Documenting Software Architecture: Documenting Interfaces

Felix Bachmann
Len Bass
Paul Clements
David Garlan
James Ivers
Reed Little
Robert Nord
Judith Stafford

June 2002

TECHNICAL NOTE
CMU/SEI-2002-TN-015

Carnegie Mellon
Software Engineering Institute

Pittsburgh, PA 15213-3890

Documenting Software Architecture: Documenting Interfaces

CMU/SEI-2002-TN-015

Felix Bachmann
Len Bass
Paul Clements
David Garlan
James Ivers
Reed Little
Robert Nord
Judith Stafford

June 2002

Architecture Tradeoff Analysis Initiative

Table of Contents

List of Figures

Abstract

This is the fourth in a series of Software Engineering Institute reports on documenting software architectures. This report details guidance for documenting the interfaces to software elements. It prescribes a standard organization (template) for recording semantic as well as syntactic information about an interface. Stakeholders of interface documentation are enumerated, available notations for specifying interfaces are described, and three examples are provided.

1 Introduction

This report represents another milestone of a work in progress. That work is a comprehensive handbook on how to produce high-quality documentation for software architectures. The handbook, titled *Documenting Software Architectures: Views and Beyond*, will be published in August 2002 by Addison Wesley Longman Inc. as part of the SEI Series on Software Engineering.

The book is intended to address a lack of language-independent guidance about how to capture an architecture in a written form that can provide a unified design vision to all the stakeholders on a development project. The book is aimed at the community of practicing architects, apprentice architects, and developers who receive architectural documentation.

Three previous reports laid out our approach and organization for the complete book and provided self-contained previews of individual chapters. The first provided guidance for one of the most commonly used architectural views: the layer diagram [Bachmann 00]. The second laid out a structure for a comprehensive architecture documentation package [Bachmann 01]. The third prescribed documentation approaches for describing the behavior of software [Bachmann 02].[1] The material in this document assumes familiarity with the language and concepts introduced in these earlier reports.

This technical note describes ways to document an important, but often overlooked, aspect of software architecture: the documentation of software interfaces.

1. Since these reports are snapshots of work in progress, the book may reflect and incorporate various changes in the details, but not in philosophy.

2 Overview

Early treatments of architecture and architecture-description languages devoted loving attention to the elements of software systems and their interactions but tended to overlook the interfaces to those elements. It was as though interfaces were not part of the architecture. Clearly, however, interfaces are supremely architectural, for one cannot perform analyses or system building without them. Therefore, a critical part of documenting a view includes documenting the interfaces of the elements shown in that view.

What is an *interface*? Various communities use various definitions, but we use the following one:

> *An interface is a boundary across which two independent entities meet and interact or communicate with each other.*

The characteristics of an interface depend on the view type of its element. If the element is a component, the interface represents a specific point of its potential interaction with its environment. If the element is a module, the interface is a definition of services. There is a relation between these two kinds of interfaces, just as there is a relation between components and modules.

By the element's environment, we mean the set of other entities with which it interacts. We call those other entities *actors*:

> *An element's actors are the other elements, users, or systems with which it interacts.*

In general, an actor is an abstraction for external entities that interact with the system. Here, we focus on elements and expand the definition of interaction to include anything one element does that can impact the processing of another element. This interaction is part of the element's interface. Interactions can take a variety of forms. Most involve the transfer of control and/or data. Some are supported by standard programming-language constructs, such as local or remote procedure calls (RPCs), data streams, shared memory, and message passing.

These constructs, which provide points of direct interaction with an element, are called *resources*. Other interactions are indirect. For example, the fact that using resource `X` on element `A` leaves element `B` in a particular state is something that other elements using the resource may need to know if it affects their processing, even though they never interact with

A directly. That fact about A is a part of the interface between A and the other elements in A's environment.

An interaction extends beyond merely what happens. For example, if element X calls element Y, the amount of time that Y takes before returning control to X is part of Y's interface to X because it affects X's processing.

Let's establish some principles about interfaces:

- *All elements have interfaces.* All elements interact with their environment.
- *An element's interface contains view-specific information.* Because an element can occur in more than one view, aspects of its interface can be documented in each view, using the vocabulary of that view. For instance, an interface to a module in a *uses* view might describe which methods are provided, but an interface to the same module in a *work-assignment* view would not include this information. In fact, some views may have little interface information to document. (Whether an architect chooses to document an element's interface separately in different views or in a single treatment is a packaging issue. An interface that transcends views can be documented in the package of documentation that applies to more than one view.)
- *Interfaces are two way.* When considering interfaces, most software engineers first think of a summary of what an element provides. What methods does the element make available? What events does it process? But an element also interacts with its environment by making use of resources or assuming that its environment behaves in a certain way. If these resources are missing or if the environment doesn't behave as expected, the element can't function correctly. So an interface is more than what is *provided* by an element; an interface also includes what is *required* by an element.

  The *requires* part of an element's interface typically comes in two varieties:

  - resources on which an element builds. This kind of resource is something that is used in the implementation of the element, such as class libraries or toolkits, but is usually not information that other elements use in interacting with the element. This type of resource requirement is typically documented by naming the library, version, and platform of the resource. A build will generally and quickly uncover any unsatisfied interface requirements of this kind.
  - assumptions that the element makes of other elements with which it interacts. For example, an element could assume the presence of a database using specific schema over which it can make SQL (structured query language) queries. Or, an element may require its actors to call an `init()` method before it allows queries. This type of information is critical to document—after all, the system won't work if the requirement is not met—and if not satisfied, it is hard to uncover.

When we say that an interface includes what is required, we're focusing on what interactions an element requires from its environment in order to complete an interaction that it provides.

- *An element can have multiple interfaces.* Each interface contains a separate collection of resources that either has a related logical purpose, or represents a role that the element could fill and that serves a different class of elements. Multiple interfaces provide a separation of concerns, which has obvious benefits. A user of the element, for example, might require only a subset of the functionality provided by the element. If the element has multiple interfaces, perhaps the developer's requirements line up nicely with one of the interfaces, meaning that the developer would have to learn only the interface that mattered to him or her rather than the complete set of resources provided by the element. Conversely, the provider of an element might want to grant users different access rights, such as read or write, to prevent resource contention or to implement a security policy. Multiple interfaces support different levels of access and support evolution in open-market situations. If you put an element in the commercial market and the element's interface changes, you can't recall and fix everything that uses the old version. So you can support evolution by keeping the old one but adding the new interface to it.
- *An element can interact with more than one actor through the same interface.* Document any limits on the number of actors that can interact with an element via a particular interface. For example, Web servers often restrict the number of simultaneously open HTTP (hypertext transfer protocol) connections.
- *Sometimes, it's useful to have interface types, as well as interface instances.* Like all types, an interface type is a template for a set of instances. Many times, all the interfaces you are designing will include a standard set of resources, such as an initialization program; a set of standard exception conditions, such as failing to have called the initialization program; a standard way to handle exceptions, such as invoking a named error handler; or a standard statement of semantics, such as persistence of stored information. It is convenient to write these standard interface "parts" as an interface type. Sometimes, an element has multiple interfaces that are identical: a component that merges two input streams might be designed with two separate but identical interfaces. It is convenient to write these identical interfaces as an interface type that can be documented in the architecture's cross-view documentation.

