most one connector on each side; connectors may be connected to zero or more components or connectors on each side.	Connecting a component to more than one connector on any side will result in the previous connection being undone before the new one is created.	No explicit support; developers are assumed to check this constraint.
Components may make assumptions about services provided above, but no assumptions about services provided below.	No explicit support; developers are assumed to build components in a way that obeys this constraint.	No explicit support; developers are assumed to build components in a way that obeys this constraint.

Table 9-1
(Continued)

Concern	Lightweight Framework	Flexible Framework
Platform support	Java Virtual Machine on multiple platforms.	Java Virtual Machine on multiple platforms.
Fidelity	Assists developers in dealing with many C2-style constraints, but does not actively enforce them.	Assists developers in dealing with many C2-style constraints, but does not actively enforce them.
Matching assumptions	Component and connector main classes must inherit from provided abstract base classes; all communication must be through messages that consist of string names and name-value pair properties.	Component and connector main classes must implement from provided Java interfaces; all communication must be through messages which can be in any serializable format.
Efficiency	Framework is small and lightweight; can use only a single thread of control if desired, but this risks application deadlock.	Framework is larger but more flexible; can select from many queuing and threading policies to tune efficiency on an application-by-application basis.

Table 9-2
Comparison
Rubric for
Frameworks.

9.3 EXAMPLES

We have introduced many architectural alternatives for the Lunar Lander application; Chapter 4 presents a catalog of architectural styles, with Lunar Lander designed to fit the constraints of each style. Here, we will examine how architecture frameworks can be used to assist in the construction of working Lunar Lander implementations in two of those styles: pipe-and-filter and C2.

Note that the following sections will include code samples showing actual implementations of Lunar Lander. Certain good coding practices, such as the use of externalized string constants, comprehensive exception and null-value checking, and so on will be left out for simplicity's sake. Real implementations should take these practices into account as they apply equally when doing architecture-based software development.

9.3.1 Implementing Lunar Lander in the Pipe-and-Filter Style Using the `java.io` Framework

Recall the introduction of a pipe-and-filter-style Lunar Lander architecture from Chapter 4, as shown in Figure 9-4. Here, Lunar Lander is broken up into three components: the first gets the burn rate from the user, the second computes new values, and the third outputs those values back to the user. Communication among the components is one-way and is done through character streams as mandated by the style.

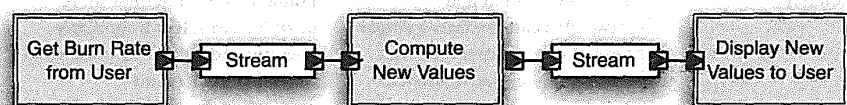
Assuming we want to implement this application in Java, we have two obvious choices for architecture frameworks: the `java.io` package and the `java.nio` package. Because the amount of data being transferred is small and we want a simple implementation, we will implement the system using `java.io`. The next choice we must make is whether to implement the system as an in-process or multiprocess system.

In-Process. In an in-process system, each component will be implemented by one or more Java classes. These will communicate by way of internal streams, leveraging internal stream classes provided by the `java.io` framework. The application configuration will need to be created in an application `main` method that we write.

Multiprocess. In a multiprocess system, each component will be implemented by one or more Java classes comprising a small application. They will communicate by way of the operating-system-provided streams `System.in` and `System.out`, and the operating system will also provide the pipe connectors. The application configuration will be done on the command line.

For simplicity, we will implement the application as a multiprocess system, which saves us from having to write additional code creating our own pipes and doing tasks such as

Figure 9-4.
Lunar Lander in
the pipe-and-filter
architectural
style.



data buffering and threading. The operating system's internal services will provide buffers among the filters as well as process-level concurrency.

The first task in implementing the Lunar Lander application is to implement the three filters depicted in the proposed architecture. Because the filters are independently composable, this can be done in any order, although the order of implementation does have practical consequences. For example, in a large application, you may want to implement stubs and skeletons for testing purposes, and whether you implement the application from left-to-right or right-to-left has implications on the sorts of stubs and skeletons you must implement. For our implementation, we will work from left-to-right, starting with the `GetBurnRate` filter.

Recall that, in a pipe-and-filter application, all data travels from left-to-right in character streams. If applications need to communicate structured data, that structure must be encoded in the character streams. Additionally, in a strict pipe-and-filter application, all user input comes from the system input stream on the left-most filter, and all output to the user console comes from the system output stream on the right-most filter.

In this application, we need to send structured data down the pipeline. We will use a simple encoding scheme: Messages will be separated by newline characters, and each message will be preceded by a control character indicating the type of message. We will preface user-output messages with a pound sign (#) and data messages with a percent sign (%). There is nothing particularly special about these characters; they are chosen arbitrarily for this example.

Figure 9-5 shows the `GetBurnRate` filter implementation for Lunar Lander. This component effectively represents the user interface of the application. First, it opens a `BufferedReader` on the system-provided class `System.in`, which the Java virtual machine connects to the operating system's input stream. `BufferedReader` is a class provided by the `java.io` package for reading structured data such as lines and integers from a character stream. Next, it enters a loop, continually prompting the user for a new burn rate, reading the value, and sending it on to the next filter.

Figure 9-6 shows the implementation of the `CalcNewValues` filter for Lunar Lander. This is the most complex of the three filters; it must read the new burn rate from the `GetBurnRate` filter, store and update the application's state, and then send output values to the final filter, which formats those values for display. The basic structure of the filter is similar to that of `GetBurnRate`—the program opens a reader on the system input stream, processes the input, and writes data to the system output stream. Here, the filter reads two kinds of data from `GetBurnRate`: user messages and burn rates. User messages are passed unchanged to the next filter; burn rates trigger a computation. When a new burn rate is read, the filter updates its internal state values—the current altitude, velocity, fuel, and time—and sends those new values to the next filter. It continues until the altitude reaches zero (or less), at which point it determines whether the lander crashed or landed successfully—either way, the game is over.

The same control character scheme is used in this filter: User messages are prefaced with pound signs and data messages are prefaced with percent signs. The data messages are further coded with a second character indicating the type of data: altitude, fuel, velocity, or time. As should be obvious by this point, ensuring that the various filters have matching assumptions about the encoding scheme is critical in constructing a working application. The encoding scheme is, in effect, the interface contract for the pipe-and-filter application.

Figure 9-5.
The
GetBurnRate
filter for Lunar
Lander.

```
//Import the java.io framework
import java.io.*;

public class GetBurnRate{
    public static void main(String[] args){
        //Send welcome message
        System.out.println("#Welcome to Lunar Lander");

        try{
            //Begin reading from System input
            BufferedReader inputReader =
                new BufferedReader(new
                    InputStreamReader(System.in));

            //Set initial burn rate to 0
            int burnRate = 0;
            do{
                //Prompt user
                System.out.println(
                    "#Enter burn rate or <0 to quit:");

                //Read user response
                try{
                    String burnRateString = inputReader.readLine();
                    burnRate = Integer.parseInt(burnRateString);

                    //Send user-supplied burn rate to next filter
                    System.out.println("%" + burnRate);
                }
                catch(NumberFormatException nfe){
                    System.out.println("#Invalid burn rate.");
                }
            }while(burnRate >= 0);
            inputReader.close();
        }
        catch(IOException ioe){
            ioe.printStackTrace();
        }
    }
}
```

