

Another interesting characteristic of the Weaves model is that it provides information about structural connections, but does not capture aspects of how those connections are used (for example, the sequences or protocols of objects that are passed through them). These details could be specified in an additional model, in natural language or a more formal notation. To elaborate the model above, we might include the following natural language specification:

The connection from *User Interface* to *Calculation* (via Q1) carries objects that include a burn rate and instruct the calculation component to calculate a new lander state. The connection from *Calculation* to *User Interface* (via Q2) indicates when the calculation is complete and also includes the termination state of the application. The connections from *User Interface* and *Calculation* to *Data Store* (via Q3) carry objects that either update or query the state of the lander. The connections back to *User Interface* and *Calculation* from *Data Store* (via Q4) carry objects that contain the lander state, and are sent out whenever the state of the lander is updated.

Takeaways. Weaves shows how binding a notation to a particular style can greatly simplify that notation. Syntactically, Weaves diagrams are extraordinarily simple (even simpler than Darwin diagrams). Unlike Darwin or other general-purpose ADLs, however, the constructs in Weaves diagrams have more specific meanings. The connection from a read port to a write port does not mean just any kind of data or control flow. Rather, it implies a very specific notion of control and data flow involving object streams passed by components running in independent threads of control. When a style is being used, tailored style-specific notations can significantly reduce the cognitive overhead of specifying and interpreting architecture models.

The Architecture Analysis and Design Language (AADL)

The Architecture Analysis and Design Language (AADL, formerly the Avionics Architecture Description Language) (Feiler, Lewis, and Vestal 2003) is an architecture description language for specifying system architectures. While its historical name indicates that its initial purpose was for modeling avionics systems, the notation itself is not specifically bound to that domain—instead, it contains useful constructs and capabilities for modeling a wide variety of embedded and real-time systems such as automotive and medical systems. It is an outgrowth of the earlier MetaH architecture description language (Binns et al. 1996) developed by Honeywell.

Like many of the other ADLs we have surveyed, AADL can describe the structure of a system as an assembly of components, though AADL has special provisions for describing both hardware and software elements, and the allocation of software components to hardware. It can describe interfaces to those components for both the flow of control and data. It can also capture non-functional aspects of components (such as timing, safety, and reliability attributes).

Syntactically, AADL is primarily a textual language, accompanied by a graphical visualization and a UML profile for capturing AADL information in different ways. The syntax of the textual language is defined using Backus-Naur Form (BNF) production rules.

The basic structural element in AADL is the component. AADL components are defined in two parts: a component *type* and a component *implementation*. A component

type defines the interfaces to a component—how it will interact with the outside world. A component implementation is an instance of a particular component type. There may be many instances of the same component type. The component implementation defines the component's interior—its internal structure and construction, and so on. One additional element that affects components is a component's *category*. AADL defines a number of categories (or kinds) of components; these can be hardware (for example, memory, device, processor, bus), software (for example, data, subprogram, thread, thread group, process), or composite (for example, system). The category of a component prescribes what kinds of *properties* can be specified about a component or component type. For example, a thread may have a period and a deadline, whereas memory may have a read time, a write time, and a word size.

AADL is supported by a growing base of tools, including a set of open-source plug-ins for the Eclipse software development environment that provide editing support and import/export capabilities through the Extensible Markup Language (XML) (Bray, Paoli, and Sperberg-McQueen 1998). An additional set of plug-ins is available for analyzing various aspects of AADL specifications—for example, whether all the elements are connected appropriately, whether resource usage by the various components exceeds available resources, and whether end-to-end flow latencies exceed available time parameters.

Evaluation Rubric for AADL	Scope and Purpose	Multilevel models of interconnected hardware and software elements.
	Basic Elements	Myriad hardware and software elements: networks, buses, ports, processes, threads, and many others.
	Style	No explicit support.
	Static and Dynamic Aspects	Captures primarily static aspects of a system, although properties can capture some dynamic aspects.
	Dynamic Modeling	No explicit support.
	Non-Functional Aspects	User-defined properties can capture non-functional aspects, but these cannot be automatically analyzed.
	Ambiguity	Elements such as processes and buses have real-world or physical analogs with well-understood semantics. Properties add detail about behavior to reduce ambiguity.
	Accuracy	Structural and other properties can be automatically analyzed with sufficiently annotated models.
	Precision	Properties on elements specify characteristics of each element with detail sufficient for analysis.
	Viewpoints	A number of interconnected hardware and software viewpoints: system, processor, process, thread, network, and so on.
	View Consistency	The Open Source AADL Tool Environment (OSATE) includes several plug-ins for various consistency checks.

Lunar Lander in AADL. AADL captures the hardware and software elements of a system in great detail, and relates them all to one another. As such, AADL specifications that capture all these elements can become large. Because of this, here we model only a part of the Lunar Lander system: the Calculation component and its connection to the Data Store component. In this version of Lunar Lander, the two components are connected by a physical Ethernet bus. Also, this is a real-time version of Lunar Lander—instead of

operating only when the user inputs a burn rate, the calculation component periodically queries and updates the Data Store component at regular intervals. Here, perhaps the burn rate is not input from the user at the keyboard, but read at periodic intervals from a sensor connected to a burn-rate knob. This is more consistent with a sense-compute-control architectural style.

This part of the Lunar Lander system expressed in AADL might look like Figure 6-23. The first thing to note about this model is the level of detail at which the architecture is described. A component (`calculation_type.calculation`) runs on a physical processor (`the_calculation_processor`), which runs a process (`calculation_process_type.one_thread`), which in turn contains a single thread of control (`calculation_thread`), all of which can make two kinds of request-

```

data_lander_state_data
end_lander_state_data;

bus_lan_bus_type
end_lan_bus_type;

bus_implementation_lan_bus_type.ethernet
properties
  Transmission_Time => 1 ms .. 5 ms;
  Allowed_Message_Size => 1 b .. 1 kb;
end_lan_bus_type.ethernet;

system_calculation_type
features
  network      : requires bus access lan_bus.calculation_to_datastore;
  request_get   : out event port;
  response_get  : in event data port_lander_state_data;
  request_store : out event port_lander_state_data;
  response_store : in event port;
end_calculation_type;

system_implementation_calculation_type.calculation
subcomponents
  the_calculation_processor : processor_calculation_processor_type;
  the_calculation_process   : process_calculation_process_type.one_thread;
connections
  bus_access_network -> the_calculation_processor.network;
  event_data_port_response_get -> the_calculation_process.response_get;
  event_port_the_calculation_process.request_get -> request_get;
  event_data_port_response_store -> the_calculation_process.response_store;
properties
  Actual_Processor_Binding => reference
    the_calculation_processor.applies_to_the_calculation_process;
end_calculation_type.calculation;

processor_calculation_processor_type
features
  network : requires bus access lan_bus.calculation_to_datastore;
end_calculation_processor_type;

```

(Continued)

Figure 6-23.
Partial Lunar
Lander model in
AADL.

Figure 6-23.
(Continued)

```

process calculation_process_type
features
  request_get : out event port;
  response_get : in event data port lander_state_data;
  request_store : out event data port lander_state_data;
  response_store : in event port;
end calculation_process_type;

thread calculation_thread_type
features
  request_get : out event port;
  response_get : in event data port lander_state_data;
  request_store : out event data port lander_state_data;
  response_store : in event port;
properties
  Dispatch_Protocol => periodic;
end calculation_thread_type;

process implementation calculation_process_type.one_thread
subcomponents
  calculation_thread : thread client_thread_type;
connections
  event data port response_get -> calculation_thread.response_get;
  event port calculation_thread.request_get -> request_get;
  event port response_store -> calculation_thread.response_store;
  event data port request_store -> request_store;
properties
  Dispatch_Protocol => Periodic;
  Period => 20 ms;
end calculation_process_type.one_thread;

```

response calls through ports (`request_get/response_get`, `request_store/response_store`) over an Ethernet bus (`lan_bus_type.ethernet`). Each of these different modeling levels is connected through composition, mapping of ports, and so on. This level of detail emphasizes the importance of tools, such as graphical editors, for modeling this information in a more understandable fashion.

The second thing to note about this specification is the use of properties to define specific characteristics of some of the elements (for example, the calculation process runs periodically, every 20ms). In a complete system specification, these properties would be even more detailed and elaborate, and could be used by tools to answer questions about availability and timing that are critical in a real-time context.

Takeaways. AADL is a high-cost, high-value notation. The syntax and capabilities of AADL are the product of a significant amount of thought and development effort, and the kinds of analyses that can be done on AADL specifications are nontrivial. However, while users leverage the effort of the AADL developers, they also incur their own costs: AADL specifications are complex and extensive, even for small systems. This, in itself, entails significant risks and costs for users. For the specific domain in which AADL is positioned (embedded real-time systems), these costs may be worth it: These systems are often safety-critical and expensive or impossible to redeploy.

6.5.4 Extensible ADLs

There is a natural tension between the expressiveness of general-purpose modeling languages such as UML and the semantic power of more specialized ADLs such as Wright, Weaves, and Rapide. On one hand, architects need the ability to model a wide variety of architectural concerns. On the other hand, they need semantically precise languages to reduce ambiguity and increase analyzability. Furthermore, as we have discussed, architectural concerns and product needs can vary widely from project to project. One solution is to use multiple notations, each addressing a different set of architectural concerns. However, this creates difficulty in keeping the different models consistent and establishing traceability between models. Additionally, even using multiple notations, there may be certain concerns that cannot be adequately captured in any existing notation. Extensible architecture description languages can be used to combine the flexibility of generic languages with the analyzability and precision of semantically rich languages.

Extensible ADLs generally provide a basic set of constructs for describing certain common architectural concerns (for example, components and connectors). Additionally, they also include support (through the notation and associated tools) for extending the notation's syntax to support new, user-defined constructs. In some cases, the extensions may only modify the existing base constructs. Other notations allow users to create entirely new constructs and concepts.

The basic approach to employing an extensible ADL is as follows:

1. Determine which concerns can be modeled using the existing (baseline) capabilities of the ADL.
2. For those concerns that cannot be modeled using the baseline capabilities, choose how to extend the ADL to support their modeling (or reuse an extension developed by another user).
3. Extend the ADL and its supporting tools as necessary to support the modeling of the unique features.

Effectively, extensible ADLs can be seen as “domain-specific ADL factories.” Through extension mechanisms, users can tailor these ADLs to meet the needs of their particular domain or project. Examples of extensible ADLs include Acme, ADML, and xADL.

Acme

Acme (Garlan, Monroe, and Wile 1997) is perhaps the earliest example of an ADL that emphasized extensibility. Acme has a base set of seven constructs: *components* and *connectors*, *ports* and *roles* (roughly equivalent to this book's notion of interfaces on components and connectors, respectively), *attachments* (equivalent to this book's notions of links), *systems* (configurations of components, connectors, ports, roles, and attachments), and *representations* (inner architectures for components and connectors). These constructs provide a vocabulary for expressing structural design decisions about a system: basically, the system's component and connector makeup in a box-and-arrow form.

Extensibility in Acme is derived through the use of *properties*. Properties are decorations that can be applied to any of the basic seven kinds of elements. Properties are optionally typed name-value pairs. Names are simple strings, and the values should conform to the property's type, if it has one. Acme includes a number of simple property types (integers,

Booleans, and so on), as well as the ability to define custom types. Either way, from Acme's point of view, properties and their values are uninterpreted. It is up to user-written tools to parse these properties (and any internal structure they contain) and do something useful with the contents (for example, provide analysis or visualization capabilities).

Originally, it was anticipated that Acme's primary use would be as an architecture *interchange* language, rather than an ADL or even an extensible ADL. That is, developers of other ADLs such as Wright and Rapide would develop translators that mapped their languages to and from Acme: In this way, models could be interchanged among many different tools for, say, analysis. This strategy was never applied on a wide scale, however, possibly because of substantial semantic differences and conflicts among target ADLs.

Evaluation Rubric for Acme	Scope and Purpose	Modeling the structural aspects of a software architecture, with the addition of properties to define other aspects.
	Basic Elements	Components, connectors, ports and roles (interfaces), attachments (links), representations (internal structure), and properties.
	Style	Stylistic similarities can be modeled through the use of Acme's type system.
	Static and Dynamic Aspects	Static structure is modeled natively, dynamic properties can be captured by decorating the static elements with formatted properties.
	Dynamic Modeling	A software library, AcmeLib, allows Acme models to be manipulated programmatically.
	Non-Functional Aspects	Properties can capture non-functional aspects, but these cannot be automatically analyzed.
	Ambiguity	The use of various elements (for example, components and connectors) is well defined, but what they mean externally (for example, what it means to be a component) is not defined. In general, Acme properties are not interpreted and can introduce ambiguity if not accompanied by tools or documentation.
	Accuracy	The use of various elements (for example, components and connectors) is well defined. Typing can be used to check whether elements have particular properties, but external tools are required to check the contents of properties.
	Precision	Properties can annotate the existing element types with whatever detail is required, but it is not generally possible to define new kinds of architectural elements.
	Viewpoints	Natively, structural viewpoints are supported; properties can be used to provide additional viewpoints.
	View Consistency	External tools must be developed to check the consistency of views.

Lunar Lander in Acme. The basic specification of Lunar Lander in Acme is similar to models shown above. It is largely structural and includes components, connectors, ports, roles, and attachments. Lunar Lander, specified in Acme, might look like Figure 6-24.

```

//Global Types
Property Type returnsValueType = bool;

Connector Type CallType = {
  Roles { callerRole; calleeRole; };
  Property returnsValue : returnsValueType;
};

System LunarLander = {
  //Components
  Component DataStore = {
    Ports { getValues; storeValues; };
  };
  Component Calculation = {
    Ports { calculate; getValues; storeValues; };
  };
  Component UserInterface = {
    Ports { getValues; calculate; };
  };
  // Connectors
  Connector UserInterfaceToCalculation : CallType {
    Roles { callerRole; calleeRole; };
    Property returnsValue : returnsValueType = true;
  };
  Connector UserInterfaceToDataStore : CallType {
    Roles { callerRole; calleeRole; };
    Property returnsValue : returnsValueType = true;
  };
  Connector CalculationToDataStoreS : CallType {
    Roles { callerRole; calleeRole; };
    Property returnsValue : returnsValueType = false;
  };
  Connector CalculationToDataStoreG : CallType {
    Roles { callerRole; calleeRole; };
    Property returnsValue : returnsValueType = true;
  };
  Attachments {
    UserInterface.getValues to UserInterfaceToDataStore.callerRole;
    UserInterfaceToDataStore.calleeRole to DataStore.getValues;

    UserInterface.getValues to UserInterfaceToDataStore.callerRole;
    UserInterfaceToDataStore.calleeRole to DataStore.getValues;

    UserInterface.calculate to UserInterfaceToCalculation.callerRole;
    UserInterfaceToCalculation.calleeRole to Calculation.calculate;

    Calculation.storeValues to CalculationToDataStoreS.callerRole;
    CalculationToDataStoreS.calleeRole to DataStore.storeValues;

    Calculation.getValues to CalculationToDataStoreG.callerRole;
    CalculationToDataStoreG.calleeRole to DataStore.getValues;
  };
};

```

Figure 6-24.
Structural Acme
model of the
Lunar Lander
application.

One interesting thing to note about this model is its verbosity, especially compared to some earlier models. This is a result of two key decisions made by the Acme designers. First, the Acme language is domain-neutral: little can be abbreviated out of the model, because the underlying Acme semantics make few assumptions that allow information to be conveyed implicitly. Second, the Acme designers intended Acme models to be edited through graphical tools and developed an environment called AcmeStudio that allows users to design Acme structures using a GUI interface rather than writing the complete textual description of the model in a text editor. This limits the amount of additional effort Acme users must expend to deal with Acme's verbosity.

A second thing to notice about the model is the use of properties. In this basic model, only one kind of property is used: Each procedure call connector is annotated with a property indicating whether or not the operation has a return value. This might be useful to designers looking to make a partially asynchronous version of Lunar Lander: Calls without return values may be able to return control to the caller without waiting for their completion. Acme's type system is used to declare a connector and a property type, such that regularity can be imposed over connectors and properties that are similar (for example, all connectors in the above Lunar Lander model are of the same type: `CallType`). Types in Acme work in a manner similar to stereotypes in a UML profile: They allow the user to set expectations on instance elements that can be checked later.

Compared to the other Lunar Lander models above, this basic Acme model is not very innovative. However, additional properties can add detail to this model in a number of ways, depending on the stakeholder's needs. For example, we might say more about how the `DataStore` component stores its data, as shown in Figure 6-25. This extended description of the component is intended to indicate that the `DataStore` component should store its data in a non-replicated table called "`LanderTable`" in a relational database. However, the fact that the model uses these particular property names and values is not a standard: Tools and stakeholders must be informed about which properties to expect and how to process their values.

Takeaways. Notwithstanding the relative failure of architectural interchange, the notion that extensibility should be a primary concern in ADL development is a good one—it is an explicit recognition that there cannot be a one-size-fits-all ADL. Property-based decorations as an extensibility mechanism provide a degree of flexibility, but the breadth of architectural concerns that stakeholders want to capture requires the ability to define new first-class constructs.

The Architecture Description Markup Language (ADML)

The Architecture Description Markup Language (ADML) (Spencer 2000) is an XML-based architecture description language whose syntax is derived from Acme. It was originally developed by the MicroElectronics and Computer Technology Consortium (MCC) and is maintained as a standard by the Open Group. Semantically, the two are nearly identical, with ADML's primary addition being support for meta-properties. In ADML, meta-properties provide a mechanism by which users can specify the properties (and property types) that should be present on particular elements.

```

Property Type StoreType = enum { file, relationalDatabase, objectDatabase };

Component DataStore = {
  Ports {
    getValues; storeValues;
  };
  Property storeType : StoreType = relationalDatabase;
  Property tableName : String = "LanderTable";
  Property numReplicas: int = 0;
};

```

Figure 6-25.
Extended
definition of the
Lunar Lander
DataStore
component in
Acme.

Using the Extensible Markup Language (XML) as the basis for an architecture modeling notation confers several benefits. XML provides a standard framework for expressing the syntax of notations. It is well suited to describing hierarchical organizations of concepts, and internal references can be used to create “pointers” from one element to another. A significant practical benefit of the use of XML is the panoply of off-the-shelf commercial and open-source tools that are available for parsing, manipulating, and visualizing XML documents. Using these tools can significantly reduce the amount of time and effort needed to construct architecture-centric tools such as analyzers, editors, and so on.

XML can be used to create languages with extensible syntax. However, early XML standards provided limited support for this. ADML’s syntax is defined using an XML document type definition (DTD), one of the earliest XML meta-languages, which uses production rules to define syntax. XML DTDs have been used to create extensible languages, but the mechanisms are somewhat cumbersome. Instead, ADML provides its extensibility through the same mechanism as Acme—name-value pair properties applied to a core set of elements.

Lunar Lander in ADML. Because ADML is so similar to Acme, we will not provide a complete ADML description for the Lunar Lander application. The description of the DataStore component in ADML might look like Figure 6-26. Although the semantic content of this ADML snippet is virtually identical to its Acme counterpart, the use of XML as a standard syntactic framework opens this specification up to a much wider array of tools such as parsers and editors that would not necessarily be otherwise available (or as ubiquitous). One obvious drawback is that by employing XML, the already verbose Acme description becomes even denser. This reflects the overall attitude of XML in general—computing resources and data storage have become inexpensive and plentiful,

```

<Component ID="datastore" name="Data Store">
  <ComponentDescription>
    <ComponentBody>
      <Port ID="getValues" name="getValues"/>
      <Port ID="storeValues" name="storeValues"/>
    </ComponentBody>
  </ComponentDescription>
</Component>

```

Figure 6-26.
Description of
one Lunar
Lander
component in
ADML.

and editing tools have become ubiquitous. Conservation of data bytes is no longer a driving force in language design.

Takeaways. XML provides many extrinsic benefits for architecture modeling, and nearly all architecture description notations still under development at least have the option to import from and export to XML. ADML does not take advantage of XML's own extensibility mechanisms, and instead provides extensibility by simply encoding Acme's name-value pair properties into XML.

xADL: An Extensible XML-based Architecture Description Language

xADL (Dashofy, van der Hoek, and Taylor 2005) is an attempt to provide a platform upon which common modeling features can be reused from domain to domain and new features can be created and added to the language as first-class entities (not just extensions to other entities as with Acme and ADML). Like ADML, xADL is an XML-based language. That is, every xADL model is a well-formed and valid XML document. The main difference is that unlike ADML, xADL fully leverages XML's extensibility mechanisms for its language extensions.

The syntax of the xADL language is defined in a set of XML schemas. XML schemas are similar to DTDs in that they provide a format in which to define the syntax of a language. Whereas DTDs define document syntax through production rules, XML schemas define syntax through a set of data types, similar to the way data structures are defined in an object-oriented programming language. Like classes in an object-oriented programming language, XML schema data types can be extended through inheritance. Derived data types can be declared in separate schemas. Through this mechanism, data types declared in one schema can be extended in a different schema to add new modeling features.

Syntactically, the xADL language is the composition of all the xADL schemas. Each xADL schema adds a set of features to the language. The constructs in each schema may be new top-level constructs or they may be extensions of constructs in other schemas. Breaking up the feature set in this way has several advantages. First, it allows for *incremental adoption*—users can use as few or as many features as makes sense for their domain. Second, it allows for *divergent extension*—users can extend the language in novel, even contradictory ways to tailor the language for their own purposes. Third, it allows for *feature reuse*—because feature sets are defined in XML schema modules, schemas can be shared among projects that need common features without each group having to develop its own (probably incompatible) representations for common concepts.

From time to time, new schemas are added to xADL as they are developed, by both its creators and outside contributors. Current xADL schemas are shown in the box.

Current xADL Schemas	Schema	Features
	Structure and Types	Defines basic structural modeling of prescriptive architectures: components, connectors, interfaces, links, general groups, as well as types for components, connectors, and interfaces.
	Instances	Basic structural modeling of descriptive architectures: components, connectors, interfaces, links, general groups.

Abstract Implementation	Mappings from structural element types (component types, connector types) to implementations.
Java Implementation	Mappings from structural element types to Java implementations.
Options	Allows structural elements to be declared optional—included or excluded from an architecture depending on specified conditions.
Variants	Allows structural element types to be declared variant—taking on different concrete types depending on specified conditions.
Versions	Defines version graphs; allows structural element types to be versioned through association with versions in version graphs.

Because xADL can be extended with unforeseen constructs and structures in nearly arbitrary ways, it induces challenges that do not exist in languages with unchanging syntaxes and semantics (most of the other ADLs we have covered). Specifically, parsers, editors, analyzers, and other tools must be developed to cope with a notation whose syntax may change from project to project.

xADL addresses these challenges with an associated set of tools that make it easier for users to implement their own tools (for example, analyzers, editors) that can handle the language's extensible syntax. These include the following.

The xADL Data Binding Library. A data binding library is a software library that provides an API for parsing, reading, writing, and serializing (writing back out to disk or other storage) documents in a particular language. In xADL's case, the data binding library consists of a set of Java classes that correspond to xADL data types. A program can query and manipulate instances of these classes to explore and change an xADL document. Although xADL documents can also be read, written, and manipulated with any XML-aware tool, the data binding library provides a much simpler interface and is the basis for most of the other xADL tools developed to date.