3 Terminology: *Signature, API,* and *Interface*

Three terms people use when discussing element interactions are *signature*, *API*, and *interface*. Often, they use the terms interchangeably, with unfortunate consequences for their projects. We have already defined an interface to be a boundary across which two independent entities meet or communicate with each other, and we have seen that documenting an interface consists of naming and identifying it, documenting syntactic information, and documenting semantic information.

A signature deals with the syntactic part of documenting an interface. When an interface's resources are invokable procedures, each comes with a signature that names the procedure and defines its parameters. Parameters are defined by giving their order, data type, and, sometimes, whether their value is changed by the procedure. A procedure's signature is the information that you would find about it, for instance, in the element's C or C++ header file.

An API, or application programming interface, is a vaguely defined term that people use in various ways to convey interface information about an element. Sometimes, people assemble a collection of signatures and call that an element's API. Other times, people add statements about programs' effects or behavior and call that an API. An API for an element is usually written to serve developers who use that element.

Signatures and APIs are useful but are only part of the story. Signatures can be used, for example, to enable automatic build checking, which is accomplished by matching the signatures of different elements' expectations of an interface, often simply by linking different units of code. Signature matching will guarantee that a system will compile and/or link successfully. But it guarantees nothing about whether the system will operate successfully, which is, after all, the ultimate goal.

For a simple example, consider two elements: one provides a `read()` method, and the other wants to use a `read()` method. Let's assume that the signatures match as well. So a simple automated check would determine that the elements are syntactically compatible. But suppose that the `read()` method is implemented such that it removes data from its stream as `read()` is executed. The user, on the other hand, assumes that `read()` is free of side effects and hence that it can read and reread the same data. This semantic mismatch here will lead to errors and illustrates why interfaces need to be specified beyond signatures.

As we will see in Section 6, a full-fledged interface is written for a variety of stakeholders, includes both *requires* and *provides* information, and specifies the full range of effects of each resource, including quality attributes. Signatures and low-end APIs are simply not enough to let an element be put to work with confidence in a system. Any project that adopts them as a shortcut will pay the price either when the elements are integrated or after the system has been delivered to the customer.

An analogy can be found in aviation. Every year in the United States, the Federal Aviation Administration and the National Transportation Safety Board spend millions of dollars tracking down counterfeit, low-quality aircraft parts. Jet engines, for example, are attached to aircraft by special bolts that have been engineered to have the right strength, durability, flexibility, and thermal properties. The next time you board a jet aircraft, imagine that the mechanic who reattached the jet engines after their last overhaul used whichever bolts were long enough and thick enough and happened to be lying around the parts bin. That's the mechanical engineering version of signature matching.

4 Interface Specification

An interface is documented with an *interface specification*:

> *An interface specification is a statement of what an architect chooses to make known about an element in order for other entities to interact or communicate with it.*

Although an interface comprises every interaction an element has with its environment, what we choose to disclose about an interface— that is, what we document in an interface specification—is more limited. Writing down every aspect of every possible interaction is not practical and almost never desirable. Rather, the architect should expose only what users of an element *need* to know in order to interact with it. Put another way, the architect chooses what information is permissible and appropriate for people to assume about the element and unlikely to change.

The interface specification documents what other developers need to know about an element in order to use it in combination with other elements and provides a statement of other visible properties. Note that a developer might observe element properties that are an artifact of how the element is implemented but that are not in the interface specification. Because these properties are not in the interface specification, they are subject to change and used by developers at the developers' own risk.

Documenting an interface is a matter of striking a balance between disclosing too little information and disclosing too much. Disclosing too little information will prevent developers from successfully interacting with the element. Disclosing too much will make future changes to the system more difficult and widespread, and make the interface complicated for people to understand.

Also recognize that different people need to know different kinds of information about the element. The architect may have to provide separate sections in the interface document or multiple interface documents to accommodate different stakeholders of the element.

The following guidelines are intended as rules of thumb for what to include in an interface specification (Section 5 suggests an organization for this information):

- Focus on how elements interact with their environments, not on how elements are implemented. Restrict the documentation to phenomena that are externally visible.

- Expose only what the actors in an element's environment need to know. Including a piece of information in the documentation is an implicit promise to the element's stakeholders that the information is reliable and stable. Once information is exposed, other elements may rely on it, and changes will have a more widespread effect.
- If you don't want people to rely on a piece of information, don't include it in the interface documentation. Make it clear that information that "leaks" through an interface but is not included in its interface documentation can be used only at the peril of the actors that exploit it and of the system as a whole.
- Keep in mind who will be using the interface documents and what types of information they will need. Avoid documenting more than is necessary.
- Be as specific and as precise as you can, remembering that an interface specification that can be interpreted differently by different people is likely to cause problems and confusion.

When writing down the semantics of a resource, follow these guidelines:

- Write down only those effects that are visible to a user: the actor invoking the resource, another element in the system, or a human observer of the system. Ask yourself how a user will be able to verify what you have said. If your semantics cannot be verified, the effect you have described is invisible, and you haven't captured the right information. Either replace it with a statement about something that the user will be able to observe or omit it. Although sometimes, the only visible effect of a resource is to disable certain exceptional conditions that might otherwise occur. For instance, a program that declares a named object disables the error associated with using that name before the object is declared.
- Make it a goal to avoid prose as the only medium of your description. Instead, try to define the semantics of invoking a resource by describing ways other resources will be affected. For example, in a stack object, you can describe the effects of `push(x)` by saying that `pop() returns x` and that the value returned by `g_stack_size()` is incremented by 1.
- If you use prose, be as precise as you can. Be suspicious of all verbs. For every verb in the specification of a resource's semantics, ask yourself exactly what it means and how the resource's users will be able to verify it. Eschew verbs that describe invisible actions—such as *creates* and *destroys*—and instead use statements about the effects of other resources as a result. Eliminate vague words, such as *should*, *usually*, and *may*. For operations that position something in the physical world, be sure to define the coordinate system, reference points, points of view, and so on, that describe the effects.
- Avoid describing resource use in place of specifying the semantics. Usage is a valuable part of an interface specification and merits its own section in the documentation. However, it is given as advice to users and should not be expected to serve as a definitive statement of resources' semantics. Strictly speaking, an example defines the semantics of a resource for only the single case illustrated by the example. The user might be able to

make a good guess at the semantics from the example, but we do not wish to build systems based on guesswork. We should expect users to use elements in ways the designers did not envision, and we do not wish to artificially limit the users.

- Avoid giving an implementation in place of specifying the semantics. Do not use code to describe the effects of a resource.

As in all architectural documentation, the amount of information conveyed in an interface specification may vary, depending on the stage of the design process captured by the documentation. If the interface is part of an element that is being developed in the system, the interface might be partially specified early in the design process: for example, "module `A` provides the following services." Later, when the responsibilities of the elements become stable, the interface specification is elaborated more fully, for example, "module `A` provides method `X` with signature `Y` and semantics `Z`."

5 A Standard Organization for Interface Documentation

This section suggests a standard organization for interface documentation (see Figure 1). You may want to change it to remove items not relevant to your situation or to add items unique to your business.