Figure 9-7 shows the implementation of the *DisplayValues* filter for Lunar Lander. The structural similarities to the other two filters are again evident here—the filter reads lines from system input and writes them to system output. The purpose of this filter is simply to format data for output to the console. Unformatted user messages from previous filters are output directly, while data values are annotated with descriptions for output. Here, control characters are parsed on input, but no control characters are written to the output stream; the application assumes that it is the final filter in the application and that the output data is intended for display on a console rather than as input to another application filter.

```

import java.io.*;

public class CalcNewValues{

    public static void main(String[] args){
        //Initialize values
        final int GRAVITY = 2;
        int altitude = 1000;
        int fuel = 500;
        int velocity = 70;
        int time = 0;

        try{
            BufferedReader inputReader = new
                BufferedReader(new InputStreamReader(System.in));

            //Print initial values
            System.out.println("%a" + altitude);
            System.out.println("%F" + fuel);
            System.out.println("%v" + velocity);
            System.out.println("%t" + time);

            String inputLine = null;
            do{
                inputLine = inputReader.readLine();
                if((inputLine != null) &&
                    (inputLine.length() > 0)){

                    if(inputLine.startsWith("#")){
                        //This is a status line of text, and
                        //should be passed down the pipeline
                        System.out.println(inputLine);
                    }
                    else if(inputLine.startsWith("%")){
                        //This is an input burn rate
                        try{
                            int burnRate =
                                Integer.parseInt(inputLine.substring(1));
                            if(altitude <= 0){
                                System.out.println("#The game is over.");
                            }
                            else if(burnRate > fuel){
                                System.out.println("#Sorry, you don't" +
                                    "have that much fuel.");
                            }
                        }
                        else{
                            //Calculate new application state
                            time = time + 1;
                            altitude = altitude - velocity;
                            velocity = ((velocity + GRAVITY * 10 -
                                burnRate * 2) / 10;
                            fuel = fuel - burnRate;
                            if(altitude <= 0){
                                altitude = 0;

```

Figure 9-6.
The CalcNewValues
filter for Lunar
Lander.

Figure 9-6.
(Continued)

```

        if(velocity <= 5){
            System.out.println("#You have" +
                               "landed safely.");
        }
        else{
            System.out.println("#You have" +
                               "crashed.");
        }
    }
    //Print new values
    System.out.println("%a" + altitude);
    System.out.println("%f" + fuel);
    System.out.println("%v" + velocity);
    System.out.println("%t" + time);
}
catch(NumberFormatException nfe){
}
}
}
}while((inputLine != null) && (altitude > 0));
inputReader.close();
}
catch(IOException ioe){
    ioe.printStackTrace();
}
}
}
}

```

Together, these three filters make up the Lunar Lander application. One task remains, which is to determine how to instantiate and connect the application. This is done easily on the command line:

```
java GetBurnRate | java CalcNewValues | java DisplayValues
```

This command line invokes all three filters as separate processes. (The need for the invocation of `java` in each process is an artifact of how the Java Virtual Machine works: Each process is an instance of the virtual machine running the class passed as the first parameter to the `java` command.) Input from the console is fed to the system input stream of the left-most filter (`GetBurnRate`). The output of `GetBurnRate` is piped to the input of `CalcNewValues`. The output of `CalcNewValues` is likewise piped to the input of `DisplayValues`, and output to the console comes from the system output stream of that filter.

Reflections on the Pipe-and-Filter Lunar Lander Implementation

The architecture framework (`java.io`) in this implementation of Lunar Lander provides several useful services to the application. It includes a number of classes for reading and writing to the system input and output streams (such as `BufferedReader`). These classes allow data to be read from and written to character streams such as the system input and output streams in different ways: as individual characters, lines, integers, and so on. This allows the implementation to focus more on application functionality and less on the constraints of the architectural style.

```

import java.io.*;

public class DisplayValues{

    public static void main(String[] args){
        try{
            BufferedReader inputReader = new
                BufferedReader(new InputStreamReader(System.in));

            String inputLine = null;
            do{
                inputLine = inputReader.readLine();
                if((inputLine != null) &&
                    (inputLine.length() > 0)){
                    if(inputLine.startsWith("#")){
                        //This is a status line of text, and the
                        //should be passed down the pipeline with
                        //the pound-sign stripped off
                        System.out.println(inputLine.substring(1));
                    }
                    else if(inputLine.startsWith("%")){
                        //This is a value to display
                        if(inputLine.length() > 1){
                            try{
                                char valueType = inputLine.charAt(1);
                                int value =
                                    Integer.parseInt(inputLine.substring(2));
                                switch(valueType){
                                    case 'a':
                                        System.out.println("Altitude: " +
                                            value);
                                        break;
                                    case 'f':
                                        System.out.println("Fuel remaining: " +
                                            value);
                                        break;
                                    case 'v':
                                        System.out.println("Current Velocity: " +
                                            value);
                                        break;
                                    case 't':
                                        System.out.println("Time elapsed: " +
                                            value);
                                        break;
                                }
                            }
                            catch(NumberFormatException nfe){
                                //
                            }
                        }
                    }
                }
            }while(inputLine != null);
        }
    }
}

```

(Continued)

Figure 9-7.
The DisplayValues
filter for Lunar
Lander.

Figure 9-7.
(Continued)

```

        inputReader.close();
    }
    catch(IOException ioe){
        ioe.printStackTrace();
    }
}
}

```

The operating system itself provides the pipe connectors, as well as the concurrency policy for the architecture. The filter implementations in this system are closely aligned with the properties of the underlying framework and operating system, although this is not necessarily obvious from reading the code. For example, the filters assume that they are going to be running concurrently. In a few operating systems, such as MS-DOS, pipe-and-filter applications actually run in batch mode, collecting all the output from each filter before sending it to the next filter. If this version of Lunar Lander were to operate in batch mode, user messages sent out by the first two filters would not be output by the third filter until all the input to the first filter was completed. The application would not work properly in this circumstance. Additionally, the filters assume that lines written to the system output stream will be flushed automatically to the next filter. If this were not true, output messages would appear late and the application would operate in a broken or confusing way. These potential mismatching assumptions are subtle, but developers must be aware of them to determine whether an application will or will not operate as desired.

The implementation activity is often the time where underspecified or deficient architectures become evident. For example, the (admittedly simple) architecture for pipe-and-filter Lunar Lander specifies what data should pass from filter to filter, but not the format of that data. A component that communicated using XML-based data encoding, while being conformant with the architecture, would not interoperate with the filter components implemented above. Clearly, this architecture has not been elaborated to a point where component interoperability can be inferred from the architecture alone. A mismatch between XML and line-oriented data formats would be easily caught during system integration, but more subtle bugs can be even more dangerous. For example, in the case of the Mars Climate Orbiter, an interface mismatch between metric and imperial units of measure was a substantial cause for the loss of the orbiter.

9.3.2 Implementing Lunar Lander in the C2-Style Using the Lightweight C2 Framework

A C2-style Lunar Lander architecture shown in Figure 9-8 is different from the pipe-and-filter Lunar Lander version. First, the application functionality is broken up differently. Here, a game state component retains all game state and broadcasts updates to other components. A game logic component reads the game state, calculates a new state, and updates the game state component with that state. The GUI component is responsible for reading new burn rate values from the user and keeping the user informed as to the current game state. The most significant departure from the pipe-and-filter version is the addition of a clock component, which emits "tick" events, or messages, at periodic intervals. This changes the character of the game substantially; instead of waiting indefinitely for a new

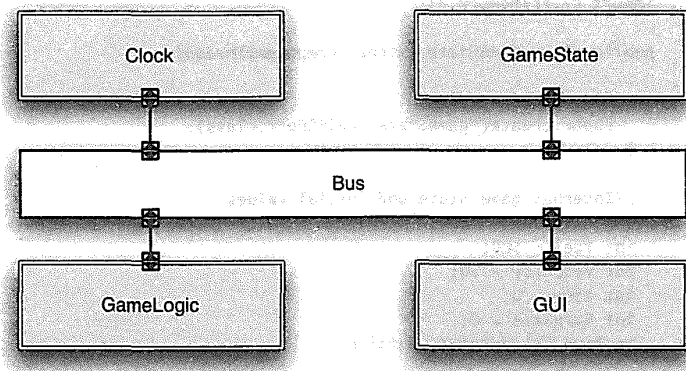


Figure 9-8.
Lunar Lander in
the C2
architectural
style.

burn rate value from the user as in the pipe-and-filter version, this version of Lunar Lander is played in real-time. In this version, the game state changes whenever the clock ticks, and not when new burn rate values are entered. Here, the user may update the current burn rate as often as desired between ticks. When the clock ticks, the current burn rate value is used to calculate the new game state.