Apigen. The data binding library would be of limited use if it had to be manually rewritten each time schemas were added to xADL. Apigen (Dashofy 2001) is xADL's data binding library generator: Given a set of XML schemas, it can generate the complete data binding library with support for those schemas. If a user changes or adds a schema, a new data binding library can be generated by rerunning Apigen; this library will contain classes for querying and manipulating the new or changed data types defined in the user's schema.

The native storage format of xADL is in an XML format. Although XML is designed to be readable by both humans and software tools, certain factors can impede the human-readability of XML documents. Specifically, xADL's extensive use of XML namespaces and multiple schemas adds a significant amount of "housekeeping" data to xADL documents. This data is not without use—it is used by tools such as the data binding library and XML validators—but it makes it more difficult to comprehend a xADL document for humans attempting to read it for content.

For example, a simple component in xADL's XML format might look like this:

```
<types:component xsi:type="types:Component"
    types:id="myComp">
  <types:description xsi:type="instance:Description">
```

xADL and
xADLite

```

    MyComponent
  </types:description>
  <types:interface xsi:type="types: Interface"
                    types:id="iface1">
  <types:description xsi:type="instance:Description">
    Interface1
  </types:description>
  <types:direction xsi:type="instance:Direction">
    inout
  </types:direction>
  </types:interface>
</types:component>

```

Even with namespace and XML typing information removed, the result is still verbose:

```

<component id="myComp">
  <description>
    MyComponent
  </description>
  <interface id="iface1">
    <description>
      Interface1
    </description>
    <direction>
      inout
    </direction>
  </interface>
</component>

```

From time to time in this book, we will use an alternate, more readable representation of xADL documents called xADLite. xADLite is a more syntactically terse format than xADL's native XML representation, but the transformation from xADLite to XML and back again is lossless—no information is lost when translating a document from xADLite to xADL or back again. The same component in xADLite looks like this:

```

component{
  id ="myComp";
  description ="MyComponent";
  interface{
    id ="iface1";
    description ="interface1";
    direction ="inout";
  }
}

```

Scope and Purpose	Modeling architecture structure, product lines, and implementations, with support for extensibility	Evaluation Rubric for xADL
Basic Elements	Components, connectors, interfaces, links, options, variants, versions, plus any basic elements defined in extensions.	
Style	Stylistic aspects of an architecture can be modeled through the use of types and type libraries.	
Static and Dynamic Aspects	Static structure is modeled natively, dynamic properties can be captured through extensions. The MTAT project (Hendrickson, Dashofy, and Taylor 2005) extends xADL to describe the external behavior of components and connectors that communicate through messages.	
Dynamic Modeling	The xADL data binding library allows xADL specifications to be manipulated programmatically.	
Non-Functional Aspects	Extensions can be written to capture non-functional aspects.	
Ambiguity	The xADL language is purposefully permissive in how its elements may be used, although documentation indicates their intended use. Tools are available to automatically check constraints on xADL documents and allow users to define their own constraints.	
Accuracy	Tools are provided to check the correctness of xADL documents; additional constraints can be written into these tools to handle extensions.	
Precision	Extensions can be used to annotate existing element types with whatever detail is required or create entirely new first-class constructs.	
Viewpoints	Natively, structural viewpoints (both run time and design time) are supported as well as product-line views; extensions can be used to provide additional viewpoints.	
View Consistency	External tools can check the consistency of views; frameworks for developing such tools are provided.	

Lunar Lander in xADL. For basic structural specifications, xADL has a great deal in common with Acme and ADML. The basic structure of the Lunar Lander application, expressed in xADL's xADLite textual visualization might look like Figure 6-27. Like Darwin and other ADLs, xADL also has an associated graphical visualization. This visualization is provided by an editor called Archipelago, which is part of the xADL tool set.

The model of Lunar Lander rendered by Archipelago is shown in Figure 6-28. As with Acme, the key notational contribution of xADL lies in its extensibility. xADL's extensibility goes beyond adding properties to the core constructs. In fact, xADL has no fundamental separation between core concepts and extensions. xADL allows the addition of completely new syntactic elements in addition to structured extensions to existing elements.

Reusing the example of the improved data source component specification above for Acme, let's look at how more detail about the data store component might be added to xADL (and the xADL specification of Lunar Lander). The XML schema specification for a "plain" xADL component (as used above) is shown in Figure 6-29. Simply put, this specification says that a component has the following.

- One identifier (a string attribute).
- One description (a string element).

Figure 6-27.
Lunar Lander
architecture
described in
xADLite.

```
xArch{
  archStructure{
    id = "lunarlander";
    description = "Lunar Lander";
    component{
      id = "datastore";
      description = "Data Store";
      interface{
        id = "datastore.getValues";
        description = "Data Store Get Values Interface";
        direction = "in";
      }
      interface{
        id = "datastore.storeValues";
        description = "Data Store Store Values Interface";
        direction = "in";
      }
    }
  }
  component{
    id = "calculation";
    description = "Calculation";
    interface{
      id = "calculation.getValues";
      description = "Calculation Get Values Interface";
      direction = "out";
    }
    interface{
      id = "calculation.storeValues";
      description = "Calculation Store Values Interface";
      direction = "out";
    }
    interface{
      id = "calculation.calculate";
      description = "Calculation Calculate Interface";
      direction = "in";
    }
  }
  component{
    id = "userinterface";
    description = "UserInterface";
    interface{
      id = "userinterface.getValues";
      description = "User Interface Get Values Interface";
      direction = "out";
    }
    interface{
      id = "userinterface.calculate";
      description = "User Interface Calculate Interface";
      direction = "out";
    }
  }
}
```

Figure 6-27.
(Continued)

allows us to define many different subsets of a database that will be used to generate new information upon which the comparison will occur. By adding another schema, we can create an extended form of a comparison that

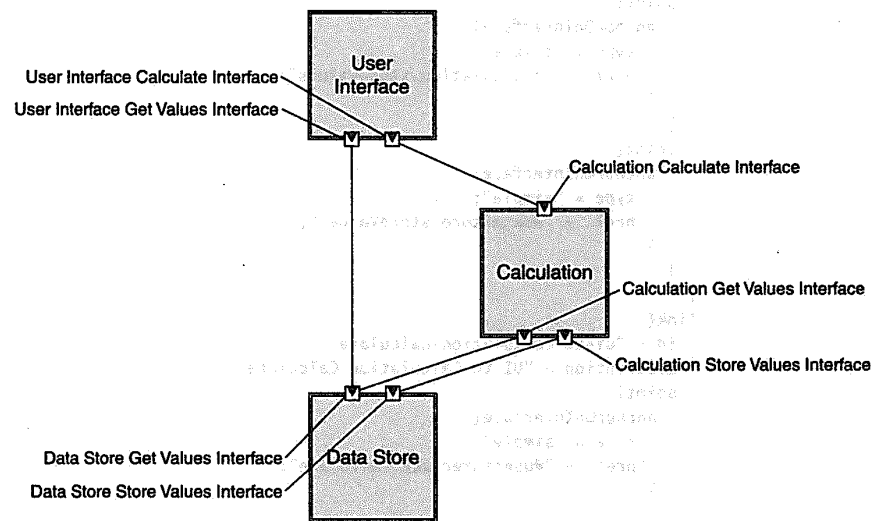
Figure 6-27.
(Continued)

```

link{
  id = "ui-to-datastore-getvalues";
  description = "UI toto Data Store Get Values"
  point{
    anchorOnInterface{
      type = "simple";
      href = "#userinterface.getValues";
    }
  }
  point{
    anchorOnInterface{
      type = "simple";
      href = "#datastore.getValues";
    }
  }
}
}
}

```

Figure 6-28.
Lunar Lander in
xADL, as
visualized by the
Archipelago
graphical editor.



- Zero or more interfaces.
- An optional link to its type.

By adding another schema, we can create an extended form of a component that can be used to capture more information about where the component will store data. An example extension is shown in Figure 6-30. This extension to the xADL language adds several new capabilities. First, it defines a new abstract data type called **Database**. This is like declaring an abstract class in an object-oriented programming language: It indicates we are going to define many different subtypes of **Database** that will be

```

<complexType name="Component">
  <sequence>
    <element name="description" type="Description"/>
    <element name="interface" type="Interface"
      minOccurs="0" maxOccurs="unbounded"/>
    <element name="type" type="XMLLink"
      minOccurs="0" maxOccurs="1"/>
  </sequence>
  <attribute name="id" type="Identifier"/>
</complexType>

```

Figure 6-29.
XML schema
description of an
xADL
component.

```

<complexType name="Database" abstract="true"/>

```

Figure 6-30.
XML schema
extension adding
database
component
capabilities to
xADL.

```

<complexType name="RelationalDatabase">
  <complexContent>
    <extension base="Database">
      <sequence>
        <element name="tableName" type="string"/>
        <element name="numReplicas" type="int"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>

<complexType name="FileDatabase">
  <complexContent>
    <extension base="Database">
      <sequence>
        <element name="fileName" type="string"/>
        <element name="hostName" type="string"
          minOccurs="0" maxOccurs="1"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>

<complexType name="DatabaseComponent">
  <complexContent>
    <extension base="Component">
      <xsd:sequence>
        <xsd:element name="database" type="Database"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

```

Figure 6-31.
Extended Lunar
Lander Data
Store
component in
xADL.

```

component type="DatabaseComponent" {
  id = "datastore";
  description = "Data Store";
  interface{
    id = "datastore.getValues";
    description = "Data Store Get Values Interface";
    direction = "in";
  }
  interface{
    id = "datastore.storeValues";
    description = "Data Store Store Values Interface";
    direction = "in";
  }
  database type="RelationalDatabase" {
    tableName = "landerData";
    numReplicas = "1";
  }
}

```

substitutable whenever a Database is needed. Then, it defines two concrete subtypes of Database: `RelationalDatabase` and `FileDatabase`. These have different characteristics: a `RelationalDatabase` has a table name and a number of replicas, while a `FileDatabase` has a file name and an optional host name on which the file resides.

Finally, it defines an extension to the plain xADL Component datatype. A `DatabaseComponent` has everything that a component does, plus a `Database` element. Because both `RelationalDatabase` and `FileDatabase` are `Databases`, either one can be used here (as well as other user-defined database types in other schemas).

We can now extend the original xADL description of the Data Store component in the Lunar Lander application, as shown in Figure 6-31. Using these mechanisms, stakeholders can define their modeling needs very precisely. When the new “DatabaseComponent” schema is ready, Apigen can generate a new data binding library that supports the new constructs. Furthermore, sharing of schemas is possible (and encouraged)—the above “DatabaseComponent” schema could be reused from project to project, as well as any tools that provide advanced processing (editing, analysis, and so on) on elements from this schema.

Takeaways. While xADL’s basic modeling capabilities rival those of other ADLs we have covered, xADL’s primary contribution is in its extensibility. xADL heavily leverages XML’s (and XML Schema’s) extensibility mechanisms to create a domain-specific ADL factory, where domain-specific ADLs (and supporting tools) can be created at a much lower cost than developing them from scratch. xADL is supported by a variety of visualization, analysis, and utility tools in the ArchStudio environment. These various tools will be discussed throughout the remainder of the book.

6.6 WHEN SYSTEMS BECOME TOO COMPLEX TO MODEL

In this chapter, we have shown how various notations can express a simple application like Lunar Lander. The Lunar Lander architecture has certain properties that simplify modeling:

It has a finite number of well-known components that run on a single host. Even large distributed systems can be modeled in similar ways, although the effort required to model these systems increases with their size. However, certain applications (especially Internet-scale applications) cannot be modeled using these same techniques. For example, it would be virtually impossible to model the Web, or Gnutella (Kan, 2001) using some of the modeling techniques above. These are gigantic and diverse applications. The configuration of components and connectors on the Internet is constantly changing. In fact, they are so big and so volatile that it is *impossible* to know the configuration of components and connectors at any given time. Thus, it is not possible to generate a model of these systems in the traditional components-connectors-and-configurations sense. There are far too many components, and the structure of the application is changing too fast.

Modeling applications with this kind of complexity is possible, using some of the techniques we have already discussed. All of them involve abstracting away aspects of the complexity to get to a point where modeling is feasible but still useful. This is the essence of modeling. Strategies to consider in this case include the following.

- *Model limited aspects of the architecture:* Many times, the largest or most complex part of an architecture is limited to one particular perspective. For example, trying to model the structure of the Web would be impossible. However, use cases or interaction patterns are more limited, and might be modeled more easily.
- *Model an instance:* The first thing to consider is whether a complete model of such a system is actually needed. In general, nobody needs to model the entire Web or Gnutella P2P network at any given time. Instead, consider modeling only the portion of the system that is relevant, which will almost certainly be a (relatively) small subset of the total system.
- *Exploit regularity:* Often, extremely large systems have low heterogeneity. Large portions of the system will, for modeling purposes, look almost exactly like other portions of the system. These portions can be modeled once and repeated automatically, or decorated with a property indicating how many of them there are. Recall Darwin's ability to specify multiple homogenous clients using a "for loop" construct.
- *Model the style:* As discussed earlier in the book, the Web is based on the REST architectural style. Instead of modeling the Web as an application, consider modeling the REST style instead. We have discussed several notations that allow modeling of different kinds of architectural style constraints; these can be used for this purpose.
- *Model the protocol:* The "contract" that large distributed systems adhere to is often captured in the interaction protocol between components. For example, the Web is, in large measure, characterized by adherence to the HTTP protocol (Fielding et al. 1999). A number of the notations discussed in this chapter can be used to model protocol details, such as natural language (albeit informally) and Wright.

6.7 END MATTER

In this chapter, we have introduced architecture modeling and identified many of the issues that occur in the modeling process. We have also introduced and provided examples of many architecture description notations.

It is impossible to provide deep coverage of the variety of architecture description notations in this text—indeed, books, theses, and academic papers abound discussing each notation in great detail. Instead, we have attempted to highlight the distinguishing aspects of each notation. We encourage interested readers to leverage the Web and the resources indicated at the end of the chapter to investigate these ADLs more thoroughly from primary sources.

The goal of this chapter is to give readers a sense of the breadth of the modeling activity, as well as the notations, tools, and techniques that are available to support it. What should be obvious is that no single notation—even an extensible notation—is sufficient to capture all the aspects of architecture that will be important for a complex project. Stakeholders must make informed choices of techniques, based on the needs of the project at hand. The following table recaps the modeling approaches we have discussed in this chapter and attempts to group them into categories that share characteristics. Note that some approaches fall in more than one category. One example is Darwin. It is grouped with the syntactically rigorous ADLs such as UML and OMT because of its usefulness as a straightforward way of rigorously capturing and communicating architectural structure. It is also in the formal category because of its formal foundations in the *pi* calculus, although these formal foundations are rarely exploited.

Ideally, stakeholders and architects would simply be able to select a mature notation that would cover their modeling and analysis needs. However, the reality is that there are not enough mature notations currently available to make this possible. Even if one were prepared to use several notations simultaneously (and deal with the costs of managing consistency among them), there will likely still be gaps between what can be modeled and what stakeholders need. This means that architects will not only have to invest in modeling, but invest in developing the technologies they use to model. Modeling would be substantially more effective if a portion of the total design budget of a system were devoted to enhancing the fundamental technologies used to do the design. Nevertheless, selecting and using at least one architecture modeling notation—preferably an extensible

Category	Category Characteristics	Examples
Informal/Free-form	High expressiveness, ambiguity, lack of rigor, no formal semantics	Natural language, informal graphical diagrams.
Syntactically rigorous	Rigorous syntax but few semantics, primarily useful for communication.	UML, OMT, Darwin.
Formal	Rigorous syntax and underlying formal semantics, automated analyses possible, steeper learning curves.	Darwin, Rapide, Wright.
Domain- or style-specific	Optimization for a particular domain or style, has strong mapping to domain or style-specific concepts.	Weaves, Koala, AADL.
Extensible	Support for a core set of constructs and the ability for users to define their own.	ACME, ADML, xADL, UML.

notation—is worlds better than not doing modeling at all, and significantly better than using a semantically impoverished notation such as PowerPoint or UML.

Selecting notations for use on a project is a subtle and complex activity. This chapter focused on what aspects of architecture are modeled by different notations, and how those notations model those aspects. These are only two factors in selecting a notation. Other important questions must also be answered: What kinds of visualizations are available with the notation? What kinds of analyses can be performed on models in the notation? How can models in this notation be tied to other life cycle phases? All these aspects of designing will be addressed in subsequent chapters. The next chapter focuses specifically on visualizations, where we separate the abstract concept of a notation (a way of organizing the information in design decisions) from the different ways that the information can be interacted with and depicted.

Deciding what to model and what approaches to use is very much a choice driven by costs and benefits. Modeling takes time and effort, as does maintaining models that have already been created. The more complex, precise, and unambiguous the models, the more cognitive effort, time, and money will be invested in creating them.

Different kinds of models produce different kinds of benefits. These benefits generally come in the form of cost mitigation: The models themselves do not produce revenue (except perhaps in the case when they are part of documentation or training that can be sold to customers). Typical benefits of modeling include:

- **As documentation:** Models can serve as points of reference and vehicles for communicating ideas about the system among diverse stakeholders. This increases understanding and reduces confusion, which can lead to fewer problems in later development stages such as integration and acceptance testing; it can also help to determine whether the system is consistent with its requirements and whether the requirements are themselves appropriate.
- **Earlier fault detection:** Good models can help stakeholders detect problems earlier (through inspection and analysis) and fix them before they become costly.
- **Lower change costs:** By giving developers a specific sense of the current architecture (and architectural style), changes can be made following established guidelines with more insight about the effect of a given change on the system.
- **Generation of other artifacts:** Some models can be used to generate specific types of documentation or even (partial) implementations automatically, thus saving the costs of generating these artifacts manually. Generation has the added benefit that it costs less to maintain the connection between models and other development activities, since models can simply be used to regenerate other artifacts when they change.

As part of project inception and business planning, it is a good idea to develop a modeling strategy, which is simply an agreement between stakeholders on what kinds of models will be constructed, what notations or tools will be used, what consistency criteria will be applied, how those models will be used in the overall development activity, and so on. It is also advisable to develop and document cost/benefit arguments for each model or type of model, capturing the rationale for each model's creation and use. Additionally, just as project scheduling can be used to estimate the time and cost of implementing various aspects of the system, scheduling can also be applied to the development and maintenance of models.

The Business Case

6.8 REVIEW QUESTIONS

1. What is an architectural model? What is its relationship to an architecture? To design decisions?
2. What is the difference between accuracy and precision in general? In the context of architectural modeling?
3. What is the difference between a view and a viewpoint?
4. Enumerate some common architectural viewpoints. What kind of data is captured in each viewpoint?
5. What does it mean for two views to be consistent? What kinds of inconsistencies can arise in multiview models?
6. What kinds of dimensions can you use to evaluate a modeling notation?
7. Enumerate some modeling notations and describe what kinds of design decisions they can capture.
8. What are the distinctive modeling features of Darwin? Of Wright? Of Rapide? Of Weaves?
9. What are the advantages of using an extensible ADL? What are some disadvantages?
10. What are some modeling strategies that can be used when a system is too big or complex to create a complete architectural model?

6.9 EXERCISES

1. Research in more depth the syntax and semantics of one of the ADLs in this chapter and model a system of your choosing in that ADL.
2. Download a tool set for one of the modeling notations described in this chapter and model a simple application like Lunar Lander in that tool set. Reflect on your experiences and the strengths and weaknesses of the tool set.
3. Map the syntax of an earlier ADL (for example, Darwin, Wright, Rapide) to an extensible ADL. What services are provided by the extensible ADL? What extensions are needed to fill in the gaps? Define these extensions using the mechanisms provided by the extensible ADL.
4. Choose a software-intensive system and research how this system's developers employed modeling. What kind of models were created and how are they used? How are they maintained?
5. Research how other disciplines (for example, computer engineering or building architecture) approach the modeling task. What do they model? What notations do they use? Do similar issues (such as view consistency) occur in those disciplines? How are they handled there?
6. Do you agree that UML is an architecture description language? Argue for or against.
7. Model a simple system like Lunar Lander in two different notations, or in two different viewpoints in the same notation (for example, two UML diagrams). Ensure that the models are consistent, and explain why and how you established that consistency.
8. Choose a concern or view not supported by one of the extensible ADLs and develop an extension to support that concern or view.

6.10 FURTHER READING

The chapter opens with an insight about modeling from Maier and Rechtin's *The Art of Systems Architecting* (Maier and Rechtin 2000). While not solely about modeling, this book addresses architecture from the perspective of a systems engineer, which is, in general,

broader and less concrete than the perspective of a software engineer. As with much of the content of this book, Maier and Rechtin's advice is often equally applicable to both systems and software development.

The bulk of this chapter introduces and surveys a broad variety of modeling notations. Although we have endeavored to distill each notation down to its essence and comment on the aspects of the notations that we believe are distinctive, the few pages dedicated to each notation often cannot adequately capture its scope and subtlety. Interested readers should investigate source documents and specifications related to these notations for more in-depth treatments. Tools for working with unconstrained graphical notations such as PowerPoint (Microsoft Corporation 2007), Visio (Microsoft Corporation 2007), and OmniGraffle (The Omni Group 2007) are widely available. Many books have been written about UML, but the canonical reference is from the developers themselves (Booch, Rumbaugh, and Jacobson 2005). Medvidović and Taylor surveyed early ADLs (Medvidović, and Taylor 2000). These included Darwin (Magee et al., 1995), Wright (Allen and Garlan, 1997) [treated even more extensively in Allen's Ph.D. thesis (Allen 1997)], and Rapide (Luckham et al. 1995), which inspired

Luckham et al.'s later work in complex event processing (Luckham 2002). UML's takeover of modeling most of the generic architectural views resulted in second-generation ADLs like Koala (van Ommering et al. 2000) and AADL (Feiler, Lewis, and Vestal 2003) [itself based on an earlier ADL, MetaH (Binns et al. 1996)], that addressed more domain-specific concerns. An exciting direction in ADL development remains the creation and growth of extensible ADLs, from early examples like Acme (Garlan, Monroe, and Wile 1997) and ADML (Spencer 2000) to later, more flexible examples such as xADL 2.0 (Dashofy, van der Hoek, and Taylor 2005).

Clements et al. (2002) have dedicated an entire book to the issue of documenting software architectures, which focuses primarily on modeling. Their book takes a viewpoint-centric approach, surveying a wide variety of viewpoints and providing guidance as to what should be captured in each. Their book also surveys salient notations and standards.

Visualization

The previous chapter covered modeling: how we capture the design decisions that make up a software system's architecture. In practice, there are often different ways to display and interact with the information contained in those models. This is the domain of architectural *visualization*.

Definition. An architectural *visualization* defines how architectural models are depicted, and how stakeholders interact with those depictions.

This is an intentionally broad definition of visualization. Here, visualization consists of two key aspects: *depiction* and *interaction*. Put simply, a depiction is a picture or visual representation of architectural design decisions in a particular format. Visualization tools can provide one or more interaction mechanisms through which users can interact with those decisions in terms of the depiction. These mechanisms may include keyboard commands, point-and-click operations, and so on.

This chapter discusses the relationship between architecture modeling notations and visualizations, and how various modeling languages are visualized. It then covers various strategies for designing and evaluating visualizations to maximize their effectiveness. The chapter ends with a survey of various visualization techniques and evaluations of the strengths and weaknesses of each technique.