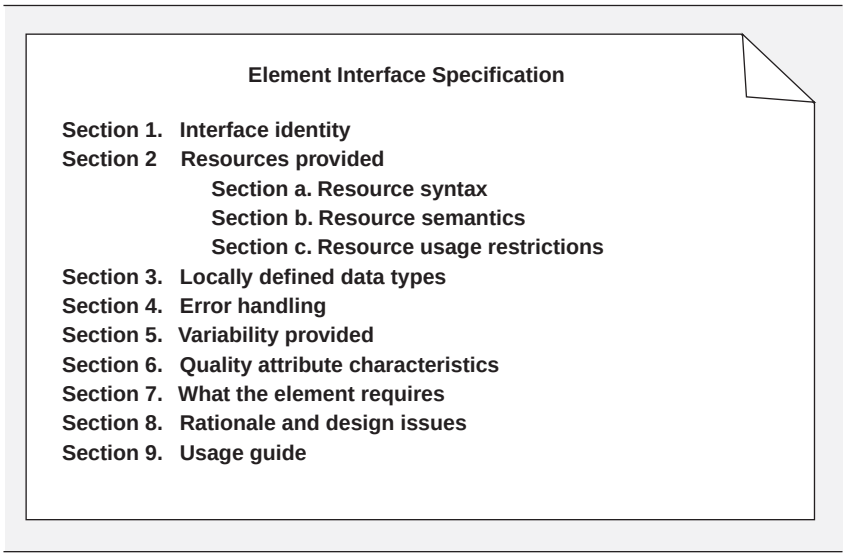

Figure 1: Outline of Interface Documentation

More important than which standard organization you use is the practice of using one. Use what you need to present an accurate picture of the element's externally visible interactions for the interfaces in your project:

1. *interface identity:* When an element has multiple interfaces, identify each one to distinguish them from one another. The most common means of doing this is to name an interface. Some programming languages, such as Java™, or frameworks, such as COM, even allow these names to be carried through into the implementation.[2] In some cases, merely

2. Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

naming an interface is not sufficient, and the version of the interface must be specified as well. For example, in a framework with named interfaces that has evolved over time, it could be very important to know whether you mean the V1.2 or V3.0 persistence interface.

2. *resources provided:* The heart of an interface document is the set of resources that the element provides to its actors. Define these resources by giving their syntax, their semantics—what happens when they're used—and any restrictions on their usage.

   a. *resource syntax:* This is the resource's signature, which includes any information that another program will need to write a syntactically correct program that uses the resource. The signature includes the name of the resource, the names and logical data types of arguments, if any, and so forth.

   b. *resource semantics:* What is the result of invoking this resource? Semantics come in a variety of guises, including (i) Assignment of values to data that the actor invoking the resource can access. The value assignment might be as simple as setting the value of a return argument or as far-reaching as updating a central database. (ii) Changes in the element's state brought about by using the resource. This includes exceptional conditions, such as side effects from a partially completed operation. (iii) Events that will be signaled or messages that will be sent as a result of using the resource. (iv) How other resources will behave differently in the future as the result of using this resource. For example, if you ask a resource to destroy an object, trying to access that object in the future through other resources will produce quite a different outcome— an error—as a result. (v) Humanly observable results. These are prevalent in embedded systems; for example, calling a program that turns on a display in a cockpit has a very observable effect—the display comes on. In addition, the statement of semantics should make it clear whether the execution of the resource will be atomic or may be suspended or interrupted.

   c. *resource-usage restrictions:* Under what circumstances may this resource be used? Perhaps data must be initialized before it can be read, or perhaps a particular method cannot be invoked unless another is invoked first. Perhaps there is a limit on the number of actors that can interact via this resource at any instant. Perhaps there is a limit of one actor that has ownership and is able to modify the element, whereas others have only read access. Perhaps only certain resources or interfaces are accessible to certain actors to support a multilevel security scheme.

      Notions of persistence or side effects can be relevant here. If the resource requires other resources to be present or makes other assumptions about its environment, that should be documented. Some restrictions are less prohibitive; for example, Java interfaces can list certain methods as *deprecated,* meaning that users should not use them, as they will likely be unsupported in future versions of the interface. Usage restrictions are often documented by defining *exceptions* that will be raised if the restrictions are violated.

3. *locally defined data types:* If any interface resource uses a data type other than one provided by the underlying programming language, the architect needs to communicate the definition of that data type. If the data type is defined by another element, a reference to the definition in that element's documentation is sufficient. In any case, programmers writing elements using such a resource need to know (a) how to declare variables and constants of the data type, (b) how to write literal values in the data type, (c) what operations and comparisons may be performed on members of the data type, and (d) how to convert values of the data type into other data types, where appropriate.

4. *error-handling capability:* Describe error conditions that can be raised by the resources on the interface. Because the same error condition might be raised by more than one resource, it is often convenient to simply list the error conditions associated with each resource and define them in a dictionary (i.e., this section). Common error-handling behavior can also be defined here.

5. *any variability provided by the interface:* Does the interface allow the element to be configured in some way? Any configuration parameters and their effects on the semantics of interactions in the interface must be documented. Examples of variability include capacities—such as of visible data structures—that can be changed easily. Name and provide a range of values for each configuration parameter, and specify the time when its actual value is bound.

6. *quality attribute characteristics of the interface:* The architect needs to document what quality attribute characteristics, such as performance or reliability, the interface makes known to the element's users. This information may be in the form of constraints on implementations of elements that will realize the interface. The qualities you choose to concentrate on and make promises about will depend on the context.

7. *what the element requires:* What the element requires may be specific, named resources provided by other elements. The documentation obligation is the same as for resources provided: syntax, semantics, and any usage restrictions. Two elements sharing this interface information—one providing it and the other requiring it—might each reference a single definition. If the element is being developed as an independent reusable component, that information needs to be fully documented. What the element requires may be expressed as something more general, such as "the presence of a process scheduler that will schedule in a fair, priority-based fashion." Often, it is convenient to document such information as a set of assumptions that the element's designer has made about the system. In this form, they can be reviewed by experts who can confirm or repudiate the assumptions before the design has progressed too far.

8. *rationale and design issues:* As with rationale for the architecture or architectural views at large, the architect should also record the reasons behind the design of an element's interface. This rationale should explain the motivation behind the design; constraints and compromises; alternative designs that were considered and rejected and why; and any insight the architect has about how to change the interface in the future.

9. *usage guide:* Sections 2b and 7 of the interface specification above document an element's semantic information on a per-resource basis. This sometimes falls short of what is needed. In some cases, semantics need to be reasoned about in terms of how a broad number of individual interactions interrelate. Essentially, a *protocol* of interaction is involved that is documented by considering multiple interactions simultaneously. These protocols could represent the complete behavior of the interaction or patterns of usage that the element designer expects to be used repeatedly. In general, if interacting with the element via its interface is complex, the interface documentation might include a static behavioral model, such as a state machine, or examples of carrying out specific interactions in the form of trace-oriented scenarios.