Many C2 frameworks exist for different programming languages and platforms. Assuming we want to implement the system in Java, we still must choose between frameworks such as the Lightweight C2 Framework and the Flexible C2 Framework. Because it results in slightly simpler component code, we will use the Lightweight C2 Framework for this implementation.

As with the pipe-and-filter example, we can implement these components in any order, but the order we choose has practical consequences. Because of the substrate independence (that is, layering) rules in C2, lower components may make assumptions about the services provided by upper components, but upper components may not make assumptions about lower components. (Note that this is reversed from traditional layered or virtual-machine depictions where upper layers depend on lower layers.) This means that the topmost components have no dependencies, and that lower components are progressively more dependent. We will take a least-dependent-first implementation strategy,² which means that the topmost components get implemented first.

The code for the GameState component is shown in Figure 9-9. The first line of code imports the `c2.framework` package, which belongs to the Lightweight C2 Framework. The class declaration shows that the GameState class extends the `ComponentThread` base class. This base class is extended by all component classes that run in their own thread of control (as is the norm in C2-style systems). This base class provides the code required for the component to receive and send requests and notifications, as well as the threading and synchronization code required to coordinate with the other components and connectors.

²This is ordinarily called a bottom-up implementation strategy, but this terminology can be confusing in C2 architectures, where the least-dependent components are represented at the top in the canonical graphical visualization.

Figure 9-9.
The
GameState
component in
C2.

```
import c2.framework.*;

public class GameState extends ComponentThread{

    public GameState(){
        super.create("gameState", FIFOPort.class);
    }

    //Internal game state and initial values
    int altitude = 1000;
    int fuel = 500;
    int velocity = 70;
    int time = 0;
    int burnRate = 0;
    boolean landedSafely = false;

    protected void handle(Request r){
        if(r.name().equals("updateGameState")){
            //Update the internal game state
            if(r.hasParameter("altitude")){
                this.altitude =
                    ((Integer)r.getParameter("altitude")).intValue();
            }
            if(r.hasParameter("fuel")){
                this.fuel =
                    ((Integer)r.getParameter("fuel")).intValue();
            }
            if(r.hasParameter("velocity")){
                this.velocity =
                    ((Integer)r.getParameter("velocity")).intValue();
            }
            if(r.hasParameter("time")){
                this.time =
                    ((Integer)r.getParameter("time")).intValue();
            }
            if(r.hasParameter("burnRate")){
                this.burnRate =
                    ((Integer)r.getParameter("burnRate")).intValue();
            }
            if(r.hasParameter("landedSafely")){
                this.landedSafely =
                    ((Boolean)r.getParameter("landedSafely"))
                    .booleanValue();
            }
            //Send out the updated game state
            Notification n = createStateNotification();
            send(n);
        }
        else if(r.name().equals("getGameState")){
            //If a component requests the game state
            //without updating it, send out the state
            Notification n = createStateNotification();
            send(n);
        }
    }
}
```

```

    }
}

protected Notification createStateNotification(){
    //Create a new notification comprising the
    //current game state

    Notification n = new Notification("gameState");
    n.addParameter("altitude", altitude);
    n.addParameter("fuel", fuel);
    n.addParameter("velocity", velocity);
    n.addParameter("time", time);
    n.addParameter("burnRate", burnRate);
    n.addParameter("landedSafely", landedSafely);
    return n;
}

protected void handle(Notification n){
    //This component does not handle notifications
}
}

```

Figure 9-9.
(Continued)

The first method, `GameState()`, is a simple boilerplate constructor. This C2 framework requires that each component and connector be given a name (in this case, `gameState`), as well as the class that will implement the ports (that is, message queues) used to exchange messages with attached connectors. For simplicity, all components and connectors in this architecture will use FIFO (first-in, first-out) ports—that is, ordinary queues.

The next block of code declares a set of member variables that represent the game state, including velocity, altitude, fuel remaining, current burn rate, whether the lander has landed safely, and so on. These will be read and updated as the game is played.

Each component and connector in the Lightweight C2 Framework has two primary responsibilities: handling requests (messages traveling upward in the architecture and arriving on the bottom port) and handling notifications (messages traveling downward in the architecture and arriving on the top port).

The next method, `handleRequest()`, is called automatically by the framework when a request arrives for this component. Recall that in the Lightweight C2 Framework, all requests and notifications have the same structure: a character string name, plus a set of name-value pair properties. This method handles two requests: an `updateGameState` request and a `getGameState` request. Upon receiving an `updateGameState` request, the component reads from the property set various new state values corresponding to elements of the game state. After updating the game state with the new values, the component always creates and emits a new `gameState` notification containing all the updated state values. Upon receiving a `getGameState` request, the component simply creates a new game state notification with current state values and sends it out. Sending out a request or a notification in the Lightweight C2 framework is simple. First, a new `Request` or `Notification` object is created, given a name, and populated with properties. This object can be passed to a `send()` method present in the abstract base class, in this case, `ComponentThread`. The framework routes the message appropriately.

The `handleNotification()` method for this component is empty; this component does not handle notifications. Because we are aware, from an architectural perspective, that no components will be connected above this one, we know that it will not receive any.

This component is entirely reactive. It is idle until a request arrives on its bottom port. If the request is of a recognized type (`updateGameState` or `getGameState`), then the component reacts, either updating its internal state and sending out a new state notification, or simply sending out the current game state. This is a relatively typical pattern of interaction for state components in C2-style systems. It is also worth noting that this component acts entirely as a data store—data validation and processing is done in other components (primarily the `GameLogic` component). Maintaining this separation allows architects to more easily swap out different data structures or game logic components.

The code for the `Clock` component is shown in Figure 9-10. The basic scaffolding for this component is similar to that for the `GameState` component—this component also

Figure 9-10.
The `Clock`
component in
C2.

```
import c2.framework.*;

public class Clock extends ComponentThread{
    public Clock(){
        super.create("clock", FIFOPort.class);
    }

    public void start(){
        super.start();

        Thread clockThread = new Thread(){
            public void run(){
                //Repeat while the application runs
                while(true){
                    //Wait for five seconds
                    try{
                        Thread.sleep(5000);
                    } catch (InterruptedException ie){}

                    //Send out a tick notification
                    Notification n = new Notification("clockTick");
                    send(n);
                }
            };
        };
        clockThread.start();
    }

    protected void handle(Notification n){
        //This component does not handle notifications
    }

    protected void handle(Request r){
        //This component does not handle requests
    }
}
```

extends `ComponentThread` and includes the same boilerplate constructor. Unlike the `GameState` component, however, the `Clock` handles neither requests nor notifications. Its job is simply to emit tick notifications at a predefined interval. This is done through the creation of a new clock thread in the component's `start()` method. The `start()` method is another distinguished method in the Lightweight C2 Framework; it is called by the framework automatically when the application starts. In this implementation, the clock thread creates and emits a new tick notification every five seconds (5000 milliseconds). The use of a separate clock thread is needed because the component's internal thread (provided by the `ComponentThread` base class) is used only for handling notifications and requests—attempting to co-opt it for sending out ticks would interfere with the message-handling behavior of the component.