This chapter is not intended to be a treatment of techniques for usability design or information visualization in general; these subjects are too broad for the scope of this book. Instead, the chapter focuses on identifying the kinds of visualizations that can be used for architectural models and discusses issues specifically related to visualization of architectures.

**Outline of
Chapter 7**

7 Visualization	
7.1 Visualization Concepts	7.3.4 Borrowed Symbol, Different Meaning
7.1.1 Canonical Visualizations	7.4 Evaluating Visualization Techniques
7.1.2 Textual Visualizations	7.5 Techniques
7.1.3 Graphical Visualizations	7.5.1 Textual Visualizations
7.1.4 Hybrid Visualizations	7.5.2 Informal Graphical Editors
7.1.5 The Relationship between Visualizations and Views	7.5.3 UML: The Unified Modeling Language
7.2 Evaluating Visualizations	7.5.4 Rapide
7.2.2 Constructing a Visualization	7.5.5 The Labeled Transition State Analyzer (LTSA)
7.2.3 Coordinating Visualizations	7.5.6 xADL 2.0
7.2.4 Beyond Design: Using Visualization Dynamically	7.6 End Matter
7.3 Common Issues in Visualization	7.7 Review Questions
7.3.1 Same Symbol, Different Meaning	7.8 Exercises
7.3.2 Differences without Meaning	7.9 Further Reading
7.3.3 Decorations without Meaning	

7.1 VISUALIZATION CONCEPTS

Visualization plays a critical role in software architecture. The most important message of this chapter is that the way architectures are *visualized* can be, to an extent, separate from the way they are *modeled*. The two are closely related; in fact, each modeling notation is associated with one or more canonical or native visualizations. (This will be discussed in more detail below.) Fundamentally, however, *a model is just organized information*. In the case of architectural models, the information consists of design decisions. Visualizations are the means by which the information in a model is given form: how it is depicted and how users interact with it.

A single architectural model can be visualized in any number of ways, and multiple diverse models can be visualized in similar ways. Thus, visualization can be used to hide (or at least smooth over) differences in back-end modeling notations. Second, visualizations can vary widely—many are graphical, but most ADLs have textual visualizations as well. Research has even been done in the area of esoteric visualizations, such as three-dimensional virtual realities (Feijs and De Jong 1998).

The goal of this section is to distinguish visualizations from their underlying modeling notations, introduce the kinds of visualizations that can be used to model architectures (textual, graphical, and hybrid), and then discuss the issues that arise when multiple visualizations (of all these kinds) are simultaneously applied to an architecture.

7.1.1 Canonical Visualizations

It is difficult to separate the abstract information in a model from the concrete ways in which that information is visualized. No information is completely divorced from visualization. However, we mentally make this separation often in our daily lives. A sign that depicts a stick figure pedestrian with a bar across it and a nearby sign that says “Do Not Cross”

are simply two different ways of visualizing the same information: one graphical and one textual.

From this perspective, architectural modeling notations are simply ways of organizing information. Every notation has at least one visualization that is directly and specifically associated with it. We call this visualization the notation's *canonical visualization*. Models with multiple views are often associated with multiple canonical visualizations—one per view; views will be covered in the next section.

Text-based ADLs (including XML-based ADLs) are natively expressed using text-based visualizations. Not all modeling notations are textual, however. PowerPoint and OmniGraffle models are manipulated entirely in graphical visualizations, and there is no easy way to extract a text-based depiction of the model. Natively, UML diagrams are primarily graphical. Some parts of UML, such as the Object Constraint Language (OCL) constraints that are used to constrain relationships between model elements, are textual, having a well-defined syntax and semantics all their own.

A common pitfall is for users to associate an architecture modeling notation only with its canonical visualization, or to view a notation and its canonical visualization as the same thing. They are not: The notation is a way of organizing (abstract) information and the visualization dictates how the information is depicted and interacted with. For example, most tools that deal with architectural models store the model in data structures in memory, associated with no specific visualization; it is not until an editor is invoked that this information is visualized. Canonical visualizations present the information in a way that is closely related to its organization, but remember that this is almost always not the only way to present the information.

Not all visualizations are optimal for all uses, so notations for which there exist multiple visualizations are generally preferable to those that have only a single canonical visualization. In complex projects, desired architectural goals may be easier to achieve by developing a new visualization for an existing notation than by developing or selecting an entirely new notation.

7.1.2 Textual Visualizations

Textual visualizations depict architectures using ordinary text files. These text files generally conform to a particular syntactic format, much like a .c or .java file conforms to the syntax of the C or Java language. (As we have discussed, architectural decisions can also be documented using natural language, in which case the textual visualization would only be constrained by the grammar and spelling rules of that language.)

Figure 7-1 shows two textual depictions of the architecture of a Web client consisting of only one component: a Web browser. The first depiction depicts an architecture in xADL's native XML format, and the second depicts the same exact architecture in xADLite. This is an example of different visualizations being applied to the same model. The XML visualization of the architecture is easily read, manipulated, and syntactically validated by XML tools. The xADLite visualization describes the same architecture (and is, in fact, directly derived from the same model), but is better optimized for human readability.

Textual visualizations have several advantages. They generally depict the entirety of an architecture in a particular notation in a single file. Hundreds of text editors are readily available that allow the user to interact with text files. Years of research have gone into technologies for parsing, processing, and editing structured text. When a textual syntax is

Figure 7-1.
Textual
depictions for an
architecture in
xADL and
xADLite.

```
<instance:xArch xsi:type="instance:XArch">
  <types:archStructure xsi:type="types:ArchStructure"
    types:id="ClientArch">
    <types:description xsi:type="instance:Description">
      Client Architecture
    </types:description>
    <types:component xsi:type="types:Component"
      types:id="WebBrowser">
      <types:description xsi:type="instance:Description">
        Web Browser
      </types:description>
      <types:interface xsi:type="types:Interface"
        types:id="WebBrowserInterface">
        <types:description xsi:type="instance:Description">
          Web Browser Interface
        </types:description>
        <types:direction xsi:type="instance:Direction">
          inout
        </types:direction>
        </types:interface>
      </types:component>
    </types:archStructure>
  </instance:xArch>
```

```
xArch{
  archStructure{
    id = "ClientArch"
    description = "Client Architecture"
    component{
      id = "WebBrowser"
      description = "Web Browser"
      interface{
        id = "WebBrowserInterface"
        description = "Web Browser Interface"
        direction = "inout"
      }
    }
  }
}
```

defined using a meta-language such as Backus-Naur Form (BNF), many tools are available to generate program libraries that can parse and check the syntax of text documents written in that language. Many text editors provide additional developer support for particular notations with features such as autocomplete and syntax checking as you type.

Textual notations have disadvantages, as well. Textual notations are good at depicting data linearly and hierarchically. (Think of a program in a language like C or Java: Linear ordering is done using lines from top-to-bottom, and hierarchical structure is captured using braces and indentation.) However, graphlike structures are not easily understood (by people) through a textual visualization. Additionally, text editors are generally limited to showing a contiguous screenful of text, with few options to organize the text differently (although some advanced environments may include features such as code folding that allow users to collapse a block of text into a single line).

7.1.3 Graphical Visualizations

Graphical visualizations depict architectures (primarily) using graphical symbols instead of text. Like textual visualizations, graphical visualizations generally conform to a syntax (this time of symbols instead of text elements), but they may also be free-form (high-level or overview diagrams of architectures are often free-form and stylistic).

Figure 7-2 shows two graphical depictions of the Lunar Lander architecture. The top depiction is a high-level overview of the lander and its mission. While this depiction lacks rigor or formality, it does convey useful information to stakeholders that are encountering the application for the first time. Such depictions are often used as conceptual overviews of complex applications, especially those consisting of many interconnected systems. Standards such as the Department of Defense Architecture Framework (DoDAF, discussed more extensively in Chapter 16) include such depictions—this depiction would satisfy the DoDAF “OV-1” view of the Lunar Lander architecture (DoD Architecture Framework Working Group 2004). Note, however, that it is ambiguous or misleading in several important ways. For example, the lander is tilted somewhat with respect to the moon’s surface, which may imply a two- or three-dimensional aspect to the game that does not exist in the final application. It also depicts Earth in the background, which may lead one to believe that communications from Earth play a role in the lander simulation.

The bottom depiction is a logical view of the Lunar Lander architecture, depicting its structure in terms of a component and connector graph. This view is more rigorous,

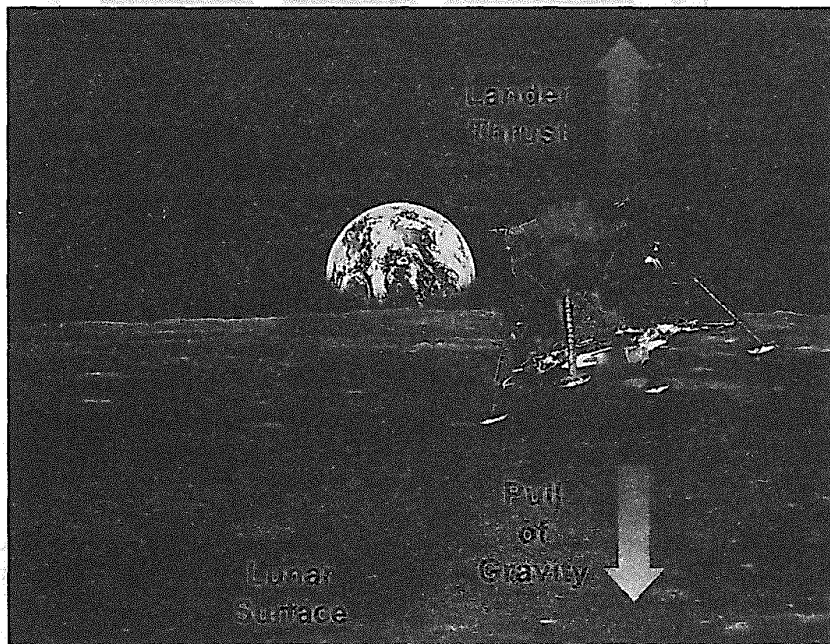
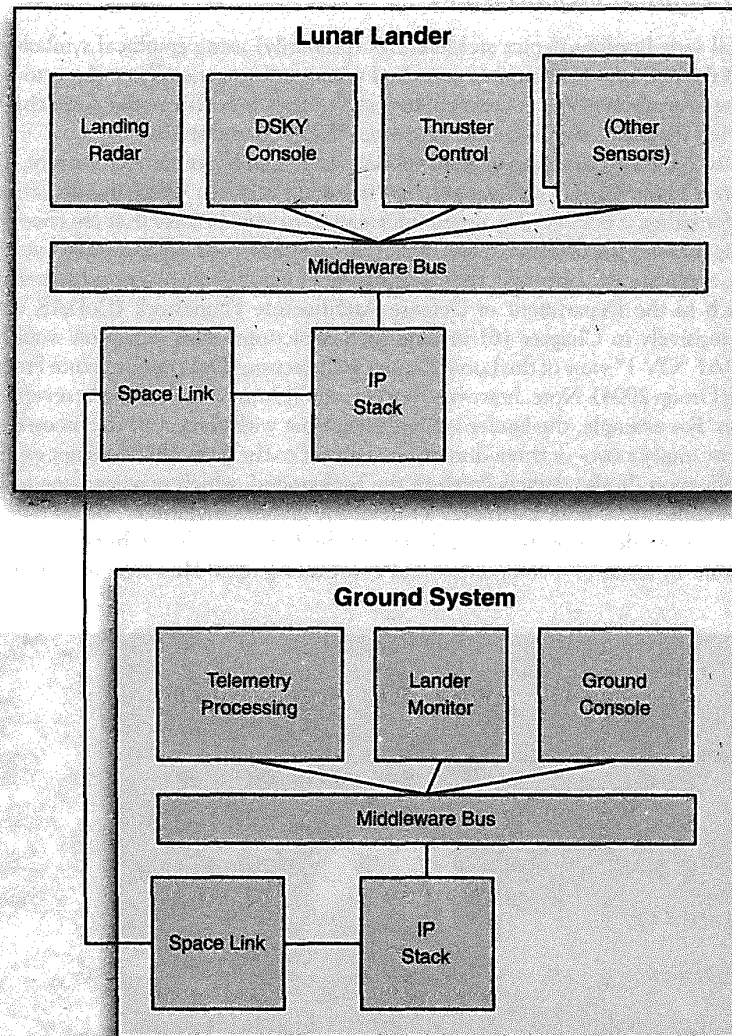


Figure 7-2.
Two graphical
visualizations of
the same
architecture.

(Continued)

Figure 7-2.
(Continued)



less ambiguous, and depicts more information at the cost of being less accessible to outside stakeholders. The overall mission and purpose of the system is lost in the detail of the components and their interconnections.

Graphical visualizations give stakeholders access to information about an architecture in many ways that textual visualizations cannot. Symbols, colors, and other visual decorations generally can be distinguished more easily than elements of structured text.

Nonhierarchical relationships between elements can be seen much more easily in a graph than a text file. Graphical visualizations can use spatial relationships to express relationships among elements. Advanced graphical visualizations may also employ animation or other visual effects to highlight or demonstrate different aspects of architecture. Options for interacting with graphical visualizations are generally superior to text visualizations, as well: Scrolling, zooming, drilling down, showing and hiding different levels of detail, and direct manipulation of objects with the mouse are all commonplace in graphical visualization tools. For example, a graphical environment might allow a user to connect components by simply drawing a line between their interfaces with the mouse.

A major disadvantage of graphical visualizations is the cost of building tools to support them. Many tools exist for creating graphical diagrams—PowerPoint, Visio, OmniGraffle, Photoshop, and Illustrator are a few of the more popular ones. However, these tools lack understanding of architectural semantics, and it is difficult or impossible to add appropriate semantics and interaction operations to these tools so they can be integrated into a wider software engineering environment. Furthermore, these tools generally have their own (usually proprietary) file formats and in-memory models that are difficult to connect to a more architecture-centric representation [(Goldman and Balzer 1999) (Ren and Taylor 2003)].

7.1.4 Hybrid Visualizations

Graphical and textual are rough ways of categorizing visualizations. Many visualizations blur the line between these categories. Few graphical visualizations use only symbols—generally, text is used to decorate, label, or explain the meaning of various elements. Some visualizations go even further, expressing some kinds of design decisions using graphics and others using text. For example, a UML class diagram is primarily composed of interconnected symbols, but constraints on relationships between the symbols are depicted in the Object Constraint Language (OCL) with an exclusively textual visualization.

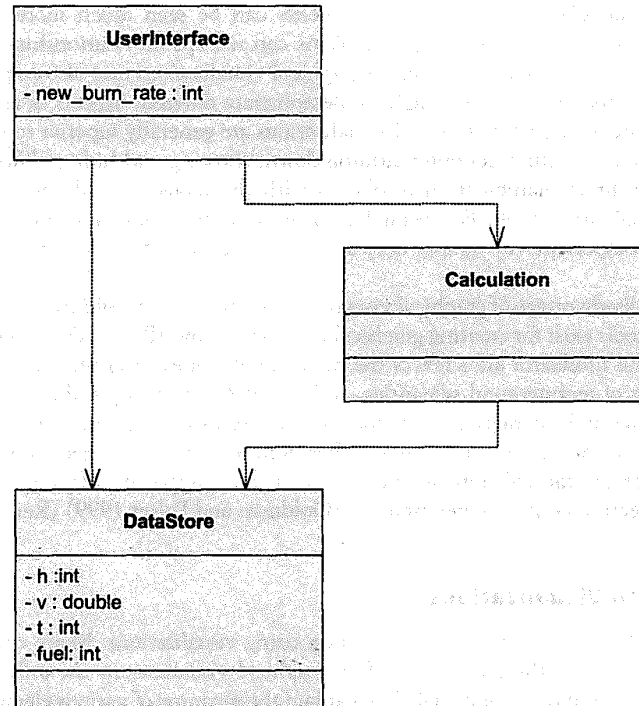
Figure 7-3 depicts UML as a hybrid visualization. The class diagram is primarily graphical, capturing the three primary Lunar Lander elements—user interface, calculation, and data store. Alongside the class diagram, textual OCL is employed to depict the constraint that the new burn rate must be nonnegative:

Some visualizations can be composites of many different visualizations, both graphical and textual. For example, the UML composite structure diagram is a primarily graphical visualization used to contain other UML diagrams. Such composite visualizations can be good for displaying relationships between different aspects of the same architecture. Composite visualizations can become complex and confusing quickly as different depiction and interaction mechanisms are combined. Strategies such as drill-down interaction mechanisms, where users can navigate to subvisualizations from a higher-level composite visualization, can mitigate this complexity.

7.1.5 The Relationship between Visualizations and Views

Some visualizations may depict the whole architectural model at once, but more often different visualizations are used to depict different views of the architecture. Chapter 6 presented the concept of views: subsets of the architecture, usually organized around a

Figure 7-3.
UML with
constraints as a
hybrid
visualization.



context UserInterface
inv: new_burn_rate >= 0

single concern or set of concerns. Recall the definitions presented:

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

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

Effectively, views and viewpoints let us consider different subsets of the design decisions in an architecture. We can apply the same concept of a subset to visualizations: A visualization for a viewpoint defines depiction and interaction mechanisms only for the kinds of design decisions included in that viewpoint. We associate visualizations with viewpoints rather than views because the same visualization can be used to visualize many different architectures; we do not create a new visualization for every architecture. For example, the UML class diagram is a visualization that can be used to visualize the class structure of many different applications. Two different class diagrams are not two separate visualizations; they are simply two instances of the UML class diagram visualization.

When a notation is associated with a set of viewpoints, it is often the case that each viewpoint has its own canonical visualization. UML is a good example of this: Each kind of UML diagram can be seen as a separate visualization. Although UML's canonical

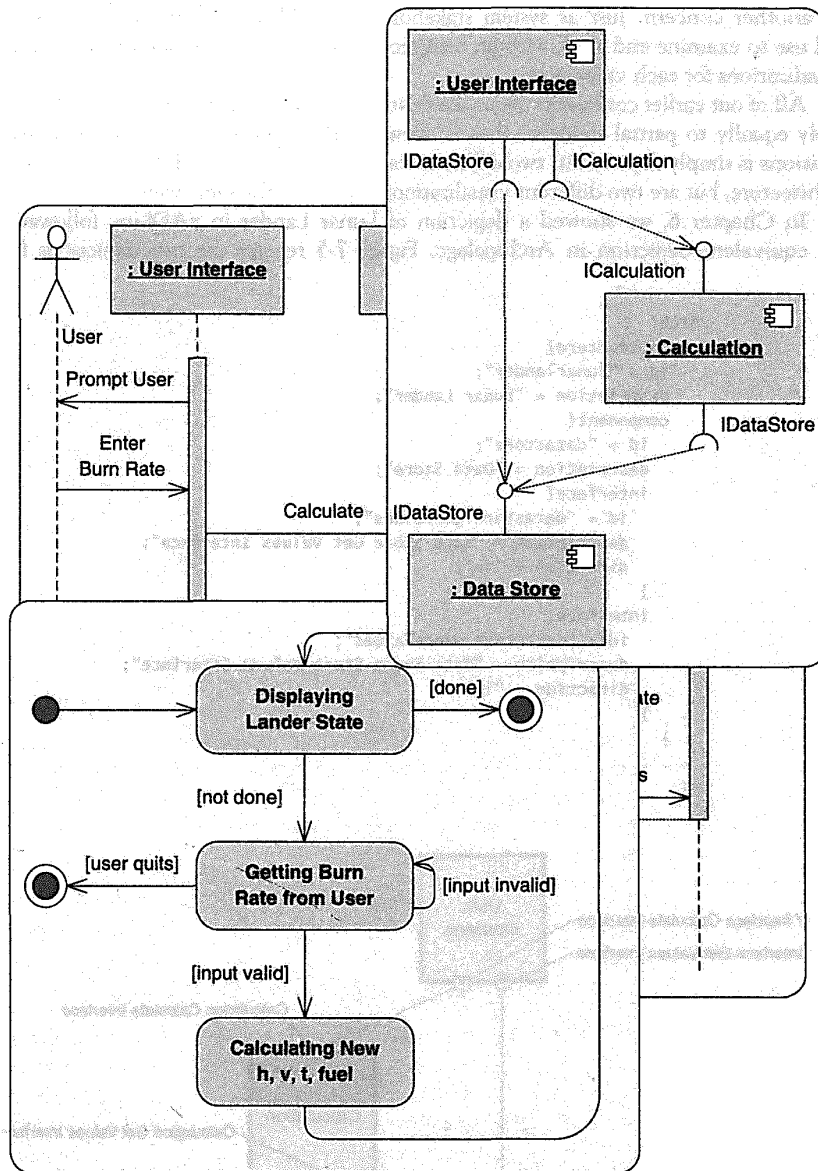


Figure 7-4.
Various UML
diagram types.

visualizations are all graphical, they differ widely. The box-and-arrow-style diagram used to depict components and their relationships bears little resemblance to the automata-like statechart or timeline-like sequence diagrams, as seen in Figure 7-4. This is a natural consequence of the fact that architectural models capture a wide variety of information about a system. What constitutes a useful visualization for one concern may be useless

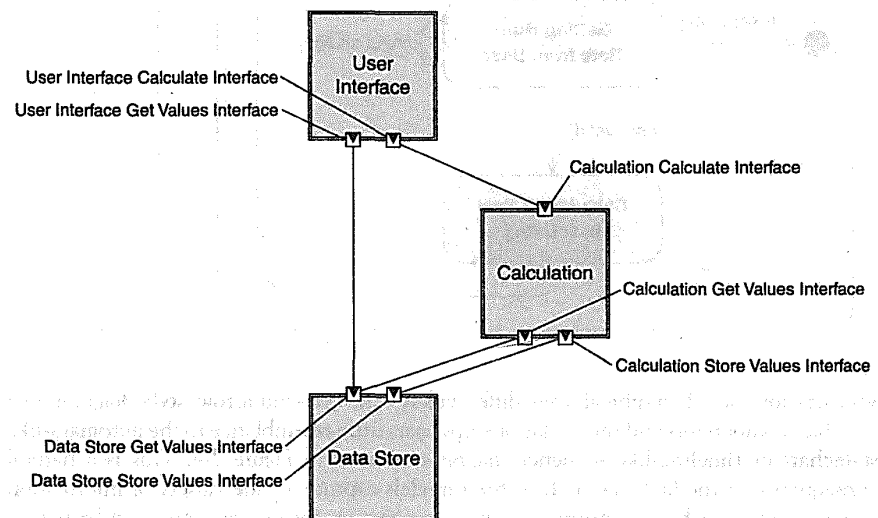
for another concern. Just as system stakeholders should identify the viewpoints they will use to examine and work with an architecture, they should also identify appropriate visualizations for each viewpoint.

All of our earlier comments about the relationship between visualizations and models apply equally to partial models—that is, views. If the same set of architectural design decisions is simply depicted in two different ways, these are *not* two different views of the architecture, but are two different visualizations applied to the same view.