6 Stakeholders of Interface Documentation

Different stakeholders of architectural documentation have different needs and expectations. Interface documentation is no exception. Some of the stakeholders of interface documentation and the kinds of information they require are as follows:

- *builder of an element,* who needs the most comprehensive documentation of an interface. The builder needs to see any assertions about the interface that other stakeholders will see and perhaps depend on, so that he or she can make them true. A special kind of builder is the *maintainer*, who makes assigned changes to the element.
- *tester of an element,* who needs detailed information about all the resources and functionality provided by an interface, because this is usually what is tested. The tester can test only to the degree of knowledge embodied in the element's semantic description. If the required behavior for a resource is not specified, the tester will not know to test for it, and the element may fail to do its job. A tester also needs information about what is required by an interface, so that a test harness can be built, if necessary, to mimic the resources required.
- *developer using an element,* who needs detailed information about the resources provided by the element, including semantic information. Information about what the element requires is needed only if the requirements are pertinent to resources the developer uses.
- *analyst*, whose information needs depend on the types of analyses being conducted. For a performance analyst, for example, the interface document should give information that can feed a performance model, such as computation time required by resources. The analyst is a prime consumer of any quality attribute information contained in an interface document.
- *system builder,* who focuses on finding *provides* for each *requires* in the interfaces of elements going together to build a system. Often, the focus is more on the syntactic satisfaction of requirements—Does it build?—than on the semantic satisfaction of requirements. This role often uses information that is not of interest to most of the other stakeholders, such as what version of the Java String class an element uses.
- *integrator,* who also puts the system together from its constituent elements but has a stronger interest in the behavior of the resulting assemblage. Hence, the integrator is more likely to be concerned with the semantic rather than syntactic matching of *requires* and *provides* among the elements' interfaces.

- *product builder,* a special kind of integrator who exploits the variability available in the elements to produce different instantiations of them. These instantiations can then be assembled into a suite of similar but differing products. Ease of integration is also a key factor for the customer, who takes on aspects of the integrator's role when comparing vendors' products.
- *architect* looking for assets to reuse in a new system, who often starts by examining the interfaces of elements from a previous system. The architect may also look in the commercial marketplace to find off-the-shelf elements that can be purchased and do the job. To see whether an element is a candidate, the architect is first interested in the general nature and capabilities of the resources it provides to determine what aspects of the interface are pertinent to the design. The architect is also interested in a basic understanding of what resources are required. As the architect continues to qualify the element, he or she becomes more interested in the precise semantics of the resources, their quality attributes, and any variability that the element provides.
- *manager*, who is likely to use interface documents for planning purposes. Managers can apply metrics to gauge the complexity and then infer estimates for how long it will take to develop an element that realizes the interface. Depending on the metrics, information might be required about the size of the interface and the contained functionality but not on further details. Managers can also spot special expertise that may be required, and this will assist them in assigning the work to qualified personnel.

7 Notation

7.1 Notation for Showing the Existence of Interfaces

The *existence* of interfaces can be shown in the primary presentations by using most of the graphical notations available for architecture. Figure 2 shows some examples using an informal notation. The existence of an interface can be implied even without using an explicit symbol for it. If a relationship symbol joins an element symbol and the relationship type involves an interaction—as opposed to, say, "is a subclass of," that implies that the interaction takes place through the element's interface.

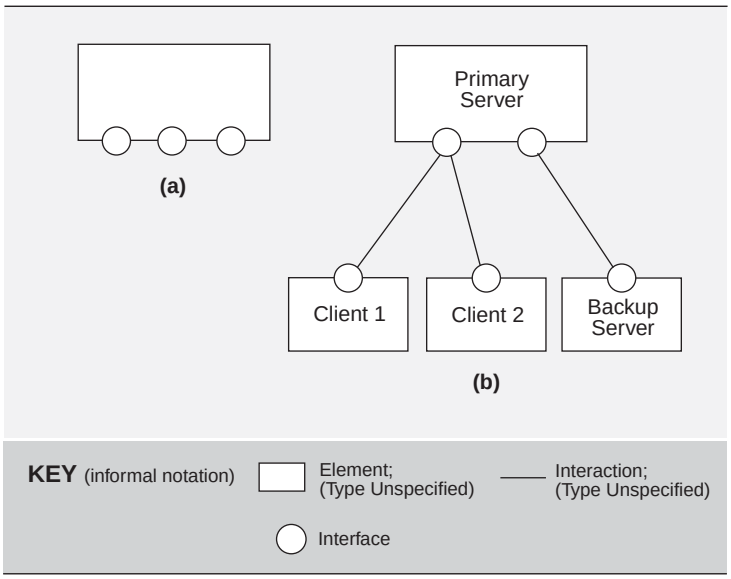

Figure 2: Sample Graphical Notation

Graphical notations for interfaces typically show a symbol on the boundary of the icon for an element. Lines connecting interface symbols denote that the interface exists between the connected elements. Graphical notations like this can show only the existence of an interface, not its definition. (a) An element with multiple interfaces. For elements with a single interface, the interface symbol is often omitted. (b) Multiple actors at an interface. Clients 1 and 2 both interact with Primary Server via the same interface.

Sometimes, interfaces are depicted by themselves, without an associated element. When actors are shown interacting through this interface, it indicates that any element implementing the in-

terface can be used. This is a useful means of expressing a particular kind of variability: the ability to substitute realizing elements, as shown in Figure 3(a). We say that an interface is *realized by* the element that implements it. Graphically, this is shown as a line resembling relationships among elements, as shown in Figure 3(b).

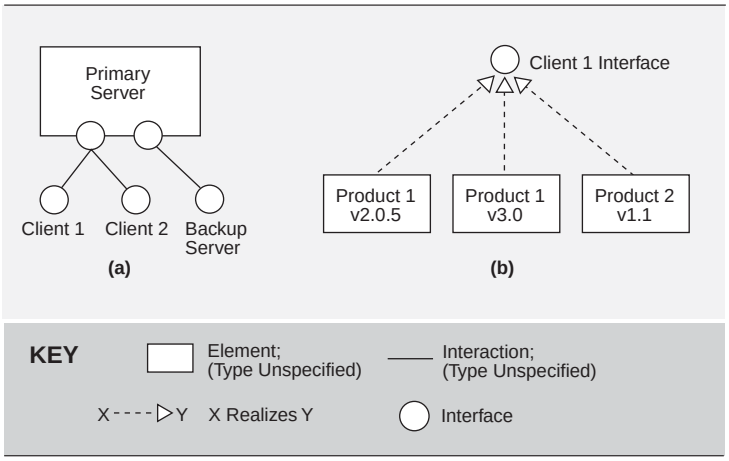

Figure 3: Showing Interfaces Separately

An interface can be shown separately from any element that realizes it, thus emphasizing the interchangeability of element implementations. (a) Another version of Figure 2(b), showing the primary server interacting with the interfaces of Clients 1 and 2 and Backup Server, without showing these elements. The emphasis here is on the interface. (b) An interface shown by itself emphasizes that many elements can realize it. If a specific set of possibilities has been identified, their candidacy can be shown graphically by using a figure like this.

Figure 4 illustrates how interfaces are shown in the Unified Modeling Language (UML). Although it shows the existence of an interface, Figure 4 reveals little about the definition of an interface: the resources it provides or requires, and the nature of its interactions. This information must be provided in the supporting documentation that accompanies the primary presentation.

