

## Chapter 3: UML Essentials, Elements, and Artifacts

*There aren't enough symbols to cover all the subtly different things we want to say.*

—Desmond D'Souza and Alan Wills (1998, 371)

This chapter provides you with a high-level summary of the main elements and artifacts of the UML—enough to get started with the pattern language.

Obviously, a single chapter is scarcely adequate to even begin the exercise of learning about the UML. But there are many good books out there that can provide an introductory service for those not already using the UML, and this book isn't meant to cover their territory. Anyone with a working knowledge of the UML can skim this chapter with the proviso that it provides information based on the 1.3 beta version of the *UML Specification* (Rational Software Corporation 1999), and so it may be useful as an update.

Also, I take a somewhat different approach than is typical for introductions. One difference is that this chapter, like this book and the UML itself, is process-independent. And, where it's important to illustrate anomalies and differences that come from the different ways that processes can interpret the UML, I use Rational's Unified Process and Desmond D'Souza and Alan Wills' book, *Objects, Components and Frameworks with UML: The Catalysis Approach* (1998) as examples of almost orthogonal processes. I've tried to distinguish between process-specific uses and notations and those that are "vanilla" UML. Where there are notable anomalies, I draw attention to them.

The other way that this chapter is different is that I focus on the packaging and notational side of the UML rather than the detailed semantics of the UML constructs. I do this for two reasons:

- The packaging, organizing, and drawing side of the UML seems to be underrepresented in the literature. How to go about making a model has taken a back seat to using the UML in creating development artifacts—a subtle but significant difference.
- For a book like this, which is aimed at modeling rather than programming, understanding packaging and notation is crucial to using the UML effectively as a tool, just as knowing the syntax is crucial to using a programming language effectively. This chapter aims to clarify what has been a notable amount of confusion in this area, in writings about the UML.

For the most part, the semantics of the UML are sufficiently object-oriented to present little difficulty to anyone with a programming background in any of the object-oriented languages. Of course, some aspects of the UML semantics that are specific to modeling or that are idiomatic to the UML need to be explained, and I'll cover these as well.

Like patterns, a summary such as this one doesn't benefit from inspired originality or novel insights. So, I relied on tried-and-true sources, and I reworked existing material, rather than providing a new way of looking at the innards of the UML. The description that follows is primarily drawn from the *UML Specification, Version 1.3* (Rational Software Corporation 1999), fleshed out by readings from *UML Distilled: Applying the Standard Object Modeling Language* (Fowler, Scott, and Jacobson 1997) and *Objects, Components and Frameworks with UML: The Catalysis Approach* (D'Souza and Wills 1998).

## 3.1 Elements, Viewpoints, and Views

The idea of a *view* is a key one in the UML: It provides the basis for the UML notion of a model. However, it is *not* a formal construct in the UML (although it *is* in the Rational Unified Process [RUP], a distinction that is typically muddled by writers about the UML). Instead, it acts as an informal concept that helps situate the role—and responsibilities—of the modeler.

In the UML, a model is a complete representation of a system from a particular viewpoint (that is, an aggregation of a set of views from specific perspectives). At the same time, systems are logically composed of many nearly independent models, representing many different viewpoints; and they can be physically composed of many independent subsystems, each of which can also be treated as a system for modeling purposes.

A system itself is implicitly represented by a top-level model with subsidiary models representing specialized views. Each model is made up of diagrams and text. The *UML Specification* describes diagrams as "views of a model," each representing a particular perspective that the overall model aggregates and integrates (Rational Software Corporation 1999). All of this applies recursively to subsystems as well.

The perspective that a model represents is a combination of the model's purpose and the level of detail it provides. Any system can contain a myriad of viewpoints and models, which depend on the role of each viewer, the conceptual stance he or she brings to the viewing, and the ultimate purpose of the view. Any viewpoint can be expressed by a myriad of views and diagrams, depending on the interests of the audience being addressed.

The modeler represents these views, wearing different hats in turn, and adopting different viewpoints. A user's view will end up being expressed differently depending on whether the audience is senior management or a technical person. A management perspective might result in a management model that emphasizes architecture and minimizes technical detail.

At the most abstract, a system should be modeled from at least two viewpoints:

- Looking at the outside world—an interpreted reality
- Looking at the product—a constructed reality

As modelers, in order to build a system, we model our understanding of the context, requirements, practices, and constraints to ensure that we have the problem and problem setting right. We then model the architecture, specifications, design, implementation, and deployment of what the builders should build. The second viewpoint provides a blueprint that builders can work from and the documentation that management needs to get the product built, keep it working, and make it useful.