In Chapter 6, we showed a depiction of Lunar Lander in xADLite, followed by the equivalent depiction in Archipelago. Figure 7-5 repeats the two depictions from

Figure 7-5.
Lunar Lander in
two different
visualizations: an
xADLite textual
visualization
(abbreviated for
space) and an
Archipelago
visualization.

```
xArch{
  archStructure{
    id = "lunarlander";
    description = "Lunar Lander";
    component{
      id = "datastore";
      description = "Data Store";
      interface{
        id = "datastore.getValues";
        description = "Data Store Get Values Interface";
        direction = "in";
      }
      interface{
        id = "datastore.storeValues";
        description = "Data Store Store Values Interface";
        direction = "in";
      }
    }
  }
}
```



Chapter 6 (although the extensive xADLite visualization has been abbreviated). This is an example of two different visualizations for the structural viewpoint being applied to the same subset of design decisions (that is, view) of the Lunar Lander architecture. Thus, we would not say that these are the xADLite view and Archipelago view of the Lunar Lander architecture; these are the xADLite and Archipelago *depictions* of the *structural view* of the Lunar Lander system.

7.2 EVALUATING VISUALIZATIONS

The above sections outlined what visualizations are, what kinds of visualizations exist, and the relationships between visualizations, models, views, and viewpoints. However, a key question now emerges: What makes a visualization “good?” How can one distinguish visualizations from one another and choose the best one? Ultimately, the worth of a visualization is dependent upon how well it fits the needs of a project’s stakeholders. As we have seen, stakeholder needs and priorities vary from project to project, so a visualization that is perfect for one project may be useless in another. Nonetheless, it is possible to identify some desirable qualities for visualizations, which can be prioritized by stakeholders to fit their specific situations. These qualities include:

Fidelity

Fidelity is a measure of how faithfully a visualization represents the underlying model. In general, the minimum acceptable fidelity for a visualization requires that information presented in the visualization be consistent with the underlying model. It would be extremely confusing, for example, if a structural visualization showed components that were not actually in the architecture, and so on. However, visualizations do not have to address *all* the information in an underlying model. Leaving some detail out can often make visualizations more effective by focusing attention on the parts of the model that matter in a given situation. Such is the case for visualizations that are associated with particular viewpoints, for example.

Fidelity affects both depiction and interaction. An interaction mechanism is faithful if it respects the underlying syntax and semantics of the visualized notation. For example, an interface that allows stakeholders to change the model in invalid ways may be confusing. A balance must be struck between fidelity and usability; preventing users from making mistakes entirely limits exploratory design.

Consistency

Consistency is a measure of how well a visualization uses similar depictions and interaction mechanisms for similar concepts. This sense is one of internal consistency (whether a visualization is consistent with itself) rather than consistency with the underlying model (which we call fidelity). In terms of depiction, consistent visualizations display similar concepts in similar ways. For example, in UML, an object is always depicted as a rectangle with an underlined name regardless of the context in which it appears. UML is not perfectly visually consistent, however: In most diagrams, a dashed open-headed arrow represents a dependency, but in a sequence diagram, it represents an asynchronous invocation or message. In terms of interaction, consistent visualizations permit the user to do similar things in similar ways. If double-clicking on one element allows the user to assign a name

to that element, requiring the user to right-click and select a menu option to assign a name to another kind of element would be inconsistent.

In general, more consistent visualizations are preferable to less consistent ones. The exception once again occurs at the extremes: Being too consistent might cause a visualization to have a huge and confusing variety of symbols (to make sure no two concepts share a symbol) or limit the conciseness of a visualization.

Comprehensibility

Comprehensibility is a measure of how easy it is for stakeholders to understand and use a visualization. This makes comprehensibility a function of both the visualization and the stakeholders who use it. Many factors contribute to comprehensibility, including the complexity of the visualization and how information is presented, the complexity of the interaction interface, and the skill sets and prior experiences of the stakeholders.

One way to improve comprehensibility is to narrow the scope of a visualization, limiting the number of concepts it tries to present and optimizing the visualization for only these concepts. Trying to display too much information at once—or many unrelated concepts simultaneously—complicates a visualization and increases its complexity. Alternatively, the comprehensibility of a visualization can be improved by leveraging stakeholder knowledge. For example, using a UML component symbol to represent components in non-UML diagrams can make a visualization more comprehensible to stakeholders that already have experience with UML. (Of course, this can backfire if stakeholders bring along assumptions about UML components that are not being implied by the use of the symbol.)

Dynamism

Dynamism is a measure of how well a visualization supports models that change over time; in Chapter 6 these are referred to as *dynamic models*. Information about changes flows two ways: changes to the model (from whatever source) can be reflected in the visualization, and changes to the visualization (through one of the interaction mechanisms) can be reflected in the model.

A range of possibilities exists here. An ideal dynamic visualization will be immediately updated when the underlying model changes from any source. Additionally, changes to the visualization through interaction mechanisms should cause the model to be updated accordingly.

In general, the depiction of a dynamic model will involve some kind of asynchronous animation; otherwise, the visualization will become inconsistent as the model changes. A less desirable alternative is to allow the user to manually refresh the visualization, optionally notifying the user when the underlying model has changed so that they can perform a refresh operation. With respect to interaction, any visualization that allows editing must be, to some extent, dynamic. Visualizations that update the underlying model in real-time, as the user works, are generally preferable to those that only synchronize changes periodically or at the user's request through, for example, a Save operation.

View Coordination

View coordination is how well one visualization is (or can be) coordinated with others. In general, environments that allow multiple visualizations to be presented and used simultaneously give users more insight and capability when designing or reviewing an

architecture. However, coordinating multiple visualizations is not always straightforward or easy. Strategies for coordinating visualizations are discussed in their own section below.

Aesthetics

Aesthetics is a measure of how pleasing a visualization is to its users. Aesthetics is not limited to depiction; user interfaces have aesthetic qualities as well. Here, depiction is the look and interaction is the feel of the visualization. Compared to other qualities, aesthetic qualities are extremely subjective. However, there is an enormous amount of literature available on evaluating and designing aesthetically pleasing displays of information. This comes from both the computer science community (for example, user interface design) and from other communities (art, advertising, marketing, and so forth). For example, color theory is instructive in choosing attractive and complementary colors for graphical visualizations. Determining which colors are complementary is easy with a color wheel and some basic knowledge of how color schemes are constructed, but difficult using intuition alone.

Technologists have a tendency to ignore or deprioritize aesthetic aspects of visualizations because they generally add little functional value. However, aesthetic qualities can often make the difference between a visualization being accepted or rejected by potential users.

Extensibility

Extensibility is a measure of how easy it is to modify a visualization to take on new capabilities, for either depiction or interaction. Just as underlying models and notations are often extended to support domain- and project-specific goals, visualizations of those models must be extended as well. A visualization that is difficult or impossible to extend will become less and less useful as underlying models expand to take on new concepts.

Mechanisms to support extensible visualizations include plug-in APIs, scripting support, and even simply open-sourcing the code that implements the visualization so others can modify it.

7.2.2 Constructing a Visualization

By now, it should be clear that the kinds of concepts that can be captured in an architecture are diverse and complex. They range from structural components and connectors to their interfaces to the schedules according to which they will be developed. Stakeholders choosing to include these elements in their architectures will also have to choose how they are depicted and manipulated in various visualizations.

If a preexisting or off-the-shelf notation is used to capture the architecture, its canonical visualizations will be available. For example, UML captures the notion of a class and has a specific symbol used to depict that class.

When decisions are captured that do not have a canonical visualization, or the canonical visualization is insufficient or inadequate, stakeholders have the option of constructing new visualizations. Creating good visualizations is somewhat of an art form, but there are a few things that can help.

Borrow Elements from Similar Visualizations. Even if you choose not to use UML to capture your architecture, it may be valuable to borrow certain symbols or conventions from UML, such as the shape of a package symbol to depict one of your packages or the

closed white-headed arrow to depict a generalization relationship. This has the advantage that many users already will be familiar with the depiction and its meaning. However, there are also drawbacks: Users may assume that your diagrams *are* UML (when they are not) or they may assume specific semantics that you did not mean to import when you used the symbol. A good visualization will strike a balance. One good source for generic symbols that do not carry extensive semantic implications is flowcharting. Although flowcharts have fallen further and further into disuse as programs have become more complex, they are still well understood by a wide variety of users. Common flowcharting symbols useful outside the context of flowcharts include the diamond (decision point), the vertical drum (disk storage), the sideways drum (memory storage), and so on.

Be Consistent among Visualizations. If you are depicting the same concept in many visualizations, use similar symbology. Likewise, try to avoid using the same symbology to depict different concepts in different visualizations.

Give Meaning to Each Visual Aspect of Elements. In a diagram depicting many components, it is tempting to assign different colors to components just so the diagram will not look too monochromatic. While this may be aesthetically pleasant, it is confusing from a semantic perspective since the visual aspect of color has no relationship with the underlying architectural model. It is a good idea to use visual decorations, but each decoration should have precise meaning.

On a related note, users have a tendency to (often subconsciously) embed real semantic information in visual decorations without making that information explicit in the architectural model. For example, in a box-and-arrow graph depiction, a user might place components close to one another to indicate that they share functionality or arrange the components in a layered fashion to reflect an implicit understanding that there are layer-like dependencies in the architecture. When this occurs (and the relationships are not formally documented), valuable information becomes embedded and lost in the visualization. Here, stakeholders should consider whether these visual relationships have semantic importance and, if so, find a way to include them explicitly in the system's models.

Document the Meaning of Visualizations. While we would all like to think that our diagrams and other visualizations are self-explanatory, this is generally not the case. Documenting what each aspect of the diagram means, using a legend, design document, or organizational standard is key to reducing confusion among stakeholders. At best, each aspect of the visualization should correspond to a piece of information in the model.

Balance Traditional and Innovative Interfaces. It is fair to assume that most stakeholders involved in software design will have used a significant amount of software themselves. As we have pointed out, borrowing well-known depiction and interaction techniques allows users to leverage their previous experience. However, adhering too closely to this guideline will result in stagnant visualization design. From time to time, consider borrowing useful nontraditional and innovative visualization features as well, or even developing one's own.

For example, most users will assume that a box-and-arrow graph visualization for an architecture's structure will look and work like that of PowerPoint or Visio. However, it is almost certainly not the case that PowerPoint and Visio have perfected box-and-arrow graph editing. Here, one could consider advanced layout paradigms such as fish-eye layouts, where information is displayed at large sizes in the center of the display and at smaller sizes at the edges, or drill-down paradigms, where zooming in is used as a visual metaphor for

looking at more detailed information. A good source of inspiration in visualization design is other software packages that are outside the realm of software design, such as CAD applications, video games, and so on. Aspiring visualization designers should take note of unusual but useful user interfaces that they encounter and determine how to apply those design ideas to architecture visualization.

Edward Tufte, professor emeritus of statistics, information design, interface design, and political economy at Yale University, is best known for a series of influential books, including *The Visual Display of Quantitative Information* (Tufte 2001) and *Envisioning Information* (Tufte 1990), that offer practical advice on how to visualize complex data sets to maximize impact and effectiveness. While Tufte's work focuses primarily on static depictions of statistical and quantitative data, many of his lessons can be useful in the design and evaluation of software architecture visualizations as well.

Tufte's approach can be summarized as one of parsimony. A core concept in Tufte's work is maximizing the *data-ink ratio*—designing depictions that focus on depicting the target data and interpretation aids with as few distractions, decorations, or embellishments as possible. He coined the term “chartjunk” to refer to such elements that detract from the display of data. Chartjunk may be obvious—the use of meaningless colors, shadows, photographic or textured fills, and so on. Chartjunk can also be subtle—excessively bold grid lines in a background, thick borders around data elements that diminish the perceived importance of the data, and so on. For Tufte, graphical depictions are not always ideal: Simple textual organizations of data (primarily tables) can often convey the same information with far less “ink” and far fewer distractions.

Tufte's advocacy of parsimony does not necessarily equate to minimalism, however. Many of Tufte's examples, particularly where complex data sets are used, are extraordinarily dense and display multiple properties of the data simultaneously. Nonetheless, these depictions still focus strongly on the data itself. *Envisioning Information* in particular delves into issues related to the display of such data sets.

In *Envisioning Information*, Tufte generally advocates increasing the dimensionality of representations as a way to deal with complex data. Tufte opens by lamenting the limitations of “flatland”—the two-dimensional space provided by pieces of paper and computer screens. He presents several different strategies for breaking out of flatland, primarily through integrating additional spatial or temporal dimensions into depictions. In a two-dimensional medium, perspective can be used to create three-dimensional spatial pictures. At the time of the book's publication in 1990, three-dimensional rendering technologies were still somewhat inaccessible, and many examples in the book show three-dimensional depictions hand-drawn by artists, although a few simple computer-generated examples appear as well. Another way to add spatial dimensions to flatland is through layering: breaking up the data into multiple depictions that are combined or overlaid on one another. Temporal dimensions can also be taken into account by showing the same data several times in different states, representing different snapshots in time. Spatial and temporal dimensions can be added through a concept Tufte calls “small multiples”—showing many small, simple depictions of related data (or the same data at different times) next to each other in succession, making it easy for readers to compare at a glance.

These techniques (three-dimensional depictions, layering, time variation, and small multiples) can all be applied to the construction of software architecture visualizations, although it is certainly the case that few visualizations take advantage of them today. This also represents an area of ongoing research; for example, the EASEL project (Hendrickson and van der Hoek 2007) is focused on bringing interactive layering concepts to an xADL-based architecture development environment.

Applying Edward Tufte's Lessons to Software Architecture Visualization

Tufte's work focuses primarily on static depictions of information, but a new set of possibilities opens up when the interactivity of a computer is brought to bear. Computers, particularly modern PCs with fast processors, high-resolution color displays, and specialized graphics hardware, can be used to create interactive visualizations that provide an even more immersive escape from flatland. Although still confined to a fundamentally two-dimensional screen, three-dimensional rendering technology is now available to everyone, and three-dimensional depictions no longer have to remain fixed on a page. A user can easily take a three-dimensional model and rotate, examine, and manipulate it in virtual space from any angle using a computer. Three-dimensional views of software architectures have not yet made a mainstream impact. This may be because they are not mature enough, or it may be that they are simply not useful for depicting the kind of data that comprises an architectural model. Projects like EASEL, cited above, not only break down architectures into layers, but allow live interaction with those layers, turning them on and off, using them to express alternatives, and so on. Animation has been used in limited ways to provide a temporal dimension to architectural visualizations (usually effect visualizations that result from some sort of simulation) but far more work needs to be done here as well.

Tufte and the designers of architecture visualizations have much to learn from each other. Tufte's work provides valuable insights on depicting information far more complex than the average architectural model in a very limited medium (static two-dimensional paper). Even with these restrictions—and perhaps because of them—he advocates exploiting dimensions that are still not well utilized by modern architecture visualizations. On the other hand, architectural visualizations, especially those implemented in an editor or other tool, can exploit the resources of fast computers with high-resolution displays to create much more dynamic visualizations than are possible on paper. Those wishing to design new visualizations (or evaluate existing ones) would do well to absorb Tufte's lessons and contextualize them in terms of an interactive computer-based medium.

7.2.3 Coordinating Visualizations

When multiple visualizations of the same information are available, it is key to coordinate these visualizations with each other, so that changes to the information via one visualization are accurately reflected in other visualizations. If the visualizations are not coordinated, they can become out of sync and cause confusion.

It is important to distinguish the *coordination of multiple visualizations* from *maintaining architectural consistency*. Here, we are only dealing with ensuring that multiple visualizations of the same (parts of the) architectural model are up to date with respect to the model. Inconsistencies and conflicts between design decisions stored in the model are a separate issue.

Stakeholders must decide how and how much to allow multiple visualizations to display the same architectural information at the same time. If users are only allowed to view information through one visualization at a time, visualizations can be synchronized with the architectural model when they are called up. They can also assume that the model will not change due to some external influence while they are active—any changes to the model will be made through this single visualization.

However, if the same information can be visualized in many ways simultaneously, it is generally a good idea to synchronize the visualizations in real-time so that they accurately depict the underlying model. This situation is much more complicated, since any visualization can change the model, and the other visualizations must respond appropriately to

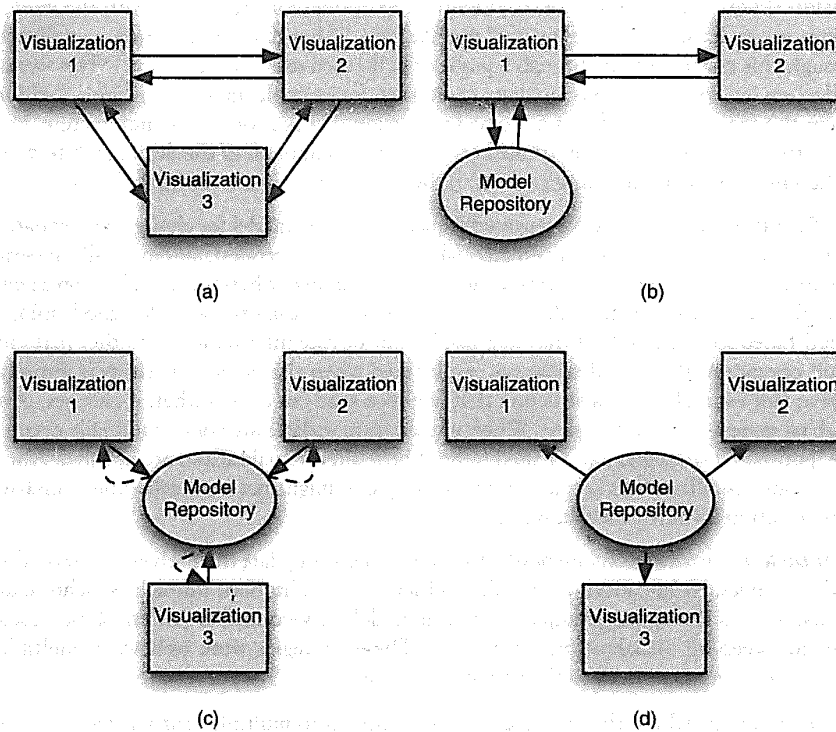


Figure 7-6.
Multiple
strategies for
coordinating
visualizations of
the same
information:
(a) peer-to-peer,
(b) master-slave,
(c) pull, and
(d) push.

that change. If the visualizations include both depiction *and* interaction, then both the depiction and the interaction state of the visualizations must be updated. This might mean changing editing modes, updating menu options, and so on.

Coordinating multiple visualizations can be accomplished through many well-known methods, depending on the situation. For situations where information may be visualized through only one visualization at a time, a simple import-export method usually works best: Initial depictions are created when the visualization is called up and stored (if necessary) when the visualization is dismissed.

Situations where multiple simultaneous visualizations are allowed are harder to deal with. In these cases, four general synchronization strategies are available (see Figure 7-6):

Peer-to-Peer. The visualizations maintain their own copies of information from the model, know about each other and explicitly notify one another about changes. These strategies can be brittle because they tend to tightly couple visualizations. They require many point-to-point dependencies. The number of dependencies is $\frac{v(v-1)}{2}$ where v is the number of visualizations. This grows exponentially as the number of visualizations increases. Because of this, the peer-to-peer strategy is most suitable for a small, fixed number of visualizations chosen in advance.

Master-Slave. One visualization is primarily responsible for interacting with the model repository, and it serves as the “master” visualization. Other slave visualizations coordinate through this master, either through a push- or pull-based strategy (see below). This works well when one visualization is auxiliary to another; for example, imagine a graphical editor where the main window shows a zoomed-in version of a portion of the architecture, but the corner of the window is portioned off to show a thumbnail of the entire architecture at the same time (for providing context to the user or serving as a navigation aid).

Pull-Based. Each visualization repeatedly queries a shared model for changes and updates itself accordingly. This may happen manually—at the user’s request, automatically at periodic intervals, or in response to certain actions (for example, when the user clicks on a new visualization or attempts to make a change to a different visualization). One disadvantage of pull-based strategies is that they may display out-of-date information until they perform a pull operation. Pull-based strategies can be used when the model repository is entirely passive (for example, a data structure that does not send out events when it changes, or a database system without triggers). When visualization updates are computationally expensive, pull-based strategies can be used to limit how often visualizations are updated. Also, if only one visualization is actually visible at a time, it might not be worthwhile to update a visualization until the user calls it up.

Push-Based. Visualizations are notified and consequently update themselves whenever the model changes. Notifications are usually multicast to visualizations through asynchronous events. This is the strategy employed by the model-view-controller pattern. Push-based strategies keep all visualizations up to date. These strategies work well when multiple visualizations are presented to the user simultaneously.

In situations where the architecture is organized into multiple (partial) models, it is sometimes possible to coordinate access to these models through a single visualization, thus masking some of the differences between notations or combining the strength of multiple visualizations. Figure 7-7 shows an architecture whose component-and-connector structure is expressed in xADL 2.0, but the detailed design of each component is expressed in UML.

7.2.4 Beyond Design: Using Visualization Dynamically

As discussed above, architecture visualizations are primarily used to depict and allow interaction with the design decisions that comprise an architecture. However, more advanced and dynamic visualizations can be used to gain an even deeper understanding of the architecture.

We now introduce the concept of effect visualizations.

Definition. An *effect visualization* is a visualization that does not represent architectural design decisions directly, but instead represents the *effects* of architectural design decisions.