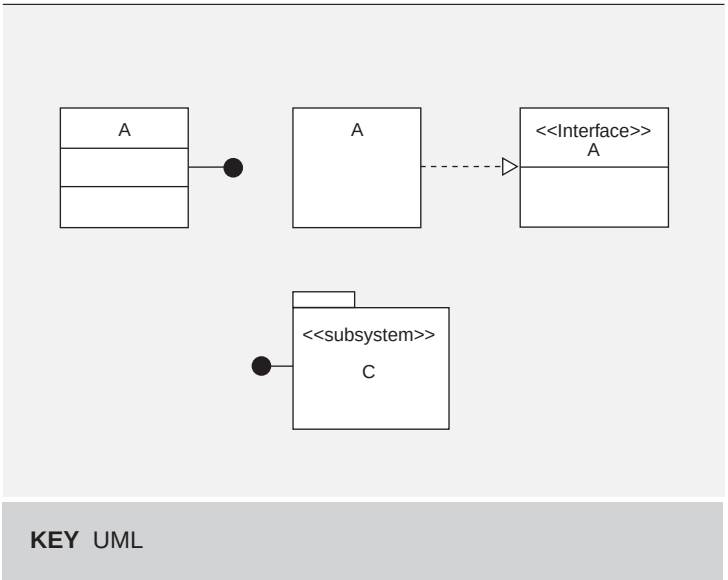

Figure 4: Showing Syntactic Information About Interfaces in UML

UML uses a “lollipop” to denote an interface, which can be appended to classes and subsystems among other things. UML also allows a class symbol, a box, to be stereotyped as an interface; the open-headed dashed arrow shows that an element realizes an interface. The bottom part of the class symbol can be annotated with the interface’s signature information: method names, arguments, argument types, and so on. The lollipop notation is normally used to show dependencies from elements to the interface; the box notation allows a more detailed interface description, such as the operations provided by the interface.

Use an explicit interface symbol in your primary presentations if

- Any element has more than one interface.
- You wish to emphasize the interface for an element: for example, if you are making provisions for multiple elements that realize the same interface.

Although it’s never wrong to show interfaces explicitly, it is not necessary to do so if

- No element has more than one interface.
- You wish to reduce the visual clutter of the diagrams.

7.2 Notations for Conveying Syntactic Information

The Object Management Group's (OMG's) interface definition language, IDL, is used in the CORBA[3] community to specify interfaces' syntactic information. IDL provides language constructs to describe data types, operations, attributes, and exceptions. But the only language support for semantic information is a comment mechanism.

Most programming languages have built-in ways to specify the signature of an element, for example, C header (.h) files and Ada package specifications. Finally, using the `«Interface»` stereotype on a class in UML, as shown in Figure 4, provides the means for conveying some syntactic information about an interface. At a minimum, the interface is named; in addition, the architect can specify signature information.

7.3 Notations for Conveying Semantic Information

Natural language is the most widespread notation for conveying semantic information. Boolean algebra is often used to write down preconditions and postconditions, which provide a relatively simple and effective method for expressing semantics. Traces are also used to convey semantic information by writing down sequences of activities or interactions that describe the element's response to a specific use.

Semantic information often includes the behavior of an element or one or more of its resources. In that case, any number of notations for behavior come into play. See the related technical note [Bachmann 02] for more information on describing behavior.

7.4 Notations Summary

No single notation adequately documents interfaces; practitioners must use a combination of notations. When showing the existence of interfaces in the views' primary presentations, use the graphical notation of choice. Use one of the syntactic notations to document the syntactic portion of an interface's specification. Use natural language, Boolean algebra for preconditions and postconditions, or any of the behavior languages to convey semantic information. Document patterns of usage, or protocols, as rich connectors, or show usage scenarios accompanied by examples of how to use the element's resources to carry out each scenario.

3. CORBA stands for Common Object Request Broker Architecture.

8 Examples

Following are a few examples of interface documentation, each of which exemplifies a different model for documenting interfaces. In each example, we point out what each model does and does not show.

8.1 SCR-Style Interface

The first example comes from a U.S. Navy software engineering demonstration project, called the Software Cost Reduction (SCR) project. One of the project's goals was to demonstrate model software architecture documentation, including interfaces. The example shown here is for a module that generates other modules, which, in turn, create and maintain tree data structures. The interface is shown for both the generator and the generated elements. The generated module lets actors create and manipulate tree data structures with characteristics determined by the generation step.

In the SCR style, each interface document begins with an introduction that identifies the element and provides a brief account of its function. An example introduction is shown in Figure 5. SCR-style interfaces do not include a usage guide per se. However, note how the introduction explains basic concepts and talks about how the element can be used.

```
TREE.1 Introduction

This module provides facilities for manipulating ordered trees. A
tree is a finite non-empty set T of nodes partitioned into disjoint
non-empty subsets { {R}, T1, ..., Tn }, n>=0, where R is the root
of T and each subset Ti is itself a tree (a subtree of R). The root
of each Ti is a child of R, and R is its parent. The children of R
(siblings of one another) are ordered, being numbered from 1 to n,
with child 1 the eldest and child n the youngest. Every node also
stores a value.

The size of the tree T is the number of nodes in T. The degree of
a node is the number of children it has; a node of degree 0 is a
leaf. The level of a node in a tree is defined with respect to the
tree's root: the root is at level 1, and the children of a level N
node are at level N+1. The height (sometimes also called depth) of
a tree is the maximum level of any node in the tree.

Using the facilities defined in section TREE.2.1, a user provides
(1) a name N for a type whose variables can hold values that denote
tree nodes, and (2) the type D of values that a tree node can hold.
This generates a submodule that defines the type N and implements
the operations on variables of type N specified in section
TREE.2.2. These operations include creating, deleting, and linking
nodes, and fetching and storing the datum associated with each
node.
```

Figure 5: Introduction of Sample SCR-Style Interface

SCR uses a special bracketing notation to denote different kinds of terms: `$data type literal$`, `!+semantic term+!`, `%exception%`, and `#configuration parameter#`. These brackets (`$`, `!+`, `%`, and `#`) convey to a reader what type of term it is, and therefore in which dictionary it can be located.

The next part of an SCR interface is a table similar to those shown in Figures 6 and 7. This table specifies the syntax of the resources and provides a quick-reference summary of those resources: in this case, method-like routines called access programs. The programs are named, their parameters are defined, and the exceptions detected by each are listed. Parameters are noted as I (input), O (output), I-OPT (optional input), or O-RET (returned as function results). This Interface Overview provides the signature for the resources in a language-independent fashion.

TREE.2.1 Generator Access Program

Program	Parameters	Parameter Info	Exceptions
++gen++	p1: id; I p2: name; I p3: typename; I p4: integer; I p5: integer; I p6: string; I-OPT p7: integer; O-RET	value for !<ID>! value for !<N>! value for !<D>! value for !<capacity>! value for !<max fanout>! exception handler command return code	%%bad capacity%% %%bad id%% %%bad max fanout%% %%bad name%% %%bad typename%% %%cannot write%% %%conflict%% %%io error%% %%recursive%% %%system error%% %%too long%%

Figure 6: Interface Overview of Generator Access Program ++gen++

TREE.2.2 Access Programs of Generated Module

Program	Parameters	Parameter Info	Exceptions
Programs that inquire about the universe of nodes			
+g_avail+	p1: integer; O_RET	!+avail+!	None
Programs that affect the structure of trees			
+add_first+ +add_last+	p1: !<N>!; I p2: !<N>!; I	reference node node to adopt	%not a node% %already a child% %is root of tree% %too many children%

Figure 7: Interface Overview of Access Programs of Generated Module (excerpt)

At this point, the syntax of the resources has been specified. Semantics are provided in two ways as illustrated in Figure 7, where overviews of two types of programs are shown. First, for programs that simply return the result of a query—called *get* programs and prefixed with the letter g—the returned argument is given a name, and its value is defined in the term dictionary, as exemplified in Figure 10. These programs have no effect on the future behavior of the element. Second, each of the other programs has an entry in an Effects section of the interface specification, such as the one shown in Figure 8, to explain its results. You can think of this section as a precondition/postcondition approach to specifying semantics, except that the preconditions are implied by the exceptions associated with each resource. That is, the precondition is that the state described by the exception does not exist. In the following, note how each statement of effects is observable; that is, you could write a program to test the specification.