The code for the `GameLogic` component is shown in Figure 9-11. This component is the most complex of the C2 Lunar Lander components. Structurally, it is very similar to the `GameState` component. Instead of responding to requests from lower components, however, it reacts to notifications coming from upper components. `GameLogic` responds to two kinds of notifications. The first is a `gameState` notification from the `GameState` component. Whenever the `GameLogic` component is notified that the game's state has changed, it updates internal state values that are used for later calculation. For a simple game like Lunar Lander, where nearly all the game state is used by the single `GameLogic` component, keeping separate copies of the game state in the `GameState` and `GameLogic` components may seem redundant. In more complex applications, however, logic components rarely need all the game state; instead, they would retain only the parts of the state necessary to do their own computations. An additional question that might occur to developers familiar with procedural or object-oriented programming is why the `GameLogic` component does not simply query the `GameState` component for data when it is needed. The answer lies in the architectural style—in C2, such synchronous component-to-component queries are not allowed.

The second notification handled by the `GameLogic` component is a clock tick from the `Clock` component. Recall that this version of Lunar Lander is driven not by user input of new burn rates, but by the tick of the real-time clock. When a clock tick occurs, the latest game state stored in the `GameLogic` component is used to calculate the next state—burning some amount of fuel, descending (or ascending) a certain distance, and so on. This state is then sent to the `GameState` component in an `updateGameState` request, which we saw handled by that component, above.

One additional detail in the `GameLogic` implementation is the `start()` method. On startup, the `GameLogic` component sends an asynchronous request upward to the `GameState` component for the initial game state. Without doing so, the calculation that occurs on the first clock tick might be based on incorrect (that is, all-zero) values as initialized in the `GameLogic` component.

Effectively, the behavior of the `GameLogic` component can be summed up by the statechart in Figure 9-12. The component starts and sends a `getGameState` request upward. It then idles, waiting for notifications. When a `GameState` notification is received, the internal state of the component is updated. When a `clockTick` notification is received, a new game state is calculated and a request to update the game state is sent upward.

The code for the GUI component of the Lunar Lander is shown in Figure 9-13. This component handles all interaction with the user. This particular implementation uses

Figure 9-11.
The
GameLogic
component in
C2.

```
import c2.framework.*;

public class GameLogic extends ComponentThread{
    public GameLogic(){
        super.create("gameLogic", FIFOPort.class);
    }

    //Game constants
    final int GRAVITY = 2;

    //Internal state values for computation
    int altitude = 0;
    int fuel = 0;
    int velocity = 0;
    int time = 0;
    int burnRate = 0;

    public void start(){
        super.start();
        Request r = new Request("getGameState");
        send(r);
    }

    protected void handle(Notification n){
        if(n.name().equals("gameState")){
            if(n.hasParameter("altitude")){
                this.altitude =
                    ((Integer)n.getParameter("altitude")).intValue();
            }
            if(n.hasParameter("fuel")){
                this.fuel =
                    ((Integer)n.getParameter("fuel")).intValue();
            }
            if(n.hasParameter("velocity")){
                this.velocity =
                    ((Integer)n.getParameter("velocity")).intValue();
            }
            if(n.hasParameter("time")){
                this.time =
                    ((Integer)n.getParameter("time")).intValue();
            }
            if(n.hasParameter("burnRate")){
                this.burnRate =
                    ((Integer)n.getParameter("burnRate")).intValue();
            }
        }
        else if(n.name().equals("clockTick")){
            //Calculate new lander state values
            int actualBurnRate = burnRate;
            if(actualBurnRate > fuel){
                //Ensure we don't burn more fuel than we have
                actualBurnRate = fuel;
            }

            time = time + 1;
        }
    }
}
```

```

altitude = altitude - velocity;
velocity = ((velocity + GRAVITY) * 10 -
    actualBurnRate * 2) / 10;
fuel = fuel - actualBurnRate;

//Determine if we landed (safely)
boolean landedSafely = false;
if(altitude <= 0){
    altitude = 0;
    if(velocity <= 5){
        landedSafely = true;
    }
}

Request r = new Request("updateGameState");
r.addParameter("time", time);
r.addParameter("altitude", altitude);
r.addParameter("velocity", velocity);
r.addParameter("fuel", fuel);
r.addParameter("landedSafely", landedSafely);
send(r);
}

protected void handle(Request r){
    //This component does not handle requests
}
}

```

Figure 9-11.
(Continued)

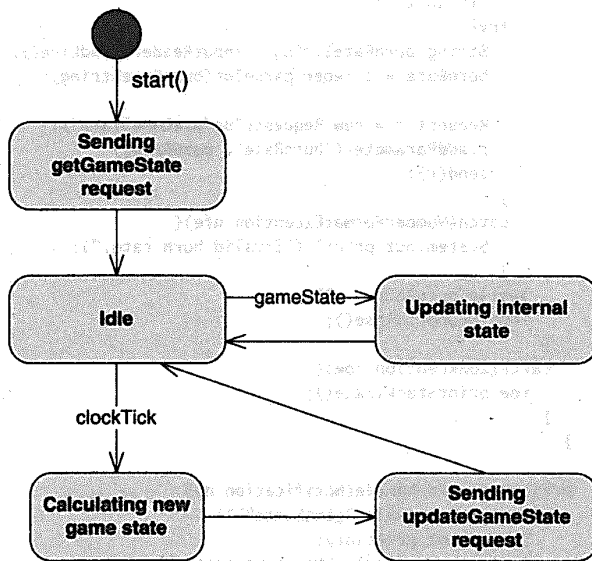


Figure 9-12.
Statechart
describing the
behavior of the
GameLogic
component in
C2.

Figure 9-13.
The GUI
component in
C2.

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import c2.framework.*;

public class GUI extends ComponentThread {
    public GUI() {
        super.create("gui", FIFOPort.class);
    }

    public void start() {
        super.start();
        Thread t = new Thread() {
            public void run() {
                processInput();
            }
        };
        t.start();
    }

    public void processInput() {
        System.out.println("Welcome to Lunar Lander");
        try {
            BufferedReader inputReader = new BufferedReader(
                new InputStreamReader(System.in));

            int burnRate = 0;
            do {
                System.out.println("Enter burn rate or <0" +
                    "to quit:");
                try {
                    String burnRateString = inputReader.readLine();
                    burnRate = Integer.parseInt(burnRateString);

                    Request r = new Request("updateGameState");
                    r.addParameter("burnRate", burnRate);
                    send(r);
                } catch (NumberFormatException nfe) {
                    System.out.println("Invalid burn rate.");
                }
            } while (burnRate >= 0);
            inputReader.close();
        } catch (IOException ioe) {
            ioe.printStackTrace();
        }
    }

    protected void handle(Notification n) {
        if (n.name().equals("gameState")) {
            System.out.println();
            System.out.println("New game state:");
        }
    }
}
```

```

        if(n.hasParameter("altitude")){
            System.out.println(" Altitude: " +
                n.getParameter("altitude"));
        }
        if(n.hasParameter("fuel")){
            System.out.println(" Fuel: " +
                n.getParameter("fuel"));
        }
        if(n.hasParameter("velocity")){
            System.out.println(" Velocity: " +
                n.getParameter("velocity"));
        }
        if(n.hasParameter("time")){
            System.out.println(" Time: " +
                n.getParameter("time"));
        }
        if(n.hasParameter("burnRate")){
            System.out.println(" Current burn rate: " +
                n.getParameter("burnRate"));
        }

        if(n.hasParameter("altitude")){
            int altitude =
                ((Integer)n.getParameter("altitude")).intValue();
            if(altitude <= 0){
                boolean landedSafely =
                    ((Boolean)n.getParameter("landedSafely"))
                        .booleanValue();
                if(landedSafely){
                    System.out.println("You have landed safely.");
                }
                else{
                    System.out.println("You have crashed.");
                }
                System.exit(0);
            }
        }
    }

    protected void handle(Request r){
        //This component does not handle requests
    }
}

```

Figure 9-13.
(Continued)

simple console (that is, text)-based input and output routines. The GUI component has two primary responsibilities. First, it creates an independent thread for reading user input—in this case, burn rates—from the console. Whenever a new burn rate is read, it is wrapped in an `updateGameState` request and sent upward. Again, a separate thread is needed here to avoid interference with the message processing thread that belongs to the component. Second, the GUI component listens for `gameState` notifications. Upon receiving an updated game state, the component formats and writes the state to the console.

Now that all the components have been coded, they must be instantiated and connected. This is done by way of a main bootstrapping program shown in Figure 9-14. In the pipe-and-filter example, we were able to use the command-line shell to instantiate and connect the components. For a C2 application, we must instead write this bootstrapping program ourselves, calling upon the services of the Lightweight C2 Framework to instantiate and connect the components. The bootstrapper itself is relatively straightforward. First, an *Architecture* is created, the C2 framework's object that represents an architectural structure. Then, instances of each of the components are created, along with a single connector called *bus*. The components and connectors are added to the *Architecture*, and then links among them (called *Welds* in C2 parlance) are created. With everything connected, the *Architecture*'s *start()* method is called, which creates internal threads, calls each component and connector's individual *start()* method, and performs other tasks needed to start the application.

Figure 9-14.
The Lunar
Lander main
program in C2.