For example, imagine an architectural model that contains enough information that it can be used as the basis for a behavioral simulation. The output or results of this simulation are not strictly an architectural model, since they are the results of the design decisions made in the architecture rather than the decisions themselves. Thus, visualizations of such results are not strictly architectural visualizations. These results, however, can be

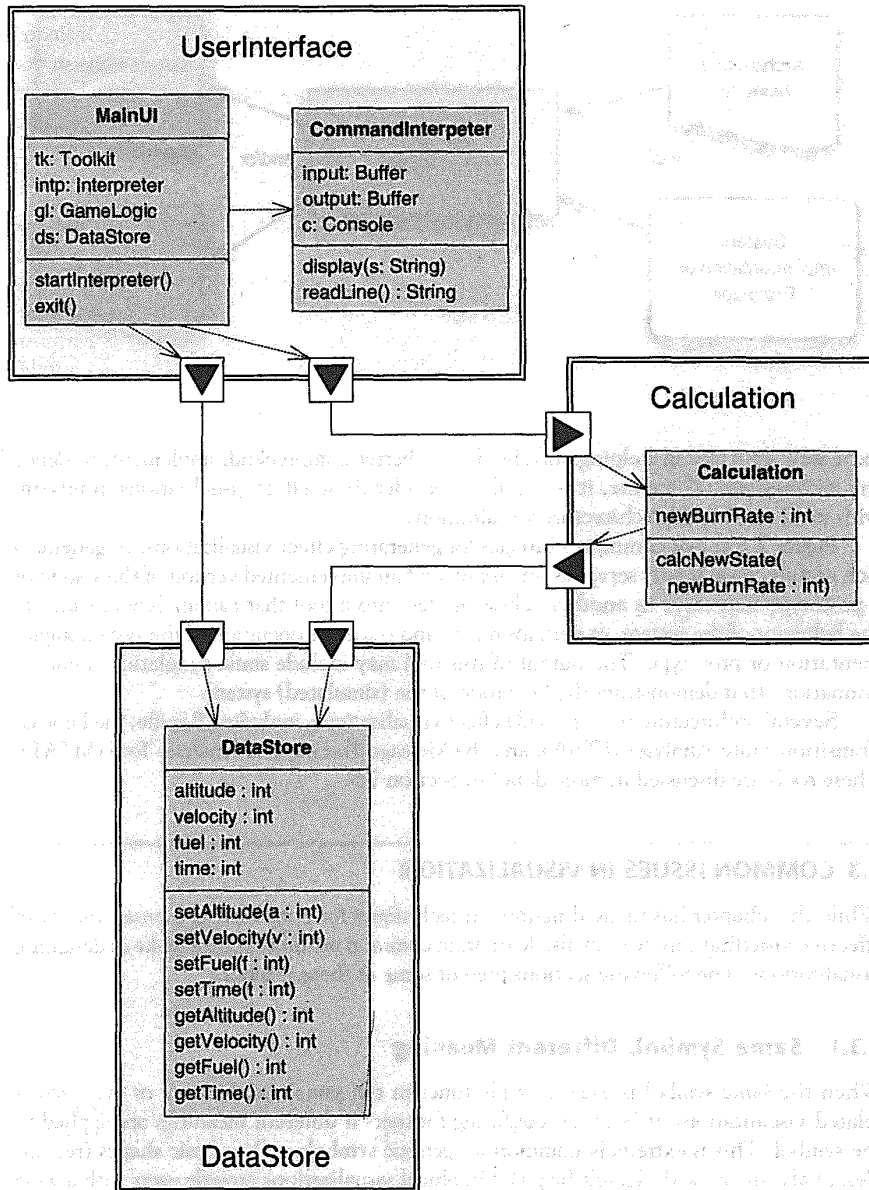
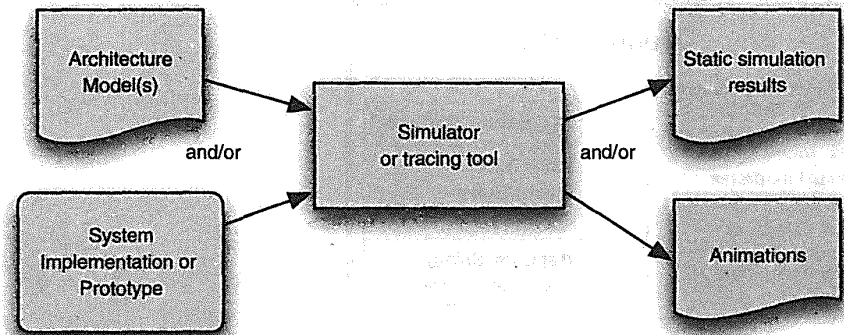


Figure 7-7.
Visualization
combining
elements from
xADL and
UML.

Figure 7-8.
Generating effect
visualizations.



extremely effective in helping stakeholders to better comprehend, implement, or debug the architecture. Therefore, it is useful to consider these effect visualizations in tandem with more traditional architecture visualizations.

Figure 7-8 shows common strategies for generating effect visualizations. In general, a rich architectural model serves as one input, and an implemented version of the system or a prototype may serve as another. These are fed into a tool that can analyze or simulate the behavior of the system, or perhaps record and trace the operation of the system implementation or prototype. The output of this tool may include static simulation results or animations that demonstrate the operation of the (simulated) system.

Several architecture tools provide effect visualizations, including Rapide, the Labeled Transition State Analyzer (LTSA), and the Message Tracing and Analysis Tool (MTAT). These tools are discussed in more detail in Section 7.5.

7.3 COMMON ISSUES IN VISUALIZATION

While this chapter has focused mainly on techniques for constructing a broad variety of effective visualizations, we can also learn from common mistakes people make in designing visualizations. The following sections present some of these mistakes.

7.3.1 Same Symbol, Different Meaning

When the same symbol is used multiple times in the same visualization, or even across related visualizations, it becomes confusing for users if different meanings are applied to the symbol. This is extremely common for generic symbols such as basic shapes (rectangles, ovals, arrows with default heads). Graphical visualizations provide users with a wide variety of ways of creating distinctive symbols—shapes, decorations, icons, borders, arrowheads, fills, and so on. All of these can be used to make visualizations richer and more precise.

Figure 7-9 shows a simple but deceptive diagram of a client-server system. Here, both the clients and the server are represented by the same symbol (a rectangle), even though

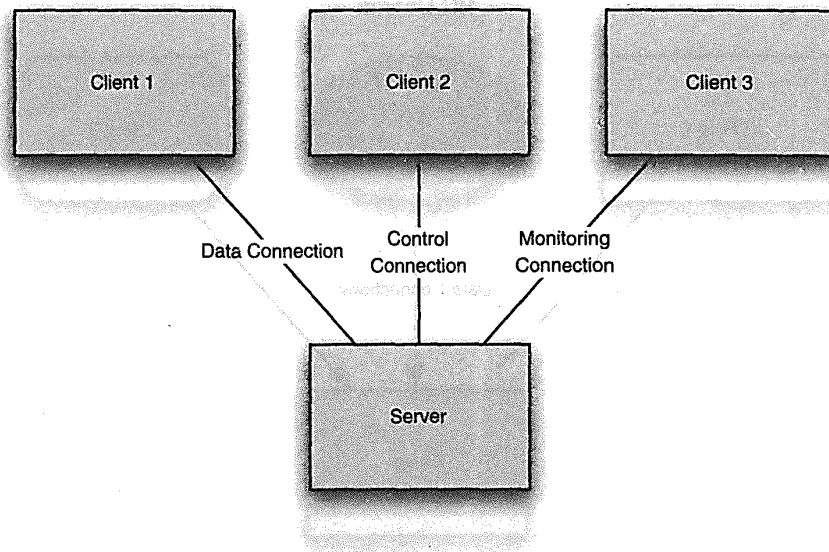


Figure 7-9.
Example of same
symbol, different
meaning.

they are distinct. The clients and server are all connected with the same type of plain line, although labels clearly indicate that these are different kinds of connections. Additionally, given the duties of the clients indicated by these connections—data, control, monitoring—it is possible that the clients are not even like one another, even though they appear to be.

7.3.2 Differences without Meaning

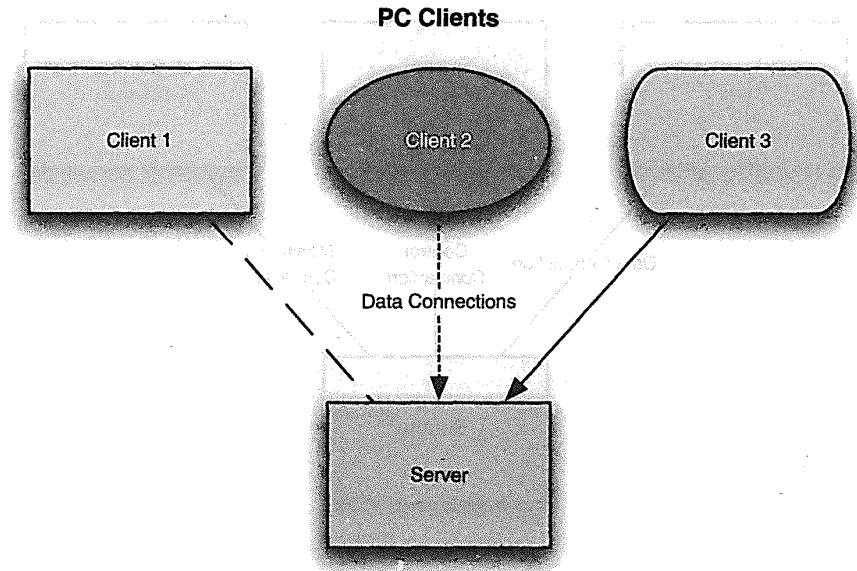
Graphical visualizations in which similar elements are repeated over and over (such as the same kinds of components or the same kinds of links) can often appear uninteresting or aesthetically flat. It is common to try to spice up such diagrams by adding decorations and other changes to symbols, primarily for their aesthetic value.

Figure 7-10 shows an example of this situation. Here, the otherwise uniform PC clients each are depicted with a different symbol. Different connection styles (including a random assortment of line styles and arrowheads) connect these uniform clients to the server, all for the same purpose. While this diagram does indeed look more interesting than the one in Figure 7-9, it only serves to confuse. It implies that there is heterogeneity among the clients, and that the connections between the clients and the server are substantially different.

7.3.3 Decorations without Meaning

Differences without meaning are not the only problem some graphical visualizations suffer. A related mistake is the consistent inclusion of visual decorations that indicate meaning but are not intended to convey it. A classic example of this is the use of double-headed arrows to indicate a simple connection between two symbols. The arrowheads imply directionality:

Figure 7-10.
Example of
“differences
without
meaning.”



that information or control is flowing in both directions across the connection. Often, the intended meaning is simply an association; in this case, the arrowheads only serve to confuse the issue by indicating flows where there are none.

Figure 7-11 shows a seemingly innocuous entity-relationship (ER) diagram. These diagrams are used to indicate various elements of a system and their quantity relationships to each other. For example, there is one business logic element in the system, associated with one ordinary server and many backup servers. However, the connections on the diagram are all two-headed arrows. Traditionally, ER diagrams do not include directional associations, since they are not meant to imply dependency, data flow, or any other meaning commonly associated with arrows. These additional decorations may imply things about the relationships between these elements that are simply not true.

7.3.4 Borrowed Symbol, Different Meaning

Visualizations are never truly interpreted anew; they are always seen through the lens of the user's previous experience and knowledge. Experienced users are familiar with a catalog of other visualizations and the symbols and meanings associated with those visualizations. Using symbols that are strongly associated with a different visualization is a good idea if the (rough) meaning of the symbol is brought along as well. Using the same symbol to mean something completely different is a recipe for disaster. For example, using the closed white-headed arrow in a diagram to mean “calls” will likely confuse UML users who will interpret that arrow as meaning “generalization.”

Figure 7-12 shows a simple logical layout for an application. In terms of the issues above, it fares well: three different kinds of symbols are used for three different kinds of

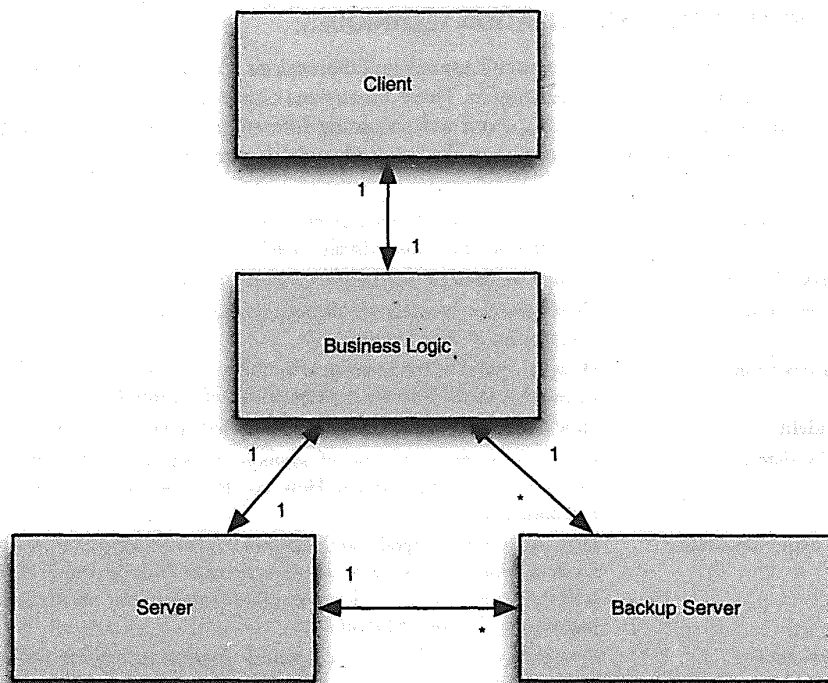


Figure 7-11.
Example of
decorations
without meaning.

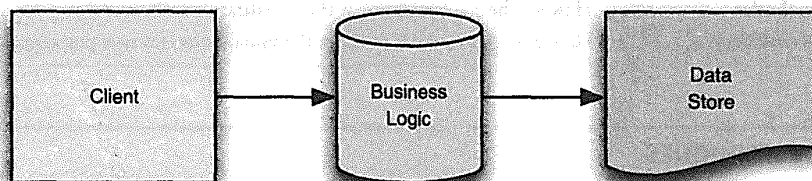


Figure 7-12.
Example of
borrowed
symbol, different
meaning.

components, only one kind of arrow is used and it indicates a calling dependency, and so on. The problem with this diagram lies in the choice of symbols. All these symbols are used in classic flowchart diagrams. The vertical cylinder, used in this diagram to represent the business logic component, is generally used in flowcharting to indicate a data store or disk. The box with a wavy bottom, used here to represent the data store component, is usually used to represent a document or data file. Users familiar with these interpretations might incorrectly infer that the data store was not a component, but simply a file, and that the business logic also acted as some sort of data store.

7.4 EVALUATING VISUALIZATION TECHNIQUES

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

Visualization Techniques Rubric	Scope and Purpose	What kinds of models can the technique visualize? Is it an architectural visualization or an effect visualization?
	Basic Type	What is the basic type of the visualization: Graphical? Textual? Hybrid?
	Depiction	How does the visualization depict the model? What are the basic constituents of the depiction?
	Interaction	How can stakeholders interact with the visualization? What kind of interface is used? What are the capabilities of the interface?
	Fidelity	How faithful is the visualization to the underlying model?
	Consistency	How consistent is the use of symbols in depicting information, especially across viewpoints? How consistent are the interaction mechanisms?
	Comprehensibility	How easy is it to comprehend the information being visualized? What kinds of information is the visualization optimized for conveying? How well does the visualization leverage existing stakeholder understandings to increase comprehensibility?
	Dynamism	How does the visualization use dynamic elements such as animation to depict changes over time?
	View Coordination	To what extent is the visualization of multiple views allowed? What strategies are used to coordinate the visualization of multiple views? How effective are these strategies?
	Aesthetics	How aesthetically pleasing is the visualization to its users?
	Extensibility	How easy is it to tailor or adapt the visualization for new purposes?

7.5 TECHNIQUES

This section surveys representative examples of a variety of architecture visualizations that are used in research and practice, from traditional textual and graphical visualizations to more exotic tools that use animation and effect visualizations.

7.5.1 Textual Visualizations

There are literally as many textual visualizations as there are text editors. Many models' canonical visualizations are text-based. The most rudimentary text-based visualizations are provided by editors such as Windows Notepad, or "pico" and "joe" on UNIX systems. These editors display architectural models in a structured text format, in a single font and color. Interactions in these basic text editors is limited—users can edit through rudimentary commands such as inserting and deleting characters, and copying-and-pasting blocks of text.

Before Code Folding:

```
[-] public int getAltitude(){
    ds = getDataStore();
    a = ds.getProperty("altitude");
    return a;
}
```

After Code Folding:

```
[+] public int getAltitude(){ ... }
```

(a)

```
component GameLogic{
    description = "my_description"
    interface{
        description = "my_description"
        direction = "none / in / out / inout"
    }
    behavior{
        my_behavior
    }
}
```

(b)

Figure 7-13.
Various advanced
techniques used
in text
visualizations:
(a) code folding,
(b) templates,
and (c)
autocomplete.

```
GameState st = application.getGameState();
st.
```

```
getAltitude() : int
setAltitude(int a) : void
getFuel() : int
setFuel(int f) : void
...
```

(c)

Enhanced text editors, such as those found in many integrated software development environments, support a similar base-feature set, but offer many improvements as well. These improvements are mostly available through the text editor having some internal knowledge of the syntax or semantics of the underlying notation. Common depiction enhancements include syntax coloring and code folding. Editors with syntax coloring support identify segments of text as tokens and color them to represent the type of token (keyword, character string, number, variable, and so on). Code folding is a technique whereby the editor can identify blocks of text and “fold” those blocks into a single line to reduce the amount of detail shown. Both these techniques require that the editor have an understanding of the syntax of the underlying notation. Common interaction enhancements include code completion and templates. Interfaces with code completion allow the user to type some or all of a token, and have the editor present options for completing the token to save typing. Templates allow the user to insert a block of text with placeholders and then enter data for each placeholder. Some of these depiction and interaction techniques are shown in Figure 7-13.

Scope and Purpose	Depicting and editing models that can be expressed as (structured) text.	Evaluation Rubric for Textual Visu- alizations
Basic Type	Textual.	
Depiction	Lines of text composed of characters; depending on the syntax characters will be grouped into ordered tokens.	
Interaction	For basic text editors, insert/delete/copy/paste. Enhanced editors may provide syntax/semantic-aware features such as syntax coloring, code folding, code completion and templates.	
Fidelity	In general, textual visualizations depict the entire model including all detail.	

Consistency	Depends on how well the language syntax is defined; a language with a poorly designed syntax might use the same token for different purposes in different contexts. Interaction mechanisms are generally consistent because they do not vary.
Comprehensibility	In general, textual visualizations are best for visualizing information that can be organized linearly or hierarchically. They have low comprehensibility for complex models with many interrelated elements, or elements organized in graph structures.
Dynamism	Depends on the editor in use; there are some integrated environments where model changes will cause immediate updates to textual visualizations.
View Coordination	In some integrated environments, textual visualizations can be coordinated with other visualizations of other views via a shared model.
Aesthetics	Depends on the syntax of the language, as well as how the visualization organizes and presents the text. Long, continuous blocks of text tend to be perceived as aesthetically poor; well-organized, attractively colored text tends to be perceived as aesthetically pleasing.
Extensibility	Depends on the editor; some editors provide well-defined extension points for adding in new language syntax and supporting features such as syntax coloring and autocompletion for new languages.

7.5.2 Informal Graphical Editors

PowerPoint and similar graphical editors [(Microsoft Corporation 2007) (Microsoft Corporation 2007) (The Omni Group 2007)] are commonly used to capture architectural design decisions despite the fact that these tools have no support for architectural semantics—they are simply diagram editors. With no real graphical syntax or semantics, the power and allure of informal graphical modeling is not derived from the model or the notation, but from its visualization. From a depiction standpoint, informal graphical diagrams are generally straightforward and aesthetically pleasing. Text, symbols, and bitmap graphics coexist in depictions as needed. Nothing is hidden; everything in the depiction is visible on a single page or slide. The page provides a natural limitation on how much detail can be presented at a time; by having finite boundaries, it limits the complexity of the visualized information. By building up a slide deck or using tools such as PowerPoint's animation capabilities, it is even possible to depict the evolution of an architecture over time.

The most attractive aspect of informal graphical editors, however, is their user interfaces. These editors are characterized by point-and-click interfaces that allow users to create and manipulate diagrams with great ease and flexibility. Symbols can be created and moved simply by dragging and dropping. PowerPoint's connector lines will even maintain their connections as the shapes they connect are moved around the canvas. Users can easily add media from outside sources as well: Bitmap graphics, vector graphics, screenshots, and even video and audio clips in nearly any format can be added to these diagrams with only a few mouse clicks.

Figure 7-14 shows PowerPoint being used to draw a Koala-style architecture. Although the visualization is attractive, certain problems are already apparent. The various elements shown are just independent shapes and text. For example, the interfaces on the components

Consistency	The user is entirely responsible for developing consistent representations; lack of consistency is a persistent problem in these visualizations.
Comprehensibility	The user is entirely responsible for the comprehensibility of the presented information.
Dynamism	Limited animation facilities are provided for depicting changes over time.
View Coordination	In general, it is extremely difficult to coordinate with other visualizations, or even connect elements on multiple pages/slides in the same file to one another.
Aesthetics	Aesthetics are the responsibility of the user; the wide palette of available symbols and decorations gives users enormous freedom to create aesthetically pleasing (or poor) visualizations.
Extensibility	In general, adding new symbols is straightforward; however, extending the user interface to provide new interaction options is more complicated. Scripting capabilities or published interfaces that allow extensions may be available.

7.5.3 UML: The Unified Modeling Language

UML as a notation was discussed in Chapter 6. UML has an associated canonical depiction that is primarily graphical. We have seen several examples of UML diagrams already, such as in Figure 7-4. Compared to the free-form nature of informal graphical editors that deal in boxes and arrows, UML diagrams are visualizations of more semantically meaningful elements such as classes and statechart nodes.

UML concepts are mapped to particular graphical symbols. For example, the “generalizes” relationship between elements is mapped to an arrow with a closed, white, triangular head. While certain concepts (and, as such, their associated symbols) are only present in certain diagrams, diagrams that incorporate common concepts use the same symbol in each diagram. For example, the “generalizes” relationship is present in both the class and object diagrams, and the same symbol (the closed white arrow) is used to represent the concept in both diagrams. Interestingly, although UML’s syntax is defined in a graphical meta-model, the UML meta-model does not actually map UML concepts to symbols; this mapping is described in other documentation.

Even when taking into account this additional documentation, UML is only associated with a canonical depiction and not a canonical user interface. That is, UML’s designers prescribe how UML diagrams should look, but not how the user should interact with them in tools or editing environments. These decisions are left up to individual tool vendors, and tools such as Rational Rose (Rational Software Corporation 2003) and ArgoUML (Argo/UML) have different mechanisms for manipulating and otherwise interacting with UML diagrams. Figure 7-15 shows a screenshot from ArgoUML editing a UML model. Note that the canonical graphical visualization is only one part of the environment. Another visualization of the model as a tree is present on the left side. Tools accessible from the menu and from the tabbed area below allow the model to be manipulated in different ways.

Tools such as Rose and ArgoUML provide compelling user interfaces, but often can blur the distinction between what is provided by the tool and what is provided by a modeling

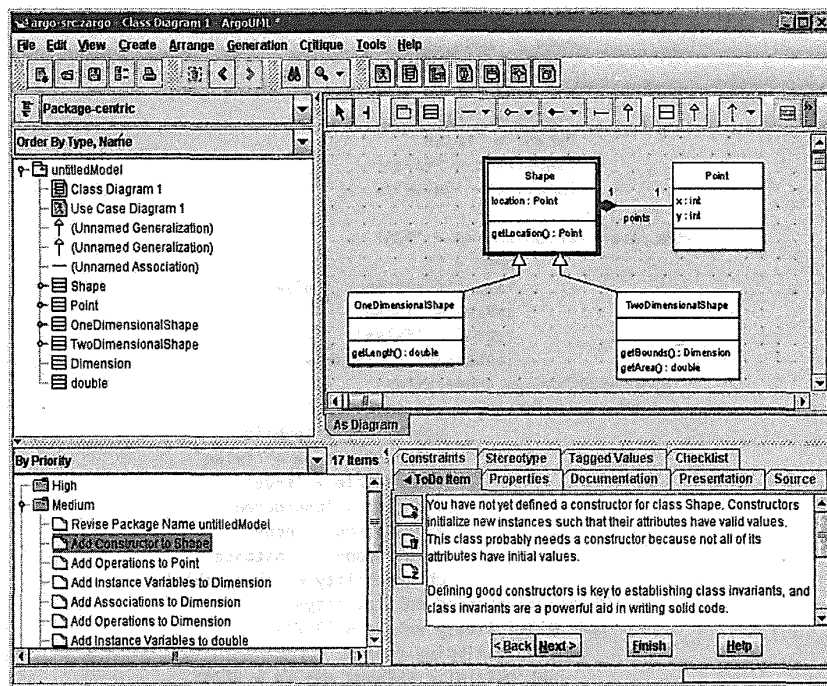


Figure 7-15.
Screenshot of
ArgoUML.

notation. For example, the screenshot in Figure 7-15 includes an editor for a to-do list. UML itself has no concept of a to-do list; this is a feature of ArgoUML. Users must be cognizant of these distinctions, as they can make it more difficult to integrate multiple tools operating on the same model, or to switch from one environment to another.