For example, as shown in Figure 8, one effect of calling `+add_first+(p1)` is that an immediately following call to `+g_num+(p1)` will return the previous value incremented by one.

```
Effects

    Note: A program name in single quotes refers to the value returned by
    that program before the call to the program being defined.

    +add_first++

        g_num+(p1) = 1+'+g_num+'(p1)

        +g_nth+(p1,1) = p2

        For all i:integer such that (1 < i and i <= 1+'+g_num+'(p1)),

              +g_nth+(p1,i) = '+g_nth+'(p1,i-1)

              +g_num+(p1)>1 ==> (

                      +g_next+(p2)='+g_nth+'(p1,1)

                      and +g_prev+('+g_nth+'(p1,1))=p2 )

        +g_parent+(p2)=p1

        For all n:!<N>!, '+g_is_in_tree+'(p1,n) ==>

              (+g_size+(n) = '+g_size+'(n) + +g_size+(p2)

              and For all k:!<N>!,

              '+g_is_in_tree+'(k,p2)==> +g_is_in_tree+(k,n) )
```

Figure 8: Effects of Program +add_first+ Shown in Figure 7

An SCR-style interface continues with a set of dictionaries, such as those shown in Figures 9 - 12, that explain, respectively, the data types used, semantic terms introduced, exceptions detected, and configuration parameters provided. Configuration parameters represent the element's variability.

TREE.3 Locally Defined Data Types

Type	Definition
integer	Common type as defined in the numeric data types module
`!<N>!`	The data type stored in the tree module generated by the program ++gen++ with this type given as p2.

Figure 9: Locally Defined Data Types (excerpt)

TREE.4 Dictionary

Term	Definition
!+avail+!	The number of new nodes that can be created without an intervening call to +destroy_tree+. Initially = #max_num_nodes#
!+equal+!	p1 and p2 denote the same node (i.e., p1 and p2 contain the same value). Assignment of *a* to *b* makes *a* and *b* denote the same node.

Figure 10: Dictionary (excerpt)

TREE.5 Exceptions Dictionary

Exception	Definition
%already a child%	+g_is_node+(+g_parent+(p2))
%is root of tree%	+g_is_in_tree+(p1,p2)
%not a node%	For some input parameter pj of type !<N>!, ~+g_is_node+(pj)
%too many children%	For +add_first/last+: +g_num+(p1) = #max_num_children# For +ins_next/prev+: +g_num+(+g_parent+(p1)) = #max_num_children#

Figure 11: Exceptions Dictionary (excerpt)

TREE.6 System Configuration Parameters

Parameter	Definition
##max_capacity##	The maximum value of #max_num_nodes# for any generated submodule
##max_max_fanout##	The maximum value of #max_num_children# for any generated submodule
#max_num_children#	The maximum number of children that any node can have (= !<max fanout>!)
#max_num_nodes#	The maximum number of nodes that can exist at a time (= !<capacity>!)

Figure 12: System Configuration Parameters (excerpt)

Following the dictionaries, an SCR-style interface includes background information: design issues, such as those shown in Figure 13, rationale, implementation notes, and a set of so-called basic assumptions that summarize what the designer assumed would be true about all elements realizing this interface. Those assumptions form the basis of a design review for the interface.

TREE.7 Design Issues

1. How much terminology to define in the introduction. Several terms (leaf, level, depth) are defined in the introduction but are not used anywhere else in this specification. These terms have been defined here only because they are expected to prove useful in the specifications of modules that use trees.

2. How to indicate a nonexistent node. How is the fact that a node has no parent, nth child, or older or younger sibling to be communicated to users of the module? Two alternatives were considered: (a) Have the access programs that give the parent, etc., of a node return a special value analogous to a null pointer; (b) Have additional access programs for determining these facts. Option (a) was chosen because (1) it allows a more compact interface with no less capability, (2) it allows a user to make a table of nodes, some entries of which are empty, much more conveniently, and (3) it has the minor advantage of resembling the common linked implementation of trees, and thus may be viewed as more natural. Note that (a) may mimic (b) quite simply; comparing the result of the returned value with the special null value is equivalent to node has a parent, eldest child, or whatever. If the set of values of type !<N>! is defined to include a null value, then (b) may also mimic (a), since (b) is then a superset of (a).

3. How to move from node to node. "Moving from node to node" consists of getting the node that bears the desired relation to the first node. Several ways of accessing siblings were considered: (a) Sequentially, allowing moves to the next or previous sibling in the sequence. (b) By an index, allowing moves to the nth of the sequence of siblings. (c) Sequentially, but allowing moves of more than one sibling at a time. Option (c) seemed of marginal utility and was thus not included. Option (b) was included for generality. Although (a) is not strictly necessary if (b) is available, (a) was nevertheless also included because (a) can usually be implemented in a considerably more efficient manner.

TREE.8 Implementation Notes: none

TREE.9 Assumptions (excerpt)

1. The children of a node must be ordered.

2. It suffices that construction of trees be possible by a combination of creation of trees consisting of single nodes and attachment of trees as subtrees of nodes.

3. For our purposes, the following manipulations of trees are sufficient: (1) Replication of a tree (2) Addition of a subtree at either end of the list of subtrees of a node (3) Insertion of a subtree before or after a subtree in the list of subtrees of a node (4) Disassociation of a subtree from the tree that contains it

Figure 13: Design Issues, Implementation Notes, and Assumptions (excerpt)

Not shown in this example is an efficiency guide that lists the time requirements of each resource, the SCR analog to the quality attribute characteristics that we prescribe in our outline for an interface. Other quality attributes could be described here as well. The one piece of interface information that SCR-style interface specifications do not provide is what the element requires.

8.2 IDL

A small sample interface specified in OMG's IDL is shown in Figure 14. This interface is for an element that manages a bank account. The element provides resources to manage a financial account with attributes of "balance" and "owner." The operations provided include "deposit" and "withdraw." Although syntax is specified unambiguously in this type of documentation, semantic information is largely missing. For example, can a user make arbitrary withdrawals? Withdrawals only up to the current account balance? Up to a daily limit? Up to a minimum balance? If any of these restrictions is true, what happens if it's violated? Is the maximum permissible amount withdrawn, or is the transaction as a whole canceled?