The advantage an object-oriented approach brings to modeling is that it allows a modeler to "bridge the chasm" (Jacobson, Booch, and Rumbaugh 1998a, 7) between the analysis and design models by providing a semantically consistent conceptual framework, regardless of the viewpoint involved. Things in the real world are represented as objects, in the same way that things in the constructed world are. The modeling modes are still distinct, but the language is very much the same.

Naturally, in an iterative development process, even the barrier between modeling modes is permeable, and there's a constant to-and-fro, as opposed to rigid distinctions. In fact, as I will discuss in Chapter 11, "Putting It All Together: Reflecting on the Work of Design," proper design practice requires the interweaving of these two modes.

Beyond the simple need for a minimum of two viewpoints, additional viewpoints can be added to the picture, depending on the expressive granularity the modeler feels is warranted. Individual processes will treat the idea of views differently, making them formal constructs (for example, RUP) or leaving them as informal concepts (for example, *Objects, Components and Frameworks with UML: The Catalysis Approach* [D'Souza and Wills 1998]). The UML also assumes that you may want to create your own kinds of diagrams and your own kinds of models, and you may want to extend the UML in various ways to establish your own local modeling dialect.

### 3.1.1 Models and Model Elements

To repeat: A UML model is an abstraction, a complete representation of a physical system. Models are about things, relationships, behaviors, and interactions in a system. Equally important, models are about how to organize this information because conceptual chunking is an important tool for successful abstracting.

The UML itself distinguishes two types of models:

- *Structural models*—Represent the pieces of a system and their relationships.
- *Dynamic models*—Represent the behavior of the elements of the system and their interactions.

Models are made up of model elements. Model elements are named uniquely in the context of a given namespace (usually a package) and have *visibility*, which defines how they can be used (and reused) by other model elements.

Visibility determines the way individual elements can connect with each other. Therefore, it is a critical part of managing the complexity of models via information hiding. Decisions about visibility can be powerful factors influencing decisions about the logical organization of models. It is one of the ways in which the UML is distinctly different from previous generations of modeling languages and tools—by leveraging and extending the notion of visibility from object-oriented languages themselves.

In the UML, model elements may be visible in one of three ways:

- *Public*—Any outside model element can see the model element.
  - *Protected*—Any descendent can see the model element.
  - *Private*—Only the model element itself, its constituent parts, and the model elements nested within it can see it.
-

**Note** - Individual parts of the UML adjust this general notion of visibility to suit specific circumstances. See the discussion of package visibility in Section 3.2, "Packages," later in this chapter.

---

Models and model elements are rendered graphically in diagrams and textually in specifications, and they are organized in packages.

In UML notation, model things are expressed as symbols or icons, and relationships and interactions are typically expressed by using various kinds of adorned lines. The lines themselves aren't just passive, decorated connectors; in the UML, they're also symbols with semantic content and underlying rules. And, as organizational elements, neither models nor diagrams have a specific notational form (that is, the UML symbol)—only packages do.

Of course, it's not the rendering of the individual elements that provides the meaning and meat to a model, but the way they all connect. The connection between a model, its diagrams, and their parts—the connection between a viewpoint and expressed views—is almost rhetorical. Intention, attention, and focus provide differences in level of detail, formality, content, and manner of presentation.

Models aren't just diagrams; in the UML, the diagrams are only the visual rendering of the model. Underlying the graphics of a model are the specifications of the model elements. These are composed in text, and may be a mix of the formal and the informal. The text is as important as the diagrams and in some cases more so: use cases are an example (see Section 3.6.2, "Use Case Diagram," later in this chapter). The *UML Specification* itself is an example of a model (although not a good one, if judged by its clarity and communicative effectiveness).

### 3.1.2 Diagrams

Semantically, *diagrams* express views of a model (that is, subsets of the viewpoint that each model represents). An individual model element can be presented in one or many of the diagrams for a model; in other words, a model element can be presented in many different ways and in many diagrams—but it must be presented in at least one diagram in one way.

Diagrams don't have any special shape assigned to them; they can be free-floating, bounded by a box, or contained within a package. Some other considerations are as follows:

- Neither geometry nor "geography" has much significance in a UML diagram. For the most part, a symbol's size or relative location generally has no semantic content, except for diagrams that have a time dimension.
- Diagrams are all two-dimensional because of the current limits imposed by available technology and tools, although some shapes are nominally three-dimensional, rendered in two dimensions (cubes, for example).
- Text can be used liberally within a diagram. Examples include expressing rules and, within symbols, identifying attributes.