```
import c2.framework.*;

public class LunarLander{

    public static void main(String[] args){
        //Create the Lunar Lander architecture
        Architecture lunarLander = new
            SimpleArchitecture("LunarLander");

        //Create the components
        Component clock = new Clock();
        Component gameState = new GameState();
        Component gameLogic = new GameLogic();
        Component gui = new GUI();

        //Create the connectors
        Connector bus = new ConnectorThread("bus");

        //Add the components and connectors to the architecture
        lunarLander.addComponent(clock);
        lunarLander.addComponent(gameState);
        lunarLander.addComponent(gameLogic);
        lunarLander.addComponent(gui);

        lunarLander.addConnector(bus);

        //Create the welds (links) between components and
        //connectors
        lunarLander.weld(clock, bus);
        lunarLander.weld(gameState, bus);
        lunarLander.weld(bus, gameLogic);
        lunarLander.weld(bus, gui);

        //Start the application
        lunarLander.start();
    }
}
```

9.4 END MATTER

It is imperative that, to the extent possible, the design decisions in the architecture are reflected in the implemented system. Conflicts or mismatches are the hallmarks of architectural drift and erosion. Sometimes these divergences are obvious and known to the system's stakeholders, and additional documentation of these cases can help to mitigate the risks or provide plans for future work to bring the architecture and implementation back in sync. Sometimes, however, they are not, and stakeholders remain unaware of the problem. This is usually compounded in system maintenance phases, when it is expedient to update the implementation without going back and updating architectural models to match. In some ways, it is more harmful to have conflicting information in the architecture and the implementation than to have an underspecified architecture, since at least an underspecified architecture will not mislead stakeholders.

Maintaining a consistent architecture-to-implementation mapping is almost never easy, and various techniques can be used to do so. The strongest mappings are possible when architectural models become *part of* the system implementation and are closely connected to implementation artifacts by way of explicit mappings embedded in the models to elements implemented in architecture implementation frameworks. This strategy works well for concrete design decisions, such as structural design decisions, but is more difficult for more abstract design decisions concerning, for example, non-functional properties. In this case, stakeholders must negotiate and make decisions about how they will convince themselves that their architecture is reflected adequately in their implementation. Doubtlessly, this will involve a combined spectrum of strategies from peer review to traceability links between documents to the use of architecture implementation frameworks.

In the past set of chapters, we focused on design activities—architectural design, modeling, visualization, and analysis. In this chapter, we discussed how to move from design to implementation. Once the system is implemented, it must be deployed to its users. This is the focus of the next chapter. As we will see, maintaining architectural models with strong mappings to implementations are useful in post-implementation activities as well.

In general, a good design should result in a good system. The way to maximize your chances of getting a good system out of a good design is to use careful practices for mapping your architecture to your code. A small investment in a good, reusable architecture framework can instill consistency throughout a project and over its lifetime.

Generation technologies are especially powerful in reducing costs and minimizing architectural drift and erosion. While it may not be a silver bullet, generating even partial implementations from architecture is always vastly superior to making the connection manually, especially if round-trip software engineering is in place. Every line of code written by a *program* rather than a *programmer* is money in the bank.

Maintenance costs are the highest in any successful software engineering project (more than 50 percent). These costs are exacerbated by spaghetti-code, post hoc changes that (often unknowingly) violate design principles set out at the beginning of the project, and lack of clear understanding of the system. When design principles are violated, the software qualities anticipated and imbued by the design may not manifest themselves in the final, implemented system.

The Business Case

Good architecture-to-implementation mappings help to mitigate risks, especially those due to personnel turnover. Turnover is inevitable in any development effort, and a well-defined architecture with a corresponding implementation will dramatically improve the ability of new personnel to be effective on an old project.

9.5 REVIEW QUESTIONS

1. What are some architectural concerns that can be mapped to implemented systems? What strategies can be used to map these concerns?
2. What is the difference between one-way and round-trip mapping?
3. What is an architecture-implementation framework? How does an architecture-implementation framework differ from middleware?
4. What is the relationship between an architecture-implementation framework and an architectural style?
5. When might multiple frameworks be developed for a single architectural style?
6. What are some criteria that can be used to evaluate architecture implementation frameworks?
7. How do middleware and component frameworks induce architectural styles?
8. What are some strategies for resolving mismatches between architectural styles and middleware?
9. When should a new architecture-implementation framework be developed? What criteria or strategies should be used in developing the new framework?
10. What kinds of generative approaches can assist in moving from architecture to implementation?
11. Enumerate some existing architecture implementation frameworks.
12. How is the Standard I/O package an architecture framework for pipe-and-filter systems? How does it support the various rules and constraints of the pipe-and-filter style?

9.6 EXERCISES

1. Run the implemented Lunar Lander applications in this chapter through a debugger and examine how control and data are exchanged through the framework.
2. The chapter compares and contrasts the Lightweight C2 Framework and the Flexible C2 Framework in terms of both structure and how they support the rules of the C2 style. Perform the same comparison for the java.io and java.nio packages vis-à-vis the pipe-and-filter style.
3. Choose an architecture implementation framework and a simple application, and implement that application atop the framework. Reflect on how you maintained the constraints of the architectural style in your implementation, and how the framework assisted (or hindered) you in doing so.
4. Choose one of the simple architectural styles in Chapter 4 and construct an architecture-implementation framework for your preferred platform/operating system combination. Enumerate the rules of the style and how your framework does or does not support those rules.
5. Construct an architecture framework as suggested in Exercise 4, and then use your framework for Exercise 3.
6. Learn about one or more middleware technologies, such as CORBA, COM, RMI, JavaBeans, and so on. Identify the architectural style rules imposed by the middleware platform. Find an example application built atop the middleware and see whether or not it obeys the style rules.

9.7 FURTHER READING

This chapter looks toward architecture-implementation frameworks as a primary method of mapping architectural design decisions to implementation artifacts. Surprisingly little has been written on the subject of such frameworks characterized in this way; Sam Malek et al. (Malek, Mikic-Rakic, and Medvidović 2005) is a notable exception. However, many software systems exist that closely resemble architecture implementation frameworks without the explicit focus on styles. Some of these, such as the Standard I/O framework in C (Kernighan and Ritchie 1988) and the Java I/O (Harold 2006) and New I/O (Hitchens 2002) packages are architecture implementation frameworks in disguise, providing the services of a framework without explicitly being identified as such.

Middleware such as CORBA (Object Management Group 2001), COM and its variants (Sessions 1997),

JavaBeans (JavaSoft 1996), Java RMI (Grosso 2001), Java Message Service implementations (JavaSoft 2001), and other message-passing systems such as MQSeries (IBM 2003) and MSMQ (Houston 1998) are often used as architecture-implementation frameworks. However, Elisabetta Di Nitto and David Rosenblum (Di Nitto and Rosenblum 1999) insightfully called out the fact that middleware *induces* an architectural style on applications that use it.