The canonical graphical visualization is not the only visualization available to UML users. Although it is not part of the core UML standard, it is possible to visualize UML data using text through the use of XMI (Object Management Group 2002). XMI is an XML-based format devised to facilitate interchange of models between tools. It is an interchange format in which a UML model can be encoded in XML. XMI helps to draw out the distinction between UML and its canonical graphical visualization. Plain XMI encodes only the information in a UML model, but not information about diagram layout from graphical visualizations. An extension to XMI that includes this information is provided in a separate standard. Figure 7-16 shows an excerpt of an XMI document generated by ArgoUML. Note the amount of text used to depict a simple point-to-point association, as well as the lack of information about graphical depiction, layout, or positioning.

One of the main advantages of UML's canonical graphical visualization is the (mostly) consistent use of symbols across diagrams and projects. The rationale behind this decision is that it increases the ability of stakeholders who are familiar with UML to quickly understand the meaning of diagrams in different contexts. In this regard, UML's popularity creates a network effect: The more people who use UML, the more valuable the consistent use of these symbols becomes.

Figure 7-16.
Excerpt of an
XML document
showing the
representation of
a UML class and
association.

```

<UML:Class xmi.id = '723'
    name = 'Data Store'
    visibility = 'public'
    isSpecification = 'false'
    isRoot = 'false'
    isLeaf = 'false'
    isAbstract = 'false'
    isActive = 'false' />

<UML:Association xmi.id = '725'
    name = ''
    isSpecification = 'false'
    isRoot = 'false'
    isLeaf = 'false'
    isAbstract = 'false'>
<UML:Association.connection>
  <UML:AssociationEnd xmi.id = '726'
    visibility = 'public'
    isSpecification = 'false'
    isNavigable = 'true'
    ordering = 'unordered'
    aggregation = 'none'
    targetScope = 'instance'
    changeability = 'changeable'>
    <UML:AssociationEnd.multiplicity>
      <UML:Multiplicity xmi.id = '727'>
        <UML:Multiplicity.range>
          <UML:MultiplicityRange xmi.id = '728'
            lower = '1'
            upper = '1' />
          </UML:Multiplicity.range>
        </UML:Multiplicity>
      </UML:AssociationEnd.multiplicity>
    <UML:AssociationEnd.participant>
      <UML:Class xmi.idref = '71F' />
    </UML:AssociationEnd.participant>
  </UML:AssociationEnd>

  <UML:AssociationEnd xmi.id = '729'
    visibility = 'public'
    isSpecification = 'false'
    isNavigable = 'true'
    ordering = 'unordered'
    aggregation = 'none'
    targetScope = 'instance'
    changeability = 'changeable'>
    <UML:AssociationEnd.multiplicity>
      <UML:Multiplicity xmi.id = '72A'>
        <UML:Multiplicity.range>
          <UML:MultiplicityRange xmi.id = '72B'
            lower = '1'
            upper = '1' />
          </UML:Multiplicity.range>
        </UML:Multiplicity>
      </UML:AssociationEnd.multiplicity>
    </UML:AssociationEnd>
  </UML:Association.connection>
</UML:Association>

```

```

<UML:AssociationEnd.participant>
  <UML:Class xmi.idref = '721' />
</UML:AssociationEnd.participant>
</UML:AssociationEnd>
</UML:Association.connection>
</UML:Association>

```

Figure 7-16.
(Continued)

UML's consistent use of symbols has limitations and disadvantages, however. Symbolic consistency in visualization cannot create semantic precision at the notational level. That is, if the concept of generalization is semantically ambiguous, the consistent use of the same arrow shape to depict the concept cannot repair this ambiguity. Additionally, the use of the same set of symbols across project and domain boundaries limits how much UML can be specialized for a particular domain. UML tools and standards do not generally support a lot of customization in UML visualizations for individual projects or domains. For example, Rational Rose allows users to associate stereotypes with graphical icons, but will not fundamentally change the symbol of the element to which the stereotype is being applied.

Scope and Purpose	Canonical visualization of UML models via (currently) thirteen diagram types.	Evaluation Rubric for UML Visual- izations
Basic Type	Hybrid: primarily graphical with some textual annotations and elements (such as OCL constraints)	
Depiction	Each UML diagram type has a canonical depiction. Each diagram type is composed of a handful of basic elements—the class diagram, for example, includes distinctive elements for classes, methods, different kinds of associations, and so on.	
Interaction	Depends on the tool in use; different UML tools have different interaction interfaces. Most allow point-and-click editing in the style of PowerPoint, and available editing options are generally guided by UML's syntax and semantics.	
Fidelity	As a canonical visualization, these depictions are generally completely faithful.	
Consistency	UML attempts to use consistent symbols, even across diagram types; there are small exceptions. (An open-headed dashed arrow might indicate a dependency in a class diagram or a message in a sequence diagram.)	
Comprehensibility	Most diagrams can be interpreted at a very general level with no prior knowledge. However, properly interpreting each diagram requires specific knowledge of the meaning of each of UML's symbols. Most diagrams have roots in earlier notations (for example, OMT and state-charts) that may be familiar to stakeholders outside the context of UML.	
Dynamism	Few UML tools use animation to depict changes in the underlying model.	
View Coordination	Multiple UML diagrams can be used to represent multiple views of a system, but there is no agreed-upon standard for linking common information in multiple views. Some editors allow multiple instances of the same element to be present in multiple diagrams simultaneously and will make changes to this element simultaneously in all diagrams.	

Aesthetics	UML diagrams use a limited set of relatively basic shapes to reduce complexity, although this can make diagrams seem overly simplistic and uniform. Some editors enhance this by allowing UML elements to be colored or enhanced with decorative icons to more clearly indicate element properties, for example.
Extensibility	Depends on the tool/editor in use; most support UML's basic extension mechanisms: stereotypes, tagged values, and constraints. Adding a new diagram type or extending UML's syntax would require major work, however.

7.5.4 Rapide

Effect visualizations, which depict the effects of architectural decisions rather than the decisions themselves, can be seen in the Rapide project (Luckham and Vera 1995). Chapter 6 shows several architectural models in Rapide's canonical textual visualization. However, the real power of Rapide comes from tools that allow users to run simulations of these architectural models. The Rapide simulator takes architecture models in the Rapide notation as input, and then simulates the interaction of the various components as defined by the behaviors specified in the model. Simulation runs generate a stream of events, some of which are causally related to one another. Because Rapide components run in parallel, the results of simulations are not strictly deterministic: Repeated simulations of the same architecture can generate different event streams depending on how the simulator's scheduler allocates time to the various components.

The result of a Rapide simulation is a directed graph of nodes, with each node representing an event and each edge representing a causal relationship between events. An example, originally presented in Chapter 6, is shown again in Figure 7-17. This can be seen as a kind of visualization of the system and its architecture. Even though it does not necessarily depict specific design decisions, it depicts the direct result of architectural design decisions and can equally serve to provide stakeholders with insights about the workings of the architecture.

Evaluation Rubric for Rapide	Scope and Purpose	Specification of architectural structure and component behavior in a textual format; display of simulation results in a graphical format.
	Basic Type	Architectural models are specified in a textual visualization; effect depictions are graphical.
	Depiction	Models in rigorous text format; effect visualizations are directed graphs containing nodes representing events and edges representing causal dependencies.
	Interaction	Models are edited in an ordinary text editor; effect visualizations are not editable.
	Fidelity	Textual visualization of the model is canonical; effect visualizations show one possible simulation run. Multiple simulations may produce different results in a nondeterministic system.
	Consistency	Limited vocabulary of symbols ensures consistency.
	Comprehensibility	Architectural models can be difficult to write and understand, although this is largely due to the complexity of the underlying notation. Effect

	visualizations are straightforward to understand, although complex systems may generate very large intertwined graphs that are difficult to interpret.
Dynamism	No support.
View Coordination	Effect visualizations are generated automatically from architectural models.
Aesthetics	Models are in a familiar programming-language style although the vocabulary of operators is fairly large. Effect visualization graphs are simple and unadorned.
Extensibility	Difficult to extend; Rapide and its tool set are a relative black box.

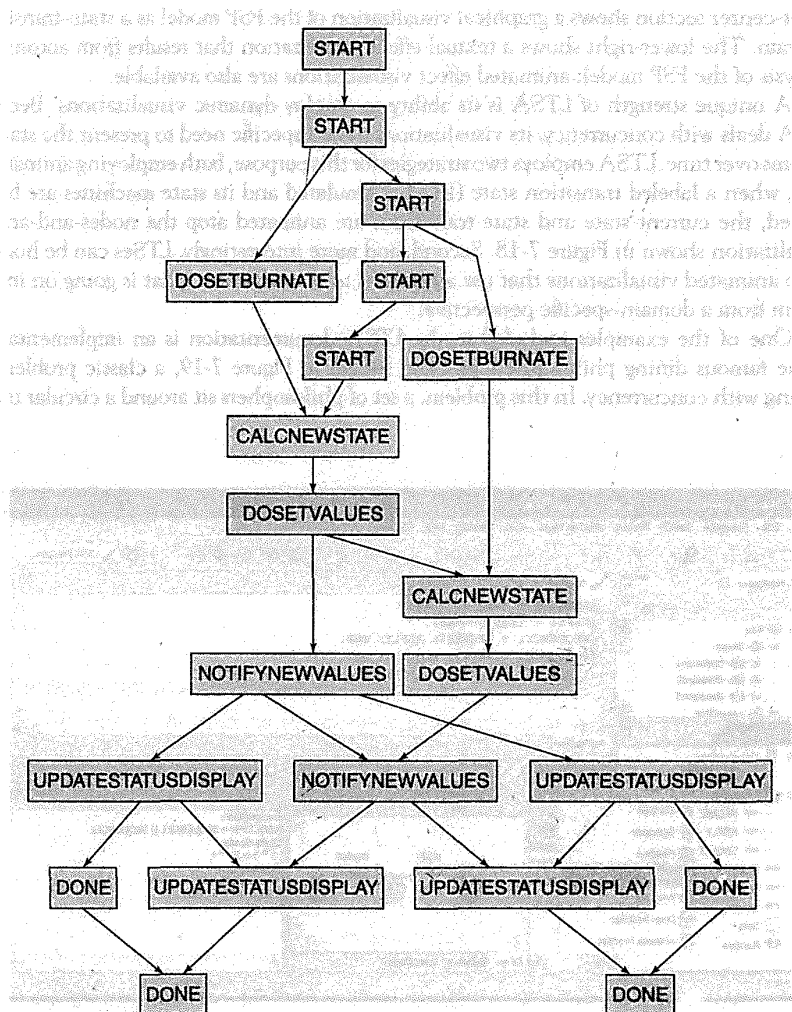


Figure 7-17.
Rapide effect
visualization of
the Lunar Lander
application.

7.5.5 The Labeled Transition State Analyzer (LTSA)

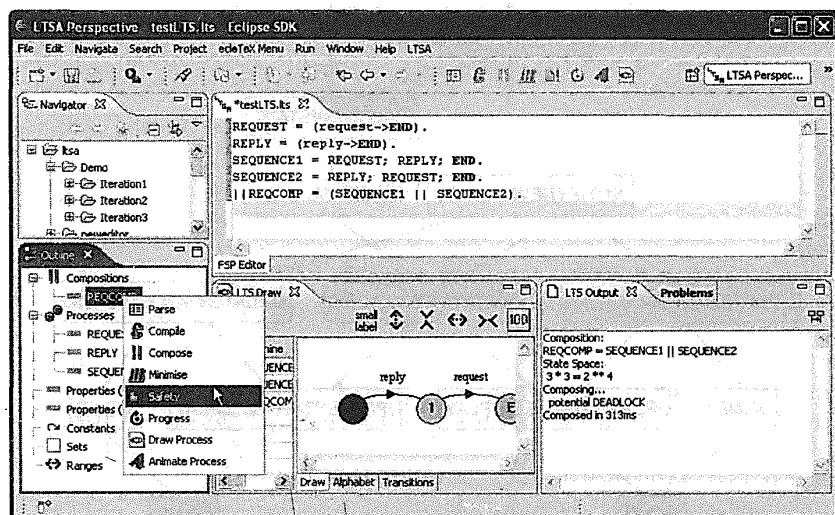
The Labeled Transition State Analyzer (LTSA) project from Imperial College in London, UK, is a way of analyzing and simultaneously visualizing concurrent systems (Magee and Kramer 2006). A system in LTSA is modeled as a set of interacting finite state machines. Users specify component behaviors in a compact process algebra called finite state processes (FSP); FSP is then compiled into state machines with labeled transitions. The LTSA tools can visualize these state machines graphically using a traditional nodes-and-arrows visualization.

Figure 7-18 shows a screenshot of the LTSA tool. Multiple concurrently maintained visualizations are shown. The upper-right area shows the canonical textual visualization of the model: raw FSP. To its left is a tree-based visualization of the project's organization. The lower-center section shows a graphical visualization of the FSP model as a state-transition diagram. The lower-right shows a textual effect visualization that results from automated analysis of the FSP model; animated effect visualizations are also available.

A unique strength of LTSA is its ability to employ dynamic visualizations. Because LTSA deals with concurrency, its visualizations have a specific need to present the state of systems over time. LTSA employs two strategies for this purpose, both employing animation. First, when a labeled transition state (LTS) is simulated and its state machines are being viewed, the current state and state transitions are animated atop the nodes-and-arrows visualization shown in Figure 7-18. Second, and more interestingly, LTSes can be hooked up to animated visualizations that use animation to directly show what is going on in the system from a domain-specific perspective.

One of the examples included in the LTSA documentation is an implementation of the famous dining philosophers problem shown in Figure 7-19, a classic problem in dealing with concurrency. In this problem, a set of philosophers sit around a circular table.

Figure 7-18.
Screenshot of the
LTSA tool.



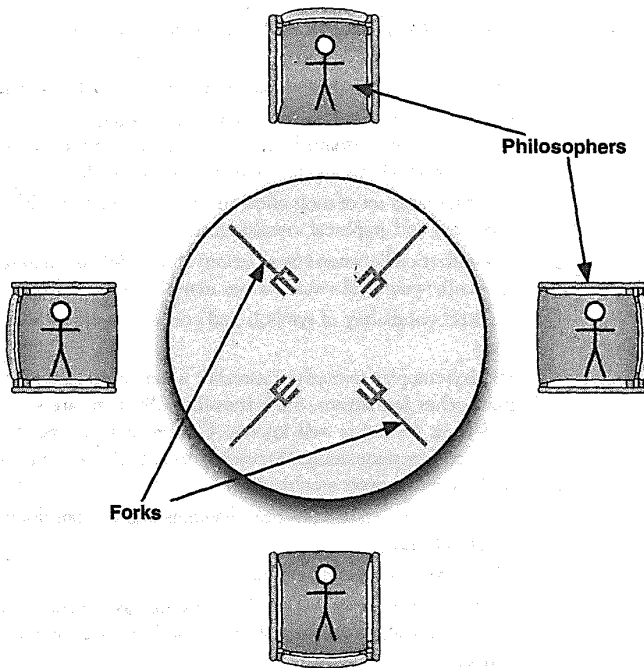


Figure 7-19.
Four dining
philosophers.

In between each pair of philosophers is a fork. Philosophers need to pick up both their adjacent forks to eat, but are also constrained by a set of rules as to when they can pick up and put down forks. The challenge in the problem is coming up with a set of rules such that all philosophers get to eat periodically and the system does not enter a locked state (for example, in which each philosopher picks up one fork and none will put one down).

The LTSA tools model the philosophers as communicating state machines, but can also visualize the models as a diagram of a table surrounded by philosophers, similar to what is shown in Figure 7-19. State changes, like a philosopher acquiring a fork, are visualized as an actual picture of a philosopher picking up fork. Simulations of other applications, such as air traffic control, use similar domain-specific visualizations hooked up directly to the component behavior simulators (such as pictures of airplanes circling and landing). Unlike Rapide, which only allows users to visualize event traffic in a graph after the fact, LTSA allows users to visualize the system being simulated in real-time, using animations and symbols that are directly drawn from the real domain. Although users of LTSA have to spend time and effort to construct these domain-specific visualizations for each application, their value comes from the fact that they go a long way to communicate the real meaning of abstract state machines. It may be difficult to understand the meaning of a particular state transition in a nodes-and-arrows diagram, but it is much easier to understand a picture of a philosopher actually picking up a fork.

Evaluation Rubric for LTSA	Scope and Purpose	Coordinated visualization of FSP models and different effect visualizations.
	Basic Type	Multiple textual and graphical model views and effect visualizations.
	Depiction	FSP models are visualized both in text and in graphical state-transition diagrams. Effect visualizations may be textual, animated on the state-transition diagrams, or custom and domain-specific.
	Interaction	An integrated set of tools allow the user to manipulate FSP models in both text and graphical visualizations.
	Fidelity	Coordination between visualizations and models are maintained automatically; graphical visualizations may elide some information.
	Consistency	Limited vocabulary of symbols and concepts helps to ensure consistency.
	Comprehensibility	FSP is a complex textual notation but is somewhat easier to understand than other formalisms. State-transition diagrams are straightforward and their format is well known. Custom domain-specific visualizations can increase comprehension substantially by tying models back to domain concepts graphically.
	Dynamism	Animation on state-transition diagrams and custom domain-specific visualizations.
	View Coordination	Views are coordinated automatically.
	Aesthetics	Well-known state-transition diagrams are easy to interpret; domain-specific visualizations tie abstract models to concrete real-world concepts.
	Extensibility	New custom domain-specific visualizations can be added as plug-ins.

7.5.6 xADL 2.0

The syntax of the xADL 2.0 language is defined in a set of XML schemas. As such, the canonical visualization for xADL 2.0 files is textual, in XML that conforms to the syntax prescribed by the schemas. One of the most interesting aspects of xADL 2.0, however, is that its canonical visualization is rarely (if ever) used or even seen by its users. Tools that support xADL 2.0 modeling provide a variety of alternative visualizations, both graphical and textual (Dashofy 2007). Some of these visualizations include the following.

xADLite

The xADLite visualization has been used throughout this book to describe various architectures. xADLite is a textual visualization that captures xADL 2.0 models using textual tokens organized in a manner similar to C-like programming languages: Hierarchical blocks are surrounded by curly braces, the “=” operator is used to denote assignment, double-quotes surround string values, and so on. This visualization was specifically crafted to capture all the data in a xADL 2.0 model in a compact format, using few extraneous characters. The programming-language-like symbology was chosen due to the popularity of this organization: Experienced software developers are comfortable with these symbols and organization and can read it easily without additional training. From a user-interface perspective, xADLite files are written and manipulated using standard text editors.

ArchEdit

ArchEdit, shown in Figure 7-20, is a tool that provides a semigraphical visualization of xADL 2.0 models. The document structure is depicted in a tree format with selectable nodes. When a node is selected, text attributes of that node are displayed for editing. Although the information in this view is organized hierarchically (much like the XML or xADLite visualizations), the user interface of ArchEdit is far more interactive. ArchEdit has a point-and-click interface that not only allows users to expand and collapse subtrees of the document, but also provides context-sensitive menus that provide the user specific options to add, remove, or manipulate elements. ArchEdit provides a *syntax-directed* visualization: the user interface and what is displayed on the screen are both derived, at least partially, from the syntax of xADL 2.0 itself. For example, when you right-click on an element in ArchEdit, it brings up a menu of children that can be added to that element. The list of available children is generated based on the definition of the xADL 2.0 language, and is not hardcoded in the tool itself. For a language with malleable syntax like xADL 2.0, syntax-directed visualizations and other tools become even more valuable. The primary disadvantage of (even the best) syntax-directed visualizations is that the syntax of the underlying notation drives how the information is visually presented. If the notation is treelike and hierarchical, it is likely that a syntax-directed visualization of the notation will also employ trees and hierarchy.

Archipelago

Archipelago, shown in Figure 7-21, is a tool that provides graphical visualizations of xADL 2.0 models. Archipelago's visualizations are semantically aware, meaning that specific

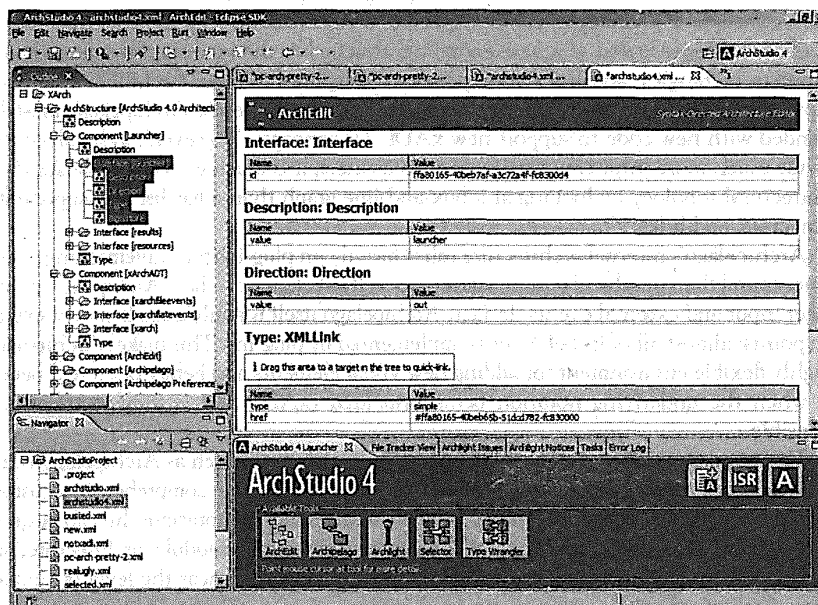
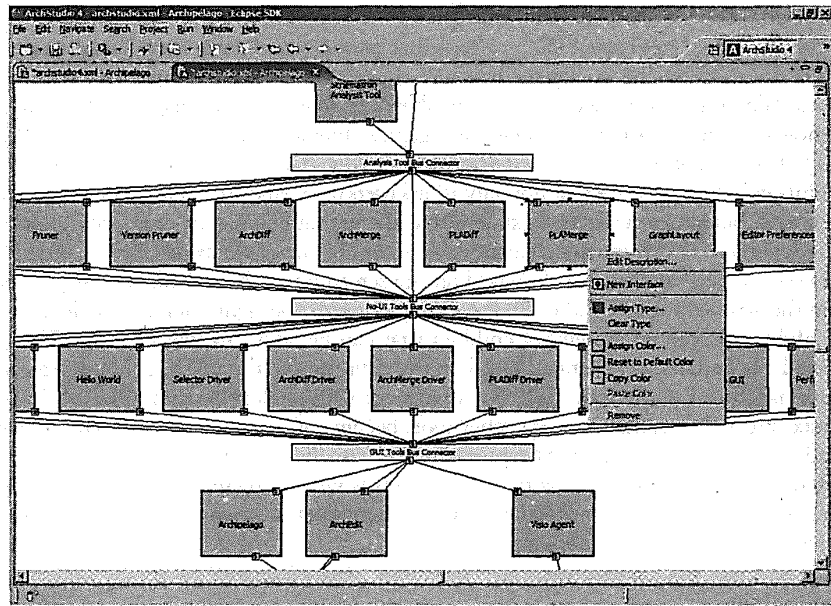


Figure 7-20.
ArchEdit
screenshot.