IDL by itself is inadequate when it comes to fully documenting an interface, primarily because IDL offers no language constructs for discussing the semantics of an interface; without expression of the semantics, ambiguities and misunderstandings will abound.

```
interface Account {
        readonly attribute string owner;
        readonly attribute float balance;
        void deposit (in float amount);
        void withdraw (in float amount);
};
interface CheckingAccount: Account {
        readonly attribute float overdraft_limit;
        void order_new_checks ();
};
interface SavingsAccount: Account {
        float annual_interest ();
};
interface Bank {
        CheckingAccount open_checking (in string name, in float
        starting_balance);
        SavingsAccount open_ savings (in string name, float
        starting_balance);
};
```

Figure 14: An Example of IDL for an Element in a Banking Application [Bass 98, pg. 177]

8.3 Custom Notation

The High-Level Architecture (HLA) was developed by the U.S. Department of Defense (DoD) to provide a common architecture for distributed modeling and simulation. To facilitate intercommunication, HLA allows simulations and simulators, called federates, to interact with each other via an underlying software infrastructure known as the Runtime Infrastructure (RTI). The interface between federates and an RTI is defined in IEEE standard 1516.1 [IEEE 00]. The RTI provides services to federates in a way that is analogous to how a distributed operating system provides services to applications. The interface specification defines the services provided by the RTI and used by the federates. This is an example in which the focus is on defining an interface that will be realized by a number of different elements.

HLA was designed to facilitate interoperability among simulations built by various parties. Hence, simulations can be built by combining elements that represent different players into what is called a federation. Any element that realizes that the HLA interface is a viable member of the simulation will be able to interact meaningfully with other simulation elements that are representing other active parties.

Because of the need to ensure meaningful cooperation among elements that are built with little knowledge of one another, a great deal of effort went into specifying not just the syntax of the interface but also the semantics. The extract from the HLA Interface Specification presented in Figure 15 describes a single resource, a method, of the interface. Lists of preconditions and postconditions are associated with the resource, and the introduction provides a context for the resource and explains its use within the context of the full HLA interface. The resource, a method, is called Negotiated Attributed Ownership Divestiture.

The full HLA interface specification contains more than 140 resources like the one in Figure 15, and the majority have some interaction with other resources. For example, using some resources will cause the preconditions of the presented resource to no longer be true with respect to specific arguments. There are a number of such restrictions on the order in which the resources can be used.

To facilitate an understanding of the implicit protocol of usage among the resources, the HLA interface specification presents a summary of this information. Figure 16 depicts the constraints on the order of use of a specific set of the resources. This type of summary information is valuable in providing both an introduction to the complexities of an interface and a concise reminder to those already familiar with the interface. Without the summary, users would need to carefully read all the preconditions and postconditions of the 140 resources to reveal the restrictions. Such a reading is not trivial, and it is unrealistic to expect every user of the interface document to go through this kind of exercise.

Negotiated Attribute Ownership Divestiture
Overview

The *Negotiated Attribute Ownership Divestiture* service shall notify the runtime infrastructure (RTI) that the joined federate no longer wants to own the specified instance attributes of the specified object instance. Ownership shall be transferred only if some joined federates accept. When the RTI finds federates willing to accept ownership of any or all of the instance attributes, it will inform the divesting federate using the *Request Divestiture Confirmation* service (supplying the appropriate instance attributes as arguments). The divesting federate may then complete the negotiated divestiture by invoking the *Confirm Divestiture* service to inform the RTI of which instance attributes it is divesting ownership. The invoking joined federate shall continue its update responsibility for the specified instance attributes until it divests ownership via the *Confirm Divestiture* service. The joined federate may receive one or more *Request Divestiture Confirmation* invocations for each invocation of this service since different joined federates may wish to become the owner of different instance attributes.

A request to divest ownership shall remain pending until either the request is completed (via the *Request Divestiture Confirmation* and *Confirm Divestiture* services), the requesting joined federate successfully cancels the request (via the *Cancel Negotiated Attribute Ownership Divestiture* service), or the joined federate divests itself of ownership by other means (e.g., the *Attribute Ownership Divestiture If Wanted* or *Unpublish Object Class Attributes* service). A second negotiated divestiture for an instance attribute already in the process of a negotiated divestiture shall not be legal.

Supplied Arguments
- Object instance designator
- Set of attribute designators
- User-supplied tag

Returned Arguments
- None

Preconditions
- The federation execution exists.
- The federate is joined to that federation execution.
- An object instance with the specified designator exists.
- The joined federate knows about the object instance with the specified designator.
- The joined federate owns the specified instance attributes.
- The specified instance attributes are not in the negotiated divestiture process.
- A Save is not in progress.
- A Restore is not in progress.

Postconditions
- No change has occurred in instance attribute ownership.
- The RTI has been notified of the joined federate's request to divest ownership of the specified instance attributes.

Exceptions
- The object instance is not known.

Figure 15: Example of Documentation for an Interface Resource, Taken from the HLA [IEEE 00, pg. 104]

Note that the one thing that the IDL example presented very clearly—the syntax of the resources—is lacking in what has been shown so far in the HLA example. In fact, the HLA interface documentation distinguishes between what it calls an "abstract interface document" (shown in Figure 15) and a number of different programming-language representations of the interface, each of which is specified in a manner similar to the IDL example. This separation is

an example of how an interface document can be packaged into units that are appropriate for different stakeholders. The semantic specification is sufficient for architects examining the HLA for potential use. Developers of elements implementing the interface, on the other hand, need both the semantic specification and one or more of the programming-language representations for syntactic information.

Figure 16 shows how to convey a resource's usage constraints using a state diagram.

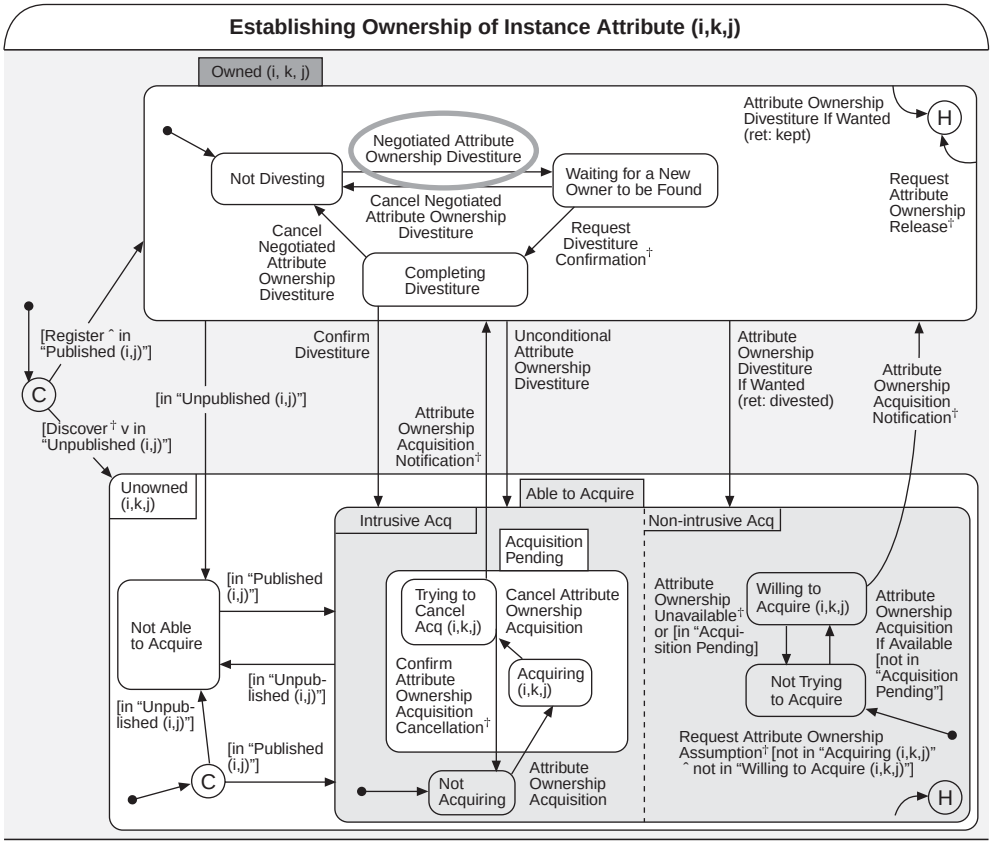

Figure 16: Sample Statechart

This statechart shows the constraints on the order of use of a specific set of resources. Statecharts like this one show an entire protocol in which a resource is used. The method described earlier is in the top center, highlighted in a circle. This shows the states during which the method can be invoked, and the state that is entered when it is invoked [IEEE 00, pg. 97].