A recent trend growing in popularity is the extensive use of generative techniques, particularly under the banner of Model-Driven Architecture (Mukerji and Miller 2003), which can generate whole or partial implementations through models. Generative approaches have been identified as silver bullets before, and time will tell whether Model-Driven Architecture lives up to its initial promise.

Deployment and Mobility

After a software system has been designed, implemented, and validated, it is ready for operation. That usually requires that the system's components and connectors first be distributed to the "target" hardware processors. A software system cannot fulfill its purpose until it is deployed, that is, until its executable modules are physically placed on the hardware devices on which they are supposed to run. The outcome of the activity of placing a system's software components on its hardware hosts is the *deployment* of the system's architecture.

Once the system is in operation, it is possible, and often necessary, to change the physical location of its hardware hosts. For example, laptop computers, personal digital assistants (PDAs), cellular telephones, software-controlled radios, and computers embedded in a vehicle, all move regularly while staying connected. More challenging, more interesting to a software engineer (and even a user), and more pertinent to this book is the relocation or migration of a *software* component or connector from one hardware host to another. This may need to be done to improve the system's performance, perhaps by collocating a processing component with the data it needs, lessen the computational load on a given host, or to achieve some other property. Relocating software modules in this manner changes the deployment view of the software system's architecture during the system's run time, and is referred to as *migration* or *redployment*. Run time system migration (or redeployment) is thus a type of a software system's *mobility*.

It should be noted that changing a system's deployment while it is running in many ways entails a superset of concerns that engineers face during initial deployment, including the following.

- During a component's migration, its run time state may need to be preserved and migrated. On the other hand, during initial deployment, prior to system start-up, components are typically stateless.

- Systems experience temporary downtimes, or at least degradations in provided capabilities and performance, during the migration process. These concerns do not affect the system during its initial deployment.
- The time at which a component is migrated during execution must be chosen very carefully, as the component must not be in the middle of computation or interaction with another component. Again, these concerns are not applicable during the system's initial deployment.
- The added complexity of run time redeployment, indicated in the above points, is coupled with the usually significantly reduced amount of time available to ensure that all critical system properties have been preserved. In contrast, engineers can carefully plan and analyze a system's initial deployment over a comparatively much longer period of time.

Fundamentally, the role of hardware in the context of deployment and mobility is to support a system's software architecture, including the functionality embodied in processing components, the information exchanged via data components, the interactions facilitated by connectors, and the overall structure defined by the configuration. The hardware configuration can, in turn, also present constraints that must be supported by the software architecture: The choice of distribution points will induce certain architectural decisions. The deployment view of a software system's architecture can be critical in assessing whether the system will be able to satisfy its requirements. For example, placing many large components on a small device with limited memory and CPU power, or transferring high volumes of data over a network link with low bandwidth will negatively impact the system, much like incorrectly implementing its functionality will.

To illustrate these issues, consider the configuration of hardware devices shown in Figure 10-1. This configuration is typical of those used in wireless sensor network systems those found in many commercial buildings, power plants, and transportation systems. The sensor devices host software that can help determine the conditions of their outside environment, such as motion, vibration, fire, and moisture. Sensors are typically highly resource-constrained and can only perform minimal amounts of computation and data storage. The gateway devices aggregate, process, and can possibly share this information. They pass it on to the hubs, which may run software that can make appropriate decisions in the case of certain events or changes in system status. For example, if a gateway fails, a hub may instruct another gateway to take over the management of the "orphaned" sensors; likewise, if multiple sensors report events of a given type (such as excess moisture), the hub may decide that it is appropriate to sound an alarm. Both the gateways and hubs have significantly higher capacities than

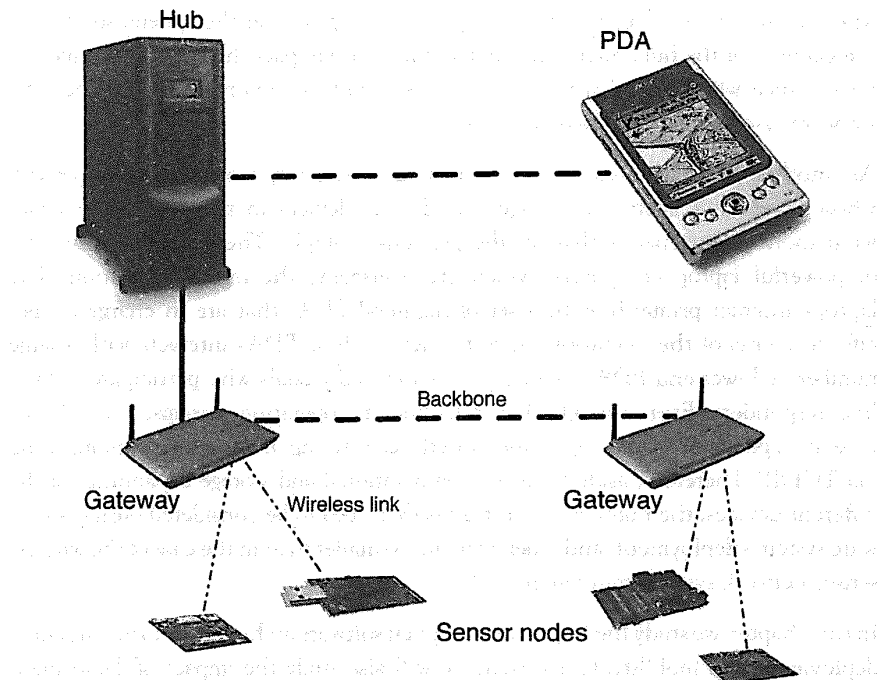


Figure 10-1.
A wireless sensor
network system
comprising
hardware devices
of several types,
on which a
software system
is deployed.

the sensors, and may be able to perform large amounts of computation and/or store large amounts of data. Finally, humans can observe the system's operation via PDAs, which communicate with the hubs. PDAs are usually more capacious than sensors, but not as capacious as the hubs or the gateways. The four types of devices may all run different operating systems and other system-level software, require different dialects of programming languages, and support different network protocols.

Software engineers must take into account information such as the above when deciding how to deploy the software system onto the requisite hardware hosts. Furthermore, this information will directly impact the options an engineer has for redeploying the system's components—and possibly connectors—during run time. The idiosyncrasies of a given platform will thus serve as software deployment and mobility constraints.

In addition to considering the characteristics of the involved hardware devices, certain application-level requirements may have a significant impact on the given software architecture's deployment and mobility. For example, the maximum allowed round-trip time between the reporting of an event by a sensor and the acknowledgment that the event has been received by an upstream processing component may affect

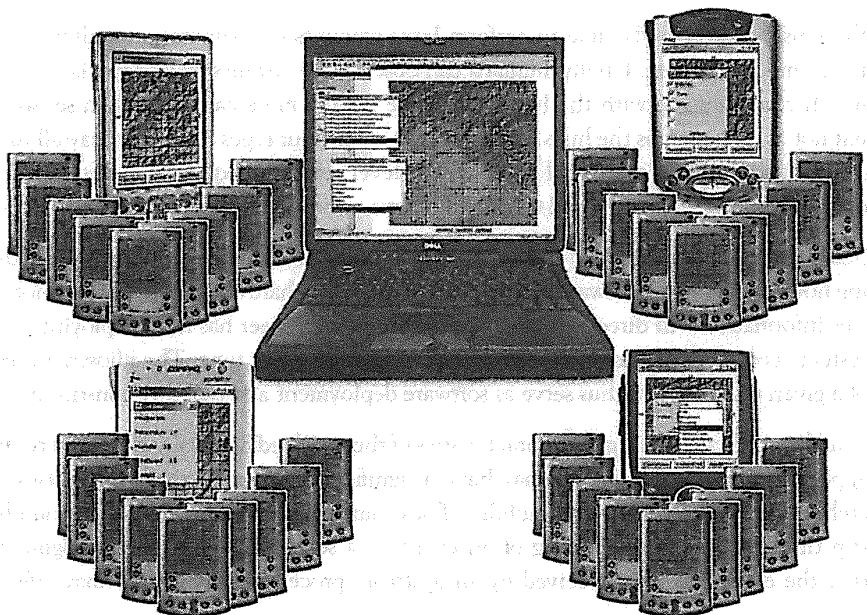
the system's overall performance. For example, if the system is deployed in a remote location, the round-trip time may be significantly longer than if the system is deployed in a local location. This may affect the system's ability to respond to events in a timely manner.

where certain processing and data components are placed in the system, such as on the gateway or the hub. Such a requirement may also impact the flow of information, for instance whether and when human users need to be informed versus when the software system itself can make a decision.

As another illustrative example, consider an emergency response system (ERS), whose screenshot is shown in Figure 10-2. The devices in this system are somewhat more homogeneous than in the previous example: There is a small number of powerful laptop computers, which are overseeing the overall operation. The laptops interact primarily with a set of high-end PDAs that are in charge of specific segments of the operation; in turn, each of these PDAs interacts with a large number of lower-end PDAs, which are used by individuals who participate as first-line responders. Even though they run different operating systems, each device type is capable of displaying a user interface, running Java, and communicating via TCP/IP. Therefore, aside from the computational and storage capabilities of the different devices, the number of constraints that need to be considered during a software system's deployment, and redeployment, is smaller than in the case of the wireless sensor network system from Figure 10-1.

In this chapter, we study the impact of an explicit software architectural focus on system deployment and mobility. Conversely, we will also study the impact of deployment

Figure 10-2.
An instance of the
family of emer-
gency response
systems (ERS),
which help with
deploying and
organizing teams
of humans in
cases of natural
disasters, search-
and-rescue
operations, and
military crises
(© IEEE 2005).