Figure 7-21.
Archipelago
screenshot.



depictions and behaviors are built into Archipelago to represent and interact with different xADL concepts. For example, components and connectors are represented as rectangles, interfaces are represented as square endpoints attached to the borders of component and connector rectangles, and links are represented as splines. Unlike ArchEdit, which can adapt its user interface automatically to new syntactic elements, Archipelago must be extended with new code to support new xADL 2.0 concepts. However, Archipelago can provide much more intuitive visualizations to its users; it is much easier to understand an architectural topology by looking at a box-and-line graph than a flat list of components, connectors, and links.

Archipelago's internal architecture relies heavily on plug-ins for implementing visual elements and their graphical representations as well as behaviors—how Archipelago reacts to user input and external events. In fact, Archipelago itself is really a small core of extension points; almost all of its behavior is implemented by plug-ins. This makes Archipelago a highly flexible environment for adding new visual elements and behaviors, a real necessity when the underlying notation being visualized (xADL 2.0) is itself modular and extensible.

The primary disadvantage of fully semantics-aware editors such as Archipelago is the expense of creating and maintaining them. Users expect intuitive, comprehensive, custom behavior tailored to individual notations, and for more extensive notations this can be quite costly to build. Archipelago attempts to limit this cost by using a modular architecture, but this cannot reduce the complexity of such editors to anywhere near the level of simpler syntax-directed editors.

Comprehensibility	Different visualizations have different levels of comprehensibility. Graphical visualizations tend to be easier to understand than textual notations, but they leave some information out. Different visualizations are optimized for displaying different kinds of information.
Dynamism	Visualizations such as MTAT use animation to depict the behavior of running systems; other visualizations are coordinated live so that changes in one visualization appear immediately in others.
View Coordination	Different visualizations are coordinated through a common model repository that maintains an in-memory representation of the underlying xADL model.
Aesthetics	Different visualizations attempt to increase aesthetic appeal by being specialized for depicting a particular kind of information.
Extensibility	New visualizations can be added by hooking into the common model repository; visualizations such as Archipelago are also extensible through their own internal plug-in mechanisms.

7.6 END MATTER

We have focused here on the role of visualization in software architecture-based development. Visualizations comprise depiction (how a set of design decisions is visually presented) and interaction (how stakeholders interact with, explore, and manipulate those depictions).

Perhaps the most important lesson from this chapter is that visualizations are not the same things as their underlying modeling notations. Every modeling notation has at least one canonical visualization, which may be textual, graphical, or a combination of both. It is difficult and sometimes counterintuitive to try to mentally separate the information content of a model from its canonical visualization, but making this distinction is useful. Once this distinction is made, it is possible to think about alternative or coordinated visualizations for the same model. It is also possible to separate the strengths and weaknesses of a modeling notation from the strengths and weaknesses of how that notation is visualized. This distinction also helps to explain phenomena such as the use of PowerPoint-like tools for architecture modeling. Here, the visualization is extremely mature and versatile, but the underlying model is devoid of semantics. This makes these tools attractive but dangerous to use in the long run.

When selecting visualizations for a project, do not neglect effect visualizations. Recall that effect visualizations do not visualize architectural design decisions directly, but the results of applying some process to those design decisions—analysis, simulation, and so on. Often, so much focus is put onto model visualization that these effect visualizations get short shrift. Remember that analysis and simulation results are critical and must be interpreted correctly; this can be made substantially easier with the use of appropriate effect visualizations.

Used appropriately, visualizations can make working with, understanding, exchanging, and communicating about architectural design decisions much easier. To maximize this effect, select visualizations with high degrees of fidelity, consistency (both internal

and external), comprehensibility, dynamism, and so on. These decisions should be considered in the context of the target stakeholders and their own needs, skills, and prior experiences.

In this chapter and Chapter 6, we discussed how architectural design decisions can be captured in models, and how those models can be visualized. We have reiterated that semantically rich models are more valuable than semantically poor ones. The next chapter explains why. There, we discuss how architectural analysis leverages semantic information in models to help stakeholders discover important system properties early, before they propagate into later design phases where they are expensive to change.

Visualizations can be seen as the user interfaces to architectural models. In many ways, they fulfill the same role as a user interface in a software system. They define the look and feel of the underlying model to stakeholders. They can make it easier to work with the underlying concepts and semantics, but do not modify them. Because of this, even the best visualizations cannot rectify semantic problems in their target notations, but they can add substantial value to already-adequate notations, and have a tremendous effect on the practical usability of those notations.

For these reasons, it is important to consider visualizations—not just notations—up front in a software development effort. One key factor to keep in mind while selecting visualizations is the set of stakeholders that will be using them. Visualizations can take complex models and present them (or a subset of them) in a way that makes intuitive sense to stakeholders. The kinds of visualizations that are most useful for a project manager, for example, are not the same as those that are useful for a software engineer. While canonical visualizations are always available, remember that alternatives are often available as well.

The decision to develop a new visualization should be made with great care. As noted in the chapter, new visualizations can be very costly to develop and maintain. If possible, leverage extensibility mechanisms available in existing visualizations rather than developing one from scratch. Developing a new visualization can be a viable strategy in certain circumstances. Some notations, particularly esoteric or research-based notations, may have valuable semantics or analysis capabilities but lack good visualization support. When developing a family of systems in a particular domain (see Chapter 15), creating new domain-specific visualizations may also be valuable, and the cost is amortized over all projects that use the visualization.

The Business Case

7.7 REVIEW QUESTIONS

1. What is a visualization? What two key elements comprise a visualization?
2. What is the difference between a visualization and a modeling notation? What is a canonical visualization?
3. Identify and describe the two primary categories of visualizations.
4. What are hybrid visualizations? Identify a hybrid visualization and describe why it is a hybrid.
5. What is the relationship between visualizations, viewpoints, and views?
6. Enumerate and describe some criteria that can be used to evaluate visualizations.
7. When should you consider creating a new visualization? Enumerate and describe some strategies for creating an effective new visualization.
8. What does the work of Edward Tufte have to say about software architecture visualizations? What are his key insights?

9. Enumerate and describe some strategies for coordinating multiple visualizations.
10. What are effect visualizations? Where do they come from? How are they different from ordinary architecture visualizations?
11. Enumerate and describe some common problems that arise in architecture visualization.
12. What kinds of visualizations are associated with UML? How do these complement and differ from each other?
13. How do Rapide and LTSA utilize effect visualizations?
14. What kinds of visualizations are associated with xADL 2.0? What are the strengths and weaknesses of these visualizations?

7.8 EXERCISES

1. Identify a notation that is supported by two different visualizations (for example, a graphical and textual visualization). Model a system of your choosing, such as Lunar Lander, in both visualizations. Compare and contrast the experiences, and note especially what kinds of information were easy, hard, or impossible to capture in either.
2. Acquire and install a system that uses effect visualizations, such as Rapide, LTSA, or MTAT. Find a partner and have each of you model a small system using these tools. Trade systems and see how the effect visualizations can be used to help understand each other's models.
3. Choose one or more architectural visualizations not described here and evaluate it using the evaluation rubric presented in the chapter. What are their strengths and weaknesses?
4. Choose an architecture modeling notation and construct a simple novel visualization for it. For example, develop a simplified graphical editor that focuses only on a few kinds of elements, or use a translation technology such as XSLT to transform a complex textual depiction into something more readable.
5. Identify an unusual user interface feature (depiction, interaction, or both) that you have seen outside the context of architecture visualizations—perhaps in another application, a Web interface, or a video game. How might you apply this to architecture visualization? Would it be an improvement or a hindrance?
6. Choose an architecture visualization (or set of visualizations) not presented here. Does it follow the guidelines in this chapter for effective visualizations? Does it exhibit any of the common problems associated with visualizations? Develop a constructive critique of the visualization along these lines.

7.9 FURTHER READING

Most of the visualizations you will encounter are canonical visualizations, and so are documented in the same places as their associated tools or underlying notations. Such is the case for PowerPoint-like tools [(Microsoft Corporation 2007) (Microsoft Corporation 2007) (The Omni Group 2007)] and UML (Booch, Rumbaugh, and Jacobson 2005), as well as several others. It is also interesting to investigate related visualizations that are not canonical, such as XMI (Object Management Group

2002). Systems like Rapide (Luckham and Vera 1995) and LTSA (Magee and Kramer 2006) are interesting for their use of effect visualizations.

The work of Edward Tufte (Tufte 2001; Tufte 1990), cited in a sidebar in this chapter, is excellent reading for anyone who wants to present complex information in a clear, coherent way. Although he focuses mainly on the display of scientific, quantitative data, many of his ideas are easily translated to other domains and applications.

Analysis

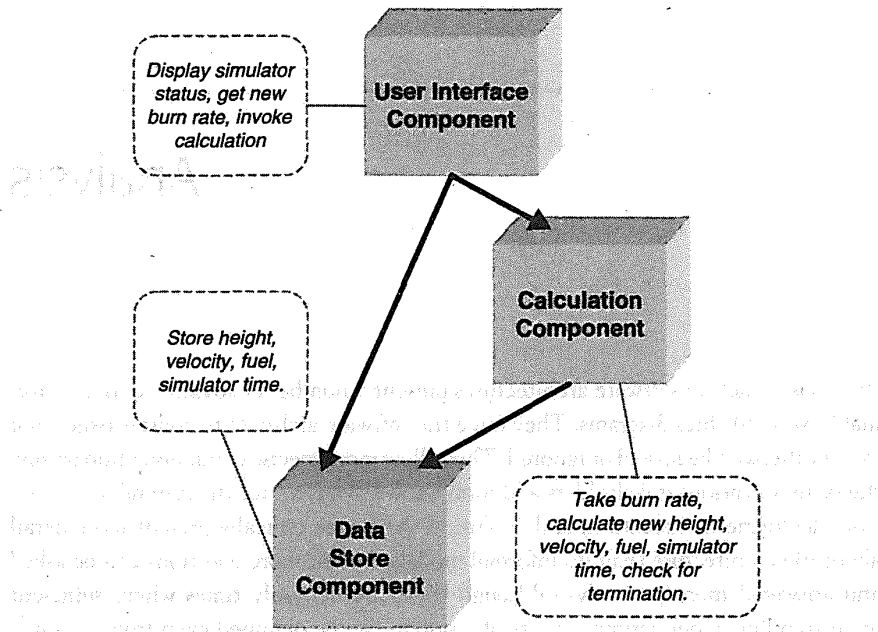
Rigorous models of software architectures present a number of advantages over informal boxes-and-lines diagrams. They force the software architect to address issues that might otherwise be missed or ignored. They allow more precise communication among the system's various stakeholders and form a solid blueprint for the system's construction, deployment, execution, and evolution. And they typically present more detail about the architecture than do informal models, so that more questions can be asked and answered more precisely—although there are certainly times where sufficient understanding about certain aspects of a system can be obtained even from informal models.

Definition. Architectural analysis is the activity of discovering important system properties using the system's architectural models.

Getting early, useful answers about relevant aspects of the system's architecture can help identify inappropriate or incorrect design decisions before they are propagated into the system, thus reducing the risk of system and project failures. It is important for a software architect, as well as other system stakeholders, to know which questions to ask about the architecture and why, how to ask them, and how best to ensure that they can be answered by extrapolating and interpreting the necessary information captured in the architecture's model.

All models will not be equally effective in helping to determine whether a given architecture satisfies a certain requirement. For example, consider the diagram representing the Lunar Lander architecture in Figure 8-1, initially introduced in Chapter 6. This diagram might help the architect get clarifications from the system's customers and vice versa; it may also be (informally) analyzed by a manager to ensure that the project's scope is appropriate. At the same time, such an early, informal model will not always be useful for communicating with, and within, the system development teams. The model, for example, does not help answer questions such as how exactly

Figure 8-1.
A diagram
informally
representing the
Lunar Lander
architecture using
Microsoft
PowerPoint.



the components (that is, the model's boxes) interact, where they are deployed with respect to each other, and what the nature of their interactions (that is, the model's lines) is.

On the other hand, a more formal architectural model of the given system may precisely define component interfaces, the conditions under which invoking a given interface is legal, a component's internal behavior, its legal external interactions, and so on. Such a model of the Lunar Lander architecture was given in Chapter 6; it is shown again in Figure 8-2. This model can be analyzed for a number of properties. For example, the model can help to ensure component composability into the system in the manner specified by the architectural configuration. After that analysis is successfully completed, the individual components may be assigned to developers for implementation. In some cases, individual component models instead may be used in further analysis to discover closely matching existing components to be grabbed off the shelf and reused in the new system. Yet another alternative would be to analyze the component model by an automated code generation tool whose output would be the component's implementation in a given programming language.

At the same time, such a rich model may not be the most effective means for answering questions about the given project's scope or about the resulting system's satisfaction of key requirements. Such questions may be more effectively answered by the system's

```

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

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

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

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

```

Figure 8-2.
A partial, formal
model of the
Lunar Lander
architecture
corresponding to
the diagram from
Figure 8-1. The
architecture is
modeled using the
Rapide ADL.

less technical stakeholders, such as managers and customers, through informal, perhaps manual analysis of less rigorous and detailed models.

It is also crucial to recognize that analyzing a software architecture does not have the same objectives, and is not dealing with the same issues, as analyzing software programs.

Architects will have to determine which problems identified in their architectures are critical, and which are not. Some problems uncovered by analysis may be acceptable if the system is still undergoing architectural design, for example, if certain parts of the architecture are still missing.

Another dimension to this issue is induced by off-the-shelf reuse of existing functionality. In naïve theory, software developers should strive to achieve perfect matches among their system's components: Each service provided by a given component will be needed by one or more components in the system, and each service required by a given component will be provided by another component in the system. In other words, there is neither any unneeded functionality provided in the system nor is there any needed functionality missing from the system. Systems developed in this manner are simpler and conceptually cleaner. However, this does not work in most large software systems, for which architecture-based development is geared. Most such systems involve the reuse of off-the-shelf functionality, and design and implementation of extensible components that will be usable in multiple systems. Therefore, certain mismatches among the components in a given architecture may be not only acceptable, but also desirable in this larger context.

The objective of this chapter is to present and relate different facets of architectural analysis. To this end, this chapter organizes the discussion around eight dimensions of concern relevant to architectural analysis:

1. The goals of analysis.
2. The scope of analysis.
3. The primary architectural concern being analyzed.
4. The level of formality of the associated architectural models.
5. The type of analysis.
6. The level of automation.
7. The system stakeholders to whom the results of analysis may be relevant.
8. The applicable analysis techniques.

The chapter expounds upon each of the seven dimensions, and discusses how they relate to and sometimes constrain each other.

**Outline of
Chapter 8**

8 Analysis	
8.1 Analysis Goals	8.2.5 Comparison of Two or More Architectures
8.1.1 Completeness	8.3 Architectural Concern Being Analyzed
8.1.2 Consistency	8.4 Level of Formality of Architectural Models
8.1.3 Compatibility	8.5 Type of Analysis
8.1.4 Correctness	8.6 Level of Automation
8.2 Scope of Analysis	8.7 System Stakeholders
8.2.1 Component- and Connector-Level Analysis	8.8 Analysis Techniques
8.2.2 Subsystem- and System-Level Analysis	8.8.1 Inspections and Reviews
8.2.3 Data Exchanged in the System or Subsystem	8.8.2 Model-Based Analysis
8.2.4 Architectures at Different Abstraction Levels	8.8.3 Simulation-Based Analysis
	8.9 End Matter
	8.10 Review Questions
	8.11 Exercises
	8.12 Further Reading

8.1 ANALYSIS GOALS

As with analysis of any software artifact, the analysis of architectural models can have varying goals. Those goals may include early estimation of system size, complexity, and cost; adherence of the architectural model to design guidelines and constraints; satisfaction of system requirements, both functional and non-functional (see Chapter 12); assessment of the implemented system's correctness with respect to its documented architecture; evaluation of opportunities for reusing existing functionality when implementing parts of the modeled system; and so forth. We categorize such architectural analysis goals into four categories, as discussed below. We refer to these as the four Cs of architectural analysis.

8.1.1 Completeness

Completeness is both an external and an internal analysis goal. It is *external* with respect to system requirements. The main goal of assessing an architecture's completeness in this context is to establish whether it adequately captures all of a system's key functional and non-functional requirements. Analyzing an architectural model for external completeness is nontrivial. Software systems for which an architecture-centric development perspective is most useful are often large, complex, long lived, and dynamic. In such settings, both the captured requirements and the modeled architecture may be very large and complex, and may be captured using a multitude of notations of various levels of rigor and formality. Furthermore, both likely will be specified incrementally and will change over time, so that the system's engineers need to carefully select points at which external completeness of the architecture can and should be assessed meaningfully.

Analyzing an architecture for *internal* completeness establishes whether all of the system's elements have been fully captured, both with respect to the modeling notation and with respect to the system undergoing architectural design.

Establishing the completeness of the model with respect to the modeling notation (recall Chapter 6) ensures that the model includes all the information demanded by the notation's syntactic and semantic rules. For example, the architectural model depicted in Figure 8-2 is captured in the Rapide architecture description language and thus must adhere to Rapide's syntax and semantics. Given this choice of modeling notation, the component instances in the `architecture` portion of the model must be attached to one another (see the `connect` statement Figure 8-2) according to the rules of Rapide: A component's `out` action must be connected to another component's `in` action; furthermore, connectors are not declared explicitly in Rapide.

Note, however, that fulfilling the modeling requirements of a language such as Rapide does not ensure that the architecture is, in fact, captured completely. Rapide is agnostic as to whether the architect accidentally omitted a major system component or whether a specified component's interface is missing critical services. Establishing the completeness of the architectural model with respect to the system being designed requires checking—often manually, as will be further discussed later in the chapter—whether there are missing components and connectors in the architecture; whether the specified components' and connectors' interfaces and protocols of interaction are fully specified; whether all of the dependencies and interaction paths are captured by the system's architectural configuration; and so on.

In principle, internal completeness is easier to assess than external completeness, and is amenable to automation. A number of software architecture analysis techniques have focused on this very analysis category, as will be elaborated later in this chapter.

8.1.2 Consistency

Consistency is an internal property of an architectural model, which is intended to ensure that different elements of that model do not contradict one another. The need for consistency derives from the fact that software systems, and thus their architectural models, are complex and multifaceted. As a result, even if no architectural design decisions are invalidated during the architectural design process, capturing the details of those decisions during architecture modeling may result in many inadvertently introduced inconsistencies. Examples of inconsistencies in a model include the following.

- Name inconsistencies.
- Interface inconsistencies.
- Behavioral inconsistencies.
- Interaction inconsistencies.
- Refinement inconsistencies.

Name Inconsistency

Name inconsistencies can occur at the level of components and connectors or at the level of their constituent elements, such as the names of the services exported by a component. The experience from using programming languages may suggest that name inconsistencies are trivial and easy to catch, but this is not always the case, especially at the architectural level. First, multiple system elements and/or services may have similar names. For example, a large system may have two or more similarly named GUI-rendering components; likewise, a

large GUI component may provide two or more similarly named widget-rendering services. Determining that the wrong component or service is accessed may be difficult.

The second problem with possible name inconsistencies concerns the richness of design choices available to the software architect. In a programming language such as Java, attempting to access a nonexistent class or method will most often result in compile-time errors which the engineer must correct before moving on. This is a by-product of relatively tight coupling of different program elements: early binding, type checking, and synchronous point-to-point procedure call semantics. On the other hand, a software architect may rely on highly decoupled architectures characterized by publish-subscribe or asynchronous event broadcast component interactions. Furthermore, the architecture may be highly adaptable and dynamic, such that tracking all name mismatches at a given time may be meaningless: A component or service referred to in the architecture initially may be unavailable but will be added to the system by the time it is actually needed.

Interface Inconsistency

Interface inconsistencies encompass the issues present in name inconsistencies. Specifically, all name inconsistencies are also interface inconsistencies, but not the other way around. A component's required service may have the same name as another component's provided service, but their parameter lists, as well as parameter and return types, may differ.

For illustration, consider the following. The interface of a required service in a simple QueueClient component, specified using an architecture description language, may be as follows:

```
ReqInt: getSubQ(Natural first, Natural last, Boolean remove)
        returns FIFOQueue;
```

This interface is intended to access a service that returns the subset of a FIFOQueue between the specified **first** and **last** indices. The original queue may remain intact, or the specified subqueue may be extracted from it, depending on the value of the **remove** parameter.

On the other hand, the QueueServer component providing the service may export two **getSubQ** interfaces as follows:

```
ProvInt1: getSubQ(Index first, Index last)
           returns FIFOQueue;

ProvInt2: getSubQ(Natural first, Natural last, Boolean remove)
           returns Queue;
```

All three interfaces have identical names, so it can be immediately observed that there is no name inconsistency. However, the three interfaces' parameter lists and return types are not identical. Specifically:

1. The types of the **first** and **last** parameters in the required interface **ReqInt** and the provided interface **ProvInt1** are different.
2. The required interface **ReqInt** introduces a Boolean **remove** parameter, which does not exist in **ProvInt1**.

3. Finally, the return types of the provided interface `ProvInt2` and required interface `ReqInt` are different.

Whether these differences result in actual interface inconsistencies will depend on several factors. If the `QueueClient` and `QueueServer` were objects implemented in a programming language such as Java, and their respective provided and required interfaces denoted method invocations, the system might not even compile. However, as has been demonstrated repeatedly throughout this book, software architecture provides a much richer set of choices to an engineer. Consider all three differences between the provided and required interfaces, and their impact on potential interface inconsistency.

1. If the data type `Natural` is defined to be a subtype of `Index`, then requesting the `getSubQ` service will not cause a type mismatch between `ReqInt` and `ProvInt1`. Instead, a simple type cast will occur and the request will be serviced normally.
2. If the connector between `QueueClient` and `QueueServer` components is a direct procedure call, then an interface inconsistency will occur between `ReqInt` and `ProvInt1` because of the additional parameter in the required interface. No such inconsistencies will occur between `ReqInt` and `ProvInt2`, which have identical parameter lists. On the other hand, if the two components interact via an implicit invocation mechanism, such as an event connector, the connector may simply package the request such that the `QueueServer` component can still service it via the `ProvInt1` interface, by ignoring the `remove` parameter: `QueueServer` will access only the two parameters it needs, and will not need to be concerned with any additional parameters that may have been delivered via the event. In this case, the default implementation of the `getSubQ` service will simply be executed. For example, the default implementation may be that the specified subqueue is extracted from the queue. It is, of course, possible that this will not match the `QueueClient`'s expectation of `getSubQ`'s behavior, if the value of `remove` is set to false. This issue is further discussed below in the context of behavioral inconsistency.
3. Unless this system's architectural description explicitly specifies that `Queue` is a subtype of `FIFOQueue`, or that they are identical types, `ProvInt2` will not be able to service the `ReqInt` request.

Determining whether there exists an interface inconsistency in a system thus depends on several factors. At the same time, this is an architecture analysis task that can be accomplished relatively easily and should be readily automatable.

Behavioral Inconsistency

Behavioral inconsistencies occur between components that request and provide services whose names and interfaces match, but whose behaviors do not. As a very simple example, consider the service exported by the following interface:

```
subtract(Integer x, Integer y) returns Integer;
```

This service takes two integers as its input and returns their difference. It is natural to assume that the subtraction is arithmetic, and many math libraries will support this as well

as much more complex operations. However, the component providing this service need not calculate the two numbers' arithmetic difference, but instead may provide a calendar subtraction operation. Thus, for example, the requesting component may expect that the difference between 427 and 27 will be 400, while the component providing the service may treat it as the subtraction of 27 days from April 27, and return 331 (March 31).