8.4 XML

The Extensible Markup Language (XML) is a language for describing documents of structured information. As such, the XML can be used to document the information that will be exchanged across an interface. Data elements can be defined as XML documents. A sample data element, a personal record, is shown in Figure 17.

```
<person>
        <firstName>John</firstName>
        <lastName>Doe</lastName>
        <address>
           <street>123 Main St.</street>
           <city>anywhere</city>
           <state>somewhere</state>
           <zipCode>12345</zipCode>
           <country>US</country>
        </address>
</person>
```

Figure 17: Sample Data Element, a Personal Record

In this example, `person` is the root-level element of the XML document and contains elements `firstName`, `lastName`, and `address`. The latter, in turn, is made up of five other elements.

Using the XML to exchange information at runtime offers several benefits:

- All information is textual, making it easily readable by humans and portable across platforms.
- It is possible to include a description of what constitutes a valid document in an XML document itself; alternatively, a reference to such a description, identified by a URI (uniform resource indicator), can be supplied.
- Actors exchanging information via the XML need not conform to exactly the same version of an interface. It is a simple task for one actor to read the subset of an XML document that it understands and ignore the rest of the document.
- Tool support, such as browsers and parsers, is readily available for a variety of commonly used programming languages.

Document type declarations (DTDs) or schemas can be used to document the types of elements allowed within a document and to constrain the order in which those elements can be arranged. Either form can be used for runtime validity checking or as a simple documentation aid.

However, DTDs provide only syntactic interface documentation at best. Just as when producing IDL-based interface documentation, the burden of specifying semantic information is left to the documenter of an XML interface. The XML contains no effective language constructs for conveying semantic information.

The Simple Object Access Protocol (SOAP) describes a framework wherein XML documents are exchanged by actors as a means of implementing an RPC mechanism. The exact documents that are exchanged are left to be defined as part of any application using the SOAP. So, although the standard does provide a bit of semantic information in terms of the role that the XML documents serve—requests and responses—it is still up to the documenter to provide application semantics to each document type: for example, what an “execute” document instructs an actor to do and what the permissible responses are.

The XML is a useful means of representing data used in interfaces and provides a convenient way to specify the resource syntax portion of an interface. But the XML does not absolve the documenter of the responsibility to fill in the resource semantics portion.

9 Summary

- All elements have interfaces.
- Interfaces are two way, consisting of *requires* and *provides* information.
- An element can have multiple interfaces and multiple actors at each interface.
- An architect must carefully choose what information to put in an interface specification, striking a balance between usability and modifiability. In an interface document, include only information on which you are willing to let people rely.
- Follow the template given in Figure 1, making sure to address the needs of the interface specification's stakeholders.
- In graphical depictions, show interfaces explicitly if elements have more than one or if you want to emphasize the existence of an interface through which interactions occur. Otherwise, interfaces can be implicit.
- Many notations for interface documentation show only syntactic information. Make sure to include semantic information as well.

References

[Bachmann 00] Bachmann, F.; Bass, L.; Carriere, J.; Clements, P.; Garlan, D.; Ivers, J.; Nord, R.; & Little, R. *Software Architecture Documentation in Practice: Documenting Architectural Layers* (CMU/SEI-2000-SR-004, ADA377988). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2000. <http://www.sei.cmu.edu/publications/documents/00.reports/00sr004.html>.

[Bachmann 01] Bachmann, F.; Bass, L.; Clements, P.; Garlan, D.; Ivers, J.; Little, R.; Nord, R.; & Stafford, J. *Documenting Software Architectures: Organization of Documentation Package* (CMU/SEI-2001-TN-010, ADA396052). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2001. <http://www.sei.cmu.edu/publications/documents/01.reports/01tn010.html>.

[Bachmann 02] Bachmann, F.; Bass, L.; Clements, P.; Garlan, D.; Ivers, J.; Little, R.; Nord, R.; & Stafford, J. *Documenting Software Architecture: Documenting Behavior* (CMU/SEI-2002-TN-001, ADA3399792). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2002. <http://www.sei.cmu.edu/publications/documents/02.reports/02tn001.html>.

[Bass 98] Bass, L.; Clements, P.; & Kazman, R. *Software Architecture in Practice*. Reading, MA: Addison Wesley, 1998.

[IEEE 00] *IEEE Standard No.: 1516.1-2000: Standard for Modeling and Simulation (M&S) High Level Architecture (HLA)—Federate Interface Specification.* New York, NY: Institute of Electrical and Electronics Engineers (IEEE), 2001.<http://shop.ieee.org/store>.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. **AGENCY USE ONLY** (leave blank)	2. **REPORT DATE** June 2002	3. **REPORT TYPE AND DATES COVERED** Final
4. **TITLE AND SUBTITLE** Documenting Software Architecture: Documenting Interfaces		5. **FUNDING NUMBERS** C — F19628-00-C-0003
6. **AUTHOR(S)** Felix Bachmann, Len Bass, Paul Clements, David Garlan, James Ivers, Reed Little, Robert Nord, Judith Stafford		
7. **PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)** Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213		8. **PERFORMING ORGANIZATION REPORT NUMBER** CMU/SEI-2002-TN-015
9. **SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)** HQ ESC/XPK 5 Eglin Street Hanscom AFB, MA 01731-2116		10. **SPONSORING/MONITORING AGENCY REPORT NUMBER**
11. **SUPPLEMENTARY NOTES**		
12.a **DISTRIBUTION/AVAILABILITY STATEMENT** Unclassified/Unlimited, DTIC, NTIS		12.b **DISTRIBUTION CODE**

13. **ABSTRACT** (maximum 200 words)

This is the fourth in a series of SEI reports on documenting software architectures. This report details guidance for documenting the interfaces to software elements. It prescribes a standard organization (template) for recording semantic as well as syntactic information about an interface. Stakeholders of interface documentation are enumerated, available notations for specifying interfaces are described, and three examples are provided.

14. **SUBJECT TERMS** software architecture, software interfaces, documentation, documentation template, API	15. **NUMBER OF PAGES** 46
	16. **PRICE CODE**

17. **SECURITY CLASSIFICATION OF REPORT** UNCLASSIFIED	18. **SECURITY CLASSIFICATION OF THIS PAGE** UNCLASSIFIED	19. **SECURITY CLASSIFICATION OF ABSTRACT** UNCLASSIFIED	20. **LIMITATION OF ABSTRACT** UL