and mobility on software architecture. The reader should note that deployment can be viewed as a special case of mobility, that is, as the mobility of software modules *prior* to the system's run time. The two concepts are thus closely related and many of their resulting challenges, ramifications, techniques, and tools are similar.

The objective of this chapter is to define and, where necessary, clarify the role of software architecture in deployment and mobility, to discuss different approaches to software deployment and mobility, and to present a set of deployment and mobility techniques that are at an engineer's disposal. We illustrate the main points in the discussion via examples from existing solutions. While software deployment and mobility are two important and growing areas of study, our focus here is specifically on their architecturally relevant aspects. An annotated list of references will be given at the end of the chapter for a broader treatment of the two areas.

10 Deployment and Mobility

10.1 Overview of Deployment and Mobility Challenges

10.2 Software Architecture and Deployment

10.2.1 Basic Concepts

10.2.2 Deployment Activities

10.2.3 Tool Support

10.3 Software Architecture and Mobility

10.3.1 Basic Concepts

10.3.2 Mobility Paradigms

10.3.3 Challenges in Migrating Code

10.4 End Matter

10.5 Review Questions

10.6 Exercises

10.7 Further Reading

Outline of Chapter 10

10.1 OVERVIEW OF DEPLOYMENT AND MOBILITY CHALLENGES

Modern software systems can present a number of deployment and mobility challenges. Several of these are outlined below.

1. The target processors may be geographically widely distributed, sometimes even throughout the solar system, as is the case with some of NASA's space missions. Physically deploying the software in such settings presents logistical problems, especially if the software has to be distributed and deployed during the system's execution. For example, it may take minutes, and even hours, for the software to reach the intended target host. An accompanying concern is the security of such systems, especially during the transfer of code.
2. The target processors may be embedded inside heterogeneous devices that have different operating environments and serve different purposes. For example, aircraft, mobile robots, consumer electronic devices, mobile phones, and desktop computers present very different characteristics. A software component running on one device may not be able to run on another. Therefore, transferring a component

from one such device to another may require use of sophisticated adaptor software connectors (recall Chapter 5), thus potentially affecting the system's overall performance. Such redeployments may often be impossible.

3. Different software components may require different hardware configurations for their successful execution, whether screen resolution, CPU speed, I/O devices, or memory. Again, this requires careful planning and analysis of the intended deployment profiles or run time software migrations.
4. Typical system life spans may stretch over decades and require periodic maintenance, meaning usually that parts of the system may need to be redeployed. For example, a "buggy" component may be replaced or a more reliable connector introduced. This means that redeployment is an unavoidable activity in most software systems. It also means that, as existing components exhibit problematic behaviors, engineers may resort to migrating software to improve system performance.
5. Similarly, the deployed system is likely to evolve over time, again requiring redeployment. New functionality may be introduced and individual components or even entire hosts may be replaced with newer versions. The danger in this, as well as in the previous case, is that the deployed system's architecture will degrade.
6. The emerging class of mobile code solutions, such as mobile agents (Fuggetta, Picco, and Vigna 1998), require that *running* components be redeployed from one host to another. This requires carefully assessing acceptable (partial) system downtimes and employing techniques for capturing and transferring the relevant portion of the system's *dynamic state* in addition to the code.
7. After a component has been relocated, it still must be able to discover and access, from its new location, the system services it needs at run time. Likewise, the rest of the system needs to be able to locate and access the services provided by that component. Selecting existing or developing new mechanisms for ensuring continuous access to system services is a major architectural consideration in mobile systems.

Traditionally the problem of mobility, and especially initial system deployment, is handled in a relatively uniform manner, and it does not always reflect the above scenarios. For example, if a Windows PC user wants to upgrade an operating system or application on his or her PC, or install a patch that will remove an existing problem, he either will obtain a CD-ROM with the needed software or go to a Web site and download the software; the user will then have to shut down all applications running on the PC in order to complete the procedure. While the PC user is controlling the deployment process, this human user-in-the-loop approach relies on certain assumptions, such as the fact that a computer's operating environment is one of a small handful of tightly controlled environments supported by the provider of new functionality. Regardless of a PC user's technical prowess, the user usually will have little insight into the changes done to the code running on the PC. In other words, the precondition for, say, installing a new spell-check component into a word processor is that its user place full trust in the component's developer, hope that the new software will work correctly, that it will eliminate any problems the user may have had with the previous spell-checker, and that it will not introduce any new bugs. As we all know

from experience, sometimes these patches fix the problems, and sometimes they do not, but eventually most PC users reach the point where the performance of the computer has degraded so badly that the only true remedy is to “reinstall the PC” completely.¹

The situation is even more extreme if a commodity device, such as an automobile, cellular telephone, or “smart” cable TV box, begins misbehaving. In such cases, chances are that the owner or user will have to rely on a trained professional for a remedy. That will often require relinquishing control of the device for a period of time, during which all software running on the device will be redeployed and reinstalled from scratch or simply will be replaced along with the processor on which it is running. In the process, any questions the owner or user may have about what is actually going on with the device will almost assuredly be given euphemistic nonanswers because even the trained professionals do not understand the underlying causes, other than the fact that over time something in the software went awry.

Whatever that “something” is, it probably has underlying architectural causes. Every time a new software system is deployed on its target hosts, its initial deployment architecture is established. When that initial system is changed—via a security patch, by deploying a new component, or by adding a new host—its architecture also changes. If the architectural implications of those changes are not carefully analyzed and clearly understood, the system’s architecture is bound to degrade. Eventually, the architecture degrades to the point where the system is unable to function properly, requiring a complete overhaul, as in the above scenarios.

Even though the two are related, the remainder of the chapter addresses separately the role of software architecture in the deployment and in the mobility of software systems.

10.2 SOFTWARE ARCHITECTURE AND DEPLOYMENT

Deployment is the set of activities that result in placing a given software system’s components and connectors on a set of physical hosts. This set of activities can take place both during the system’s construction, that is, prior to run time, as well as during its execution. In the latter case, deployment entails the transfer and activation of components/connectors that are added to the system for the first time—in other words, either new elements or new versions of existing elements. If the elements deployed on a given host had previously been running on another host in the system, thus possibly having run time state, that is considered to be a case of code mobility and is discussed in the next section.

In the rest of this section, we first introduce the basic concepts underlying software deployment. We then elaborate the set of key deployment activities with a specific focus on the role software architecture plays in them. Finally, we discuss the tool support required of architecture-driven software deployment.