An architectural model may provide a behavioral specification for the given system's components and their services. The behavioral specifications may take different forms, as illustrated in Chapter 6. For example, each required and provided interface can be accompanied with preconditions, which must hold true before the functionality exported via the interface is accessed, and postconditions, which must hold true after the functionality is exercised.

For example, let us assume that the above discussed `QueueClient` component requires a `front` operation, whose purpose is to return the first element of the queue. Furthermore, let us assume that `QueueServer` provides this operation, and that the two corresponding interfaces match. `QueueClient`'s required service behavior is specified as follows:

```
precondition q.size ≥ 0;
postcondition ~q.size = q.size;
```

where $\sim$ denotes the value of the variable `q` after the operation has been executed. Therefore, the `QueueClient` component assumes that the queue may be empty and that the `front` operation will not alter the queue.

Let us assume that the `QueueServer` component's provided `front` operation has the following pre- and postconditions:

```
precondition q.size ≥ 1;
postcondition ~q.size = q.size - 1;
```

The precondition asserts that the queue will be nonempty, while the postcondition specifies that the operation will alter the size of the queue (that is, that the front element will be dequeued).

A casual analysis of the two specifications indicates that the behavior of the `front` operation required by `QueueClient` does not match that provided by `QueueServer`. The postconditions clearly are different; moreover, the provided operation assumes that `front` will not be invoked before the existence of at least one element in the queue is ascertained.

A component's behavior may be specified in several different ways. Chapter 6 discussed several examples, including state-transition diagrams, communicating sequential processes, and partially ordered events sets. The exact manner in which the behavioral consistency between services is ensured will vary across these notations, and hence is outside the scope of this text. On the other hand, the overall analysis process will follow the general pattern outlined above, regardless of the behavior modeling notation.

Interaction Inconsistency

Interaction inconsistencies can occur even if two components' respective provided and required operations have consistent names, interfaces, and behaviors. An interaction inconsistency occurs when a component's provided operations are accessed in a manner that violates certain interaction constraints, such as the order in which the component's operations are to be accessed. Such constraints comprise the component's interaction *protocol*.

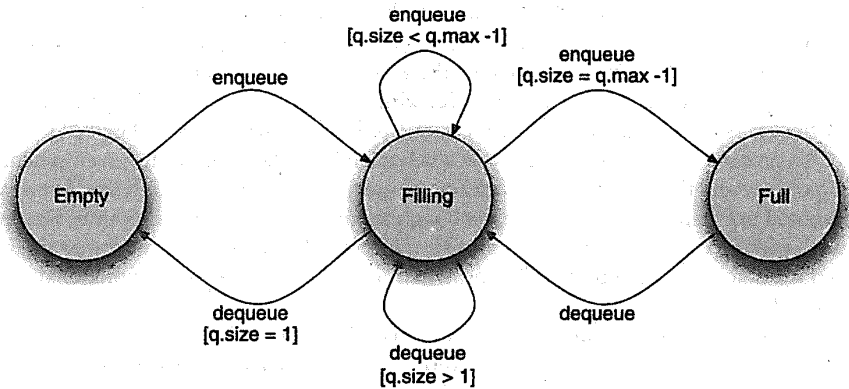


Figure 8-3. Interaction protocol for the QueueServer component. Transitions corresponding to operations such as *front* and *is.empty*, typically provided by a queue component, have been elided for clarity. Transition guards are enclosed within brackets. The assumption is that the queue can contain at least two elements.

A component's interaction protocol can be specified using different notations. Frequently, it is modeled via state-transition diagrams (Yellin and Strom 1994). Analyzing the system for interaction consistency in that case consists of ensuring that a given sequence of operation requests matches some sequence of legal state transitions specified by each component's protocol.

An example of such interaction protocol is provided in Figure 8-3. In the figure, the QueueServer component requires that at least one element always be enqueued before an attempt to dequeue an element can be made; furthermore, it assumes that no attempts to enqueue elements onto a full queue will be made. A QueueClient component that does not adhere to these constraints—that is, whose sequence of invocations cannot be executed by the state machine from Figure 8-3—will cause an interaction inconsistency with the QueueServer.

Refinement Inconsistency

Refinement inconsistencies stem from the fact that a system's architecture is frequently captured at multiple levels of abstraction. For example, a very high-level model of the architecture may only represent the major subsystems and their dependencies, while a lower-level model may elaborate on many details of those subsystems and dependencies. As an illustrative example, Figure 8-4 shows the high-level architecture of the Linux operating system, provided by Ivan Bowman and colleagues. (Bowman, Holt, and Brewster 1999), and its Process Scheduler subsystem modeled as a composite connector, provided by Mehta et al. (Mehta, Medvidović, and Phadke 2000). Analyzing Linux's architecture for consistency would require establishing the following three conditions:

1. The elements of the higher-level architectural model have been carried over to the lower-level model—that is, no existing architectural elements have been lost in the course of the refinement.

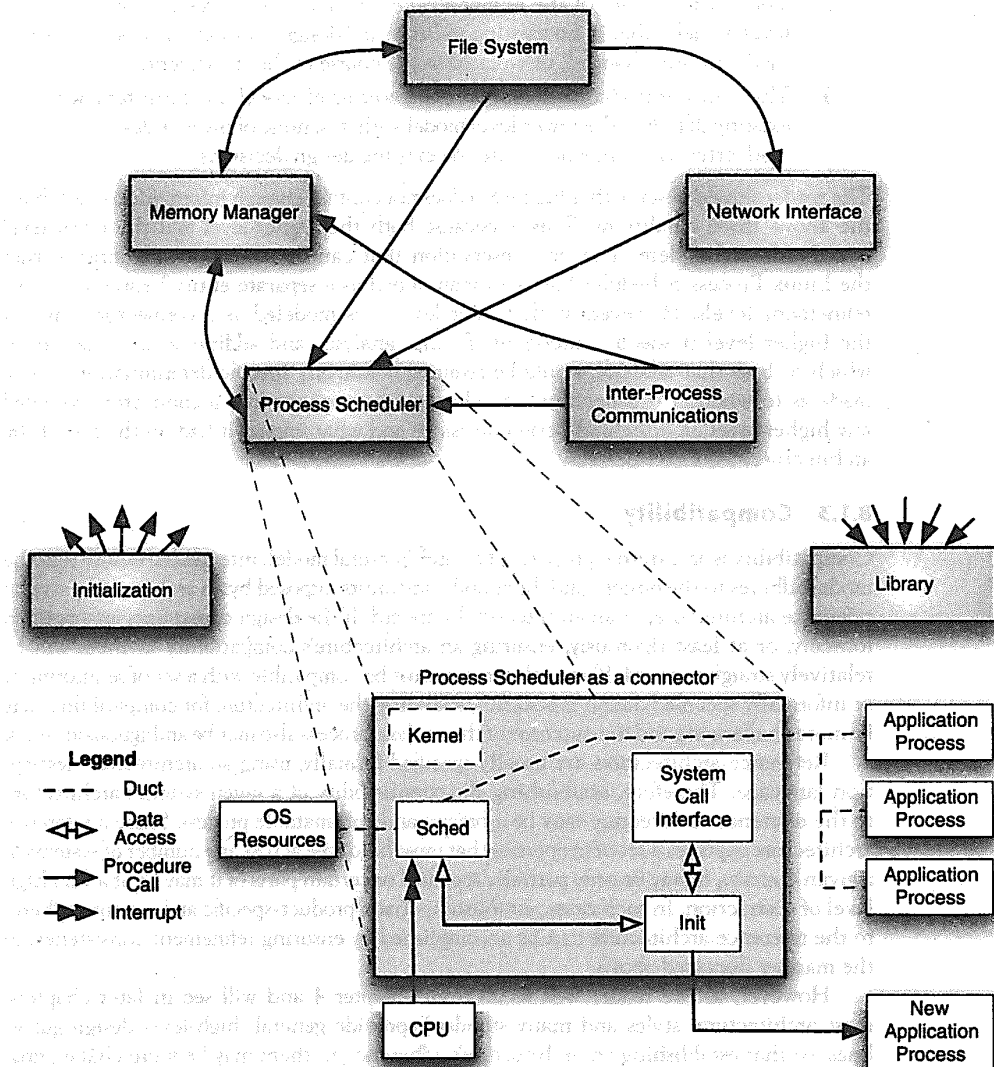


Figure 8-4. A very high-level model of the Linux operating system [adopted from Bowman et al. (Bowman, Holt, and Brewster 1999) © 1999 ACM, Inc. Reprinted by permission.] and a detailed model of the Linux Process Scheduler connector.

2. The key properties of the higher-level model have been preserved in the lower-level model—that is, no existing architectural design decisions have been omitted, inadvertently changed, or violated in the course of the refinement.
3. The newly introduced details in the lower-level model are consistent with the existing details of the lower-level model—that is, none of the new design decisions inadvertently change or violate the existing design decisions.

The reader should observe that Figure 8-4 does not contain enough information to establish the above three conditions. This is because both the higher-level and the lower-level models are incomplete. The one observation that can be made with certainty is that the Linux Process Scheduler has been maintained as a separate entity between the two refinement levels. However, at the lower level it is modeled as a connector, while at the higher level it was a component. Further analysis, and additional information on which to base that analysis, would be required before any specific determination can be made as to whether this decision—to change a component into a connector—violated any higher-level architectural design decisions and what impact it had on the rest of the architecture.

8.1.3 Compatibility

Compatibility is an external property of an architectural model, intended to ensure that the model adheres to the design guidelines and constraints imposed by an architectural style, a reference architecture, or an architectural standard. If the design constraints are captured formally, or at least rigorously, ensuring an architecture's compatibility to them will be relatively straightforward. If an architecture must be compatible with a set of semiformal or informally specified design guidelines, analyzing the architecture for compatibility may be more challenging and the outcome of the analysis process also may be ambiguous at times.

Reference architectures are usually specified formally, using an architecture description language. Therefore, establishing the compatibility of a given system's architecture to the reference architecture may be a precise and automatable process. Since a reference architecture captures a set of properties that must hold true across any number of systems in a given domain, it may be only partially specified or certain parts of it may be at a very high level of abstraction. In such cases, establishing that a product-specific architecture adheres to the reference architecture can be accomplished by ensuring refinement consistency, in the manner discussed above.

However, as the reader will recall from Chapter 4 and will see in later chapters, most architectural styles and many standards provide general, high-level design guidelines, so that establishing an architecture's adherence to them may be more challenging. The difficulty may arise from fuzziness on the part of the style definition or imprecision or incompleteness on the part of the architectural model. For example, Figure 8-5 depicts the Lunar Lander architecture according to the event-based style. This diagram was first shown in Chapter 4, with a slightly different component layout, as were the relatively similar diagrams for the Lunar Lander architectures in the C2 (Figure 4-23) and blackboard (Figure 4-16) styles. Determining the style to which this architecture adheres is a nontrivial task: The depicted configuration of components may, in fact, also adhere to C2's principles, such as substrate independence. Likewise, this particular visual layout of the architecture's topology may be misleading, for the Spacecraft component may in fact play the role of a blackboard in this system. In cases such as this, the architect may need to

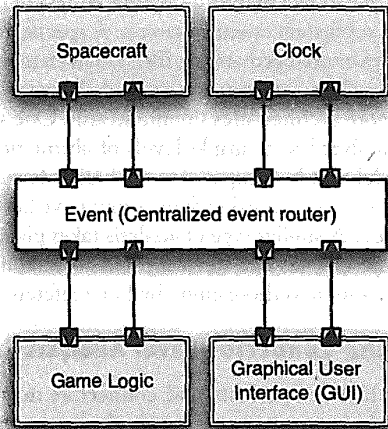


Figure 8-5.
A depiction of the
Lunar Lander
architecture
according to the
event-based style.

obtain additional information, rely on tacit knowledge, and use one or more architectural analysis techniques presented later in this chapter.

8.1.4 Correctness

Correctness is an external property of an architectural model. A system's architecture is said to be correct with respect to an external system specification if the architectural design decisions fully realize those specifications. Furthermore, the system's implementation is correct with respect to the system's architecture if the implementation fully captures and realizes all the principal design decisions comprising the architecture. Correctness is therefore *relative*: It is the result of architecture to some other artifact, where the artifact is either intended to elaborate and fulfill the architecture or the architecture is intended to elaborate and fulfill the artifact.

An interesting observation arises in the implementation of many large, modern software systems that reuse off-the-shelf functionality, or off-the-shelf architectural solutions such as reference architectures. In such cases, it is very likely that, through the off-the-shelf components, the implemented system will include structural elements or functionality, as well as non-functional properties, that are not specified in the requirements and/or not modeled in the architecture. With a 1970's notion of *refinement-based correctness*, such systems would not be considered to be correct with respect to the architecture. However, with the notion of correctness built upon fulfillment, as described above, such systems are not only correct, but they are likely efficiently created and form a suitable basis for efficiently fulfilling new system requirements.

8.2 SCOPE OF ANALYSIS

A software system's architecture can be analyzed from different perspectives and at different levels. Architects may be interested in assessing the properties of individual components or connectors, or even their constituent elements, such as interfaces or ports. More frequently,

architects may be interested in the properties exhibited by compositions of components and connectors in a given subsystem or entire system. A specific focus of (sub)system-level analysis may be on the data exchanged among the system elements.

In addition to assessing the properties of a single architecture, architects may at times need to consider two or more architectures simultaneously. One such case is when a given system's architecture is analyzed at multiple levels of abstraction. For example, a more detailed architectural model may be compared to the higher-level model from which it has been derived, to ensure that no existing design decisions have been violated or unintended design decisions introduced. A similar type of analysis takes place when two architectural models at similar levels of abstraction are compared to establish design similarities or conformance to constraints such as those embodied in a reference architecture.

8.2.1 Component- and Connector-Level Analysis

In a large system, individual components and connectors may not always be as interesting to an architect as their compositions, but both components and connectors need to provide specific services at specific quality levels. In the case of components, this typically means application-specific functionality; in the case of connectors, this means application-independent interaction services.

The simplest type of component- and connector-level analysis ensures that the given component or connector provides the services expected of it. For example, a component's or connector's interface can be inspected to make sure that no expected services are missing. As an illustration, Figure 8-6 shows Lunar Lander's Data Store component modeled in xADLite. It is trivial to analyze this component, either manually or automatically, to establish that it provides both `getValues` and `storeValues` services; furthermore, if a system stakeholder expects or requires the Data Store component to provide further services, it can easily be ascertained that no such services currently exist. Even if the component were much larger and provided many more services, this task would not be significantly harder.

Of course, "checking off" the services a component or connector provides does not ensure that those services are modeled correctly. The described analysis can be thought of as equivalent to establishing only name consistency. A component or connector may provide services with the expected names, but with incorrect interfaces. For example, `getValues` in the above example may be modeled to expect the values in a wrong format, such as untyped versus typed or string versus integer. Therefore, it is not sufficient to establish that

Figure 8-6.
Lunar
Lander's Data
Store component
modeled
in xADLite. This
model is extracted
from that provided
in Chapter 6,
in Figure 6-27.

```
component{
  id = "datastore";
  description = "Data Store";
  interface{
    id = "datastore.getValues";
    description = "Data Store Get Values Interface";
    direction = "in";
  }
  interface{
    id = "datastore.storeValues";
    description = "Data Store Store Values Interface";
    direction = "in";
  }
}
```

```

connector Pipe =
  role Writer = write → Writer □
               close → √
  role Reader =
    let ExitOnly = close → √
    in let DoRead = (read → Reader □
                    read-eof → ExitOnly)
    in DoRead □ ExitOnly
  glue = let ReadOnly = Reader.read → ReadOnly □
        Reader.read-eof → Reader.close → √ □
        Reader.close → √
    in let WriteOnly = Writer.write → WriteOnly □
      Writer.close → √
    in Writer.write → glue □
      Reader.read → glue □
      Writer.close → ReadOnly □
      Reader.close → WriteOnly

```

Figure 8-7.
Pipe connector
modeled in
Wright.

the component or connector provides appropriately named services; the complete interface of the component or connector has to be analyzed. The information provided in Figure 8-6 will thus have to be supplemented with additional details, possibly from other models.

Taking this argument a step further, it is not sufficient to ensure that the component's or connector's services are exported via an appropriate interface. The semantics of those services as modeled (and, eventually, as implemented) may be different from the desired semantics. Thus, for example, the `getValues` service from Figure 8-6 may not be modeled such that it accesses the Data Store to obtain the needed values, but instead may request those values from a system user. Since the intended usage of the component, implied in its name, is to access a repository, this implementation of `getValues`, while legitimate in principle, would be wrong for this context.

Similarly, a connector may provide interaction services with semantics that are different from the expected semantics. For example, a connector may be expected to support asynchronous invocation semantics, but is actually modeled for synchronous invocation. Establishing this type of semantic consistency is not trivial. Consider as an illustration a model of a *Pipe* connector in the Wright ADL discussed in Chapter 6; the model is depicted in Figure 8-7. The *Pipe* connector plays two roles; that is, it provides two services: **Reader** and **Writer**. Their interplay, that is, the connector's overall behavior, is captured by the connector's `glue`. Even though a pipe is a relatively simple connector, establishing manually that it adheres to its expected behavior is significantly more challenging than with name or interface conformance. For example, is the intended semantics to allow unimpeded writing to the pipe, as in the current specification of the **Writer** role, or does the pipe have a bounded buffer so that writing a certain amount of data would require that at least some of that data be read before additional data can be written? Assessing more complex connectors for these types interaction properties will be much more difficult and may be especially error-prone.

8.2.2 Subsystem- and System-Level Analysis

Even if individual components and connectors have desired properties, no guarantees can be made that their compositions in a given system will behave as expected, or even be legal. The interplay among complex components can itself be very complex. Architects may assess the properties of their compositions at the level of the entire system or incrementally, by focusing on particular subsystems.

The most manageable increment is pair-wise conformance, where only two interacting components are considered at a time, and name, interface, behavior, and interaction conformance are established as discussed above. The next step up is to take a set of components possibly interacting through a single connector, such as those shown in Figure 8-5. Most analysis techniques can easily support both these cases. Ensuring desirable properties at the level of large subsystems and entire systems can be quite challenging, and many analysis techniques have tried to get a handle on this problem.

In certain scenarios it may be obvious that compositions of two or more components that respectively possess properties α , β , γ , and so forth will have some combination of those properties. For example, combining a data encryption component—intended to provide communication security—with a data compression component—intended to provide communication efficiency—can be expected to provide both security and efficiency.

Much more frequent in practice is the situation where the interplay among the components will result in their interference and either enhancement or diminishment of each other's properties. This will be acceptable, and even desirable, in certain cases. For example, a component that provides a critical service very efficiently may be vulnerable to malicious attacks. The system architects may decide that sacrificing some of the efficiency is acceptable in order to enhance the system's security, and may compose this component with one or more components that provide services such as encryption, authentication, or authorization. Likewise, in many real-time systems, a component that computes its results or provides data more quickly and/or frequently than the system requires will be composed with a (simple) component that introduces the necessary delays. Such compositions will be synergistic: The system will ultimately be greater than the sum of its parts.

Such interference among system components will not be desirable in all situations. More importantly, such interference will often not be as obvious as in the above scenarios and will result in unintended composite properties. Examples abound. One such well-known example from software engineering literature involved integrating two components that both assumed that they owned the system's main thread of control, ultimately resulting in an unusable system (Garlan, Allen, and Ockerbloom 1995). Another similar example involved integrating concurrent components implemented in two different programming languages. The resulting system's performance was unacceptable, and a long and painstaking analysis of the system uncovered that the individual components' threading models were incompatible (Maybee, Heimbinger, and Osterweil 1996). The reader can easily envision many other such scenarios, such as when a system comprises interacting memory-efficient components (which use computationally intensive data compression algorithms) and CPU-efficient components (which use techniques such as data caching and prefetching) or, further, when a component introducing a fault-tolerance service (for example, via continuous state replication) is introduced into such a system.

Architectural Analysis and Honey- Baked Ham

Scenarios such as those discussed in this section are a direct reason why many modern software systems suffer from poor performance, security, and scalability; unnecessarily large size; high complexity; limited adaptability; and so on. For example, a concurrent system in which any component or subsystem considered in isolation is deadlock free, livelock free, and starvation-free may in fact suffer from deadlock, livelock, or starvation because of the unforeseen interplay of multiple components and/or subsystems.

This is what Dewayne Perry has colloquially referred to as the *honey-baked ham syndrome*: Honey is fat free, while ham is sugar free; honey-baked ham, therefore, must be both fat free and sugar free. Clearly, this is not true, and drawing such conclusions based on the evidence available from a system's individual components may be wrong and perilous. The honey-baked ham syndrome is a primary reason why subsystem- and system-level architectural analysis is critically important.

8.2.3 Data Exchanged in the System or Subsystem

In many large, distributed software systems large amounts of data are processed, exchanged, and stored. Examples of data-intensive systems are numerous and appear in such wide ranging domains as scientific computing, many Web-based applications, e-commerce, and multimedia. In such systems, in addition to the properties of the individual structural architectural elements—components and connectors—and the entire architectural configuration, it is important to ensure that the system's data is properly modeled, implemented, and exchanged among the structural elements. This involves assessing the data elements, including the following.

- The structure of the data, such as typed versus untyped or discrete versus streamed.
- The flow of the data through the system, such as point-to-point versus broadcast.
- The properties of data exchange, such as consistency, security, and latency.

As a simple example, consider a system consisting of a data-producer component and two data-consumer components, whose architectural configuration is depicted in Figure 8-8. The figure also shows the respective frequencies at which they are able to exchange data. The Producer component sends one megabit per second, and Consumer 1 is able to receive and process that data in a timely manner. In fact, Consumer 1 may wait idly up to 50 percent of the time to receive additional data from the Producer. However, Consumer 2 is able to receive and process the data at a rate that is only one-half of the production rate. This means that Consumer 2 may lose up to one-half of the produced data.

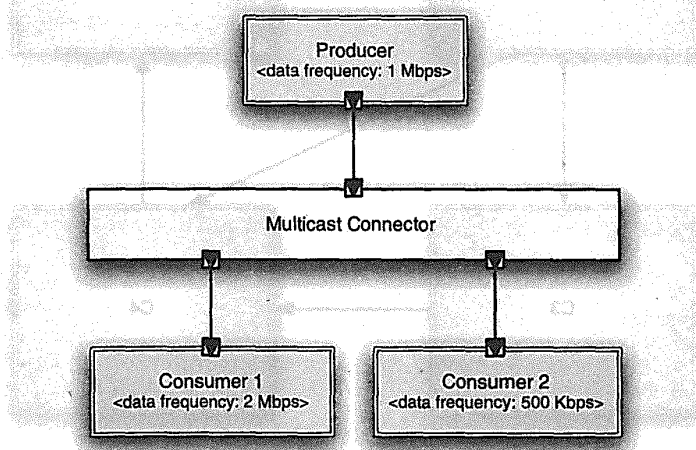


Figure 8-8.
A simple system
with one data
producer and two
consumers, with
the frequencies at
which they are
able to send and
receive data,
respectively.