¹This phrase is sometimes used by Windows PC owners and users to indicate that the hard drive must be formatted, and the operating system, device drivers, and application programs reinstalled from scratch. Unix and Macintosh users are typically unfamiliar with such behavior.

10.2.1 Basic Concepts

The overview of the key concepts that underlie software deployment, provided in this section, draws from a deployment technology characterization framework by Antonio Carzaniga, Alfonso Fuggetta, Richard Hall, Dennis Heimbigner, André van der Hoek, and Alexander Wolf (Carzaniga et al. 1998).

A software system is deployed on one or more hardware devices, referred to as *hosts* or *sites*. Each site provides a set of *resources* needed for hosting and executing the system or some of its subsystems. The resources include the different elements of the following.

- The hardware architecture (such as memory and CPU).
- The network architecture (such as available protocols and IP port numbers).
- The peripheral devices (such as hard disk and keyboard).
- The system software (such as operating system, device drivers, and middleware).
- Other application-level software (such as GUI builders and databases).
- The data resources (such as data files and Globally Unique Component Identifiers or GUIDs).

Resources can be either *exclusive* (such as IP port number, GUID) or *sharable* (such as CPU, data file).

A software system is composed from a specific set of components and connectors, with carefully prescribed interconnections and allowed interactions. Multiple versions of the components and connectors may exist, meaning that the system itself may have multiple versions as well. A *version* is defined to be a time-ordered revision, a platform-specific variant, or a functional variant.

Initial system deployment involves the transfer of system components or connectors from one or more *source* or *producer* hosts to one or more *destination* or *consumer* hosts. Subsequent deployment activity will typically involve introducing new functionality (that is, new components) to the system, or replacing existing components or connectors with different versions.

10.2.2 Deployment Activities

Architecture-driven software deployment comprises a process that must be carefully planned, modeled, analyzed, and finally effected or executed. We discuss these four activities in more detail below, and specifically focus on the relationship between each activity and software architecture.

Planning

It is critical that the deployment of a software system be carefully planned. Many important system properties, particularly in a distributed setting, will be affected by the system's deployment. For example, the system's latency in delivering a given service can be improved if the system is deployed such that the most frequent and voluminous interactions required for that service occur either locally or over reliable and capacious network links.

For any large, distributed system, many deployments—that is, mappings of software components and connectors onto hardware hosts—will be possible in principle. Some

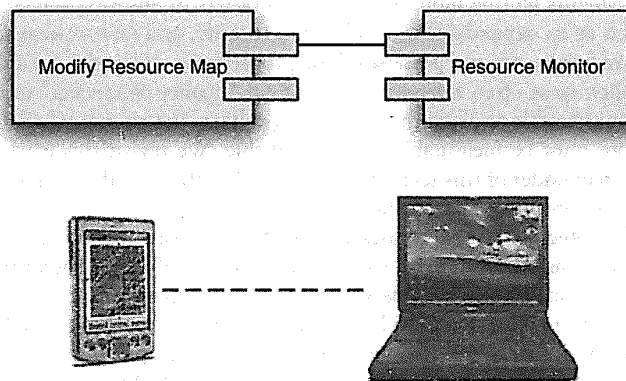
of those deployments will be more effective than others in ensuring the desired system properties, such as its dependability, availability, security, and fault-tolerance. A system that meets its requirements and possesses these properties is said to deliver a desired level of service quality, most often referred to as QoS for “quality of service,” to its users. Of course, in order to be able to claim that the system delivers the required QoS, the different QoS dimensions must be measurable and quantifiable. We revisit this issue in the next section. In the remainder of this section, we assume that the QoS dimensions in question are, in fact, measurable and quantifiable.

It should be noted that there are cases in which the deployment decisions can be and are made with relative ease—regardless of whether they are actually a good fit for the given system or not. One example is a typical desktop environment, in which a human user decides the software that he wants installed on his personal computer. Another example would be a system deployed by a space agency, comprising an interplanetary probe and a ground station. In such systems, it is typically known a priori on which side (flight or ground) the system’s different components will reside. Yet another example, to an extent, is the wireless sensor network system depicted in Figure 10-1: The GUI components implemented in Java will reside on the PDAs, while the computationally intensive components implemented in C++ will be deployed to either the hubs or gateways (Malek et al. 2007). While the potential presence of multiple hubs and gateways, and multiple types of PDAs, still will require that the system’s architects and engineers consider the effects of their deployment decisions within each class of components (in this example, GUI components and computationally intensive components, respectively), the number of possible deployments is significantly reduced.

The deployment problem is substantially more challenging in a system such as the one depicted in Figure 10-2, in which the number of hardware hosts is significantly larger and all devices provide roughly similar execution environments (in this case, Java). The problem of determining an effective deployment becomes intractable for a human engineer if, in addition to this, multiple QoS dimensions such as latency, security, availability, and power usage must be considered simultaneously, while taking into account any additional constraints. For example, component X may not be deployed on hosts Y and Z because of the component’s size, the hosts’ inability to provide the resources necessary for its execution, security concerns, or something else. This is a particularly challenging problem because of the following issues.

- A very large number of system parameters influence the QoS dimensions of a software system. It may be possible to identify a subset of system parameters, such as network bandwidth, network reliability, and frequencies of component interactions, that influence the majority of QoS dimensions; however, it may not be possible to identify all of them.
- Many services provided by a system and their corresponding QoS influence the system users’ satisfaction.
- Different service qualities may be conflicting; that is, improving one may degrade another. A simple example is security and efficiency: If the system’s designers elect to use powerful encryption facilities for all network traffic, such a decision will very likely have a direct, negative impact on the system’s performance.
- The space of possible deployment architectures for a given software system is exponentially large.

Figure 10-3.
A small subset of
the ERS system's
architecture,
comprising two
software
components and
two hardware
hosts. The
components are
depicted in UML
for convenience.



In general, for a system comprising c software components and connectors that need to be deployed onto h hardware hosts, there are h^c possible deployments. Clearly, some of those deployments may not be valid due to location constraints such as those mentioned above, memory restrictions on different devices, network bandwidth considerations, or availability of hardware and system software. This will reduce the space of possible deployments. On the other hand, the human user still must determine which deployment he prefers, why, and, in the process, may have to consider multiple invalid deployments.

As a simple example, consider the following scenario. A very small subset of the ERS system's architecture sketched in Figure 10-2 is depicted in Figure 10-3. This particular system contains only two software components and two hardware hosts; the connector enabling the interaction between the two components has been elided for simplicity. The two components interact to provide a `schedule resource` service, and the only property (that is, QoS dimension) of interest is latency. This small system has four possible deployments:

1. Both components are deployed on the PDA.
2. Both components are deployed on the laptop.
3. `ModifyResourceMap` is deployed on the PDA, while `ResourceMonitor` is deployed on the laptop.
4. `ResourceMonitor` is deployed on the PDA, while `ModifyResourceMap` is deployed on the laptop.

For such a small system, it is, in fact, possible to consider all of the possible deployments and to measure their actual latency. Let us assume that the measured latencies of the four deployments are as shown in Figure 10-4(a). These values are hypothetical, used here for illustration only. From this data, it is easy to determine that the first deployment exhibited the shortest latency and is thus the optimal deployment.

Let us now extend the example slightly and introduce another QoS dimension—durability. We define durability as the inverse of the system's rate of energy consumption. In many embedded and mobile settings, systems with higher energy consumption rates