Finally (for this introduction), the *UML Specification* (Rational Software Corporation 1999)

highlights three kinds of visual relationships in diagrams:

- *Connection*—Lines connect icons and symbols, forming connecting paths. Paths are *always* attached to symbols at both ends (that is, no dangling lines are allowed).
- *Containment*—Boxes, circles, and other fully enclosed shapes contain symbols, icons, and lines.
- *Visual attachment*—Elements that are close together may have a relationship suggested by their proximity. For example, a name above a line or next to a box may be interpreted as applying to the line or box.

## 3.2 Packages

A *package* is a grouping of UML elements, which can include diagrams and may include subordinate packages and other kinds of model elements. According to the *UML Specification*, packages "can be used for organizing elements for any purpose; the criteria to use for grouping elements together into one package are not defined within UML" (Rational Software Corporation 1999). Packages are probably the most important aspect of the UML from a modeling perspective; they play a role in UML modeling that is similar to the role classes play in programming.

Most UML books treat packages lightly. Because of their importance and this lack of attention elsewhere, I'll provide more details on them here than I will for the other UML symbols and artifacts.

Packages can be nested and can reference other packages. They provide the basis for configuration control, storage, and access control. They also provide the basis for *naming* model elements. They define the namespace for model elements; nested packages create a naming hierarchy. Because packages can't be instantiated, nominally their only formal function (in a system) is to provide this name space.

Each element in a package must be uniquely named. Elements in different packages may have the same name, but they are differentiated by the package name as an additional identifier. This capability becomes significant with components, with reuse, and as systems grows larger. A name collision between components can be resolved by placing them into separate packages and referencing their fully qualified name:

```
packageName::elementName
```

Individual packages *own* model elements directly; an element can belong only to one package. If the owning package is removed from the model, the owned model elements are removed as well. A package is "the basic unit of development product—something you can separately create, maintain, deliver, sell, update, assign to a team and generally manage as a unit" (D'Souza and Wills 1998, 285).

Elements contained in the same package can be related, and can be related to elements in a containing package at any level. Elements in contained packages are not visible; "packages do not see inwards," as the *UML Specification* puts it (Rational Software Corporation 1999). Elements in other packages (contained or separate) can be made available by importing or accessing the package(s) or by generalization.

Packages can import or access the contents of other packages. When one package imports another, it imports all those elements that are visible in the second package. Imported elements become part of the importing package's namespace (that is, they can be referenced without the qualification of a pathname), and may be used in relationships owned by the package. An access relationship keeps the namespaces separate, requiring that the accessing package reference elements in the accessed package use fully qualified names (ones that include the pathname). Packages can also "specialize" other packages in a generalization relationship that parallels inheritance among classes.

Containment and visibility are key characteristics of model elements in packages. Packages encapsulate the model elements they contain and define their *visibility* as private, protected, or public:

- *Private*—Elements that are not available at all outside the containing package
- *Protected*—Elements that are available only to packages with generalizations to the package owning the elements
- *Public*—Elements that are available also to importing and accessing packages (see the following sections)

Package visibility is a variation on the standard approach that the UML provides for the visibility of model elements. Martin Fowler rightly cautions against using the UML's standard version of visibility too glibly because it is a mishmash of compromises that don't necessarily correspond to the way visibility is used in your programming language. For object-oriented development, he suggests relying on the flavor of visibility supported by the language you're using (1997, 99). For component-based development, on the other hand, especially where packages form the basis for working with components (as in D'Souza and Wills' book), UML package visibilities are a critical tool. For components, UML visibility is a significant modeling concern, not just one that has a transitive interest, as a reflection of programming concerns. See *Objects, Components and Frameworks with UML: The Catalysis Approach* for an example (D' Souza and Wills 1998, 259–296).

Although *package* can be mapped to programming constructs, the UML version is far richer and more nuanced than any of the available programmatic translations. It is the core-modeling concept that distinguishes the UML from previous programming-oriented modeling approaches. Reflecting its importance, the idea of a package has undergone significant shifts during the brief life of the UML, contributing to user confusion.

UML 0.8 did not include packages; instead, *category* and *subsystem* provided logical and physical organizing mechanisms. UML 0.9 combined categories and subsystems into an all-purpose construct called *package*, initially limited to grouping classes. With *UML Specification, Version 1.3*, packages can be used to group a much broader range of model elements. It seems likely that package, as a modeling construct, will continue to be refined and refactored, especially given its utility in managing the complexity of components.

The downside of all this is that the literature on the UML varies in its treatment of packages, depending on the vintage of the *UML Specification* being referenced. However, the evolving notion of a package and its evolving usage is probably a key to understanding the changing environment of modeling (and development), and it is a key to the success of the UML in the post-object-oriented world of systems development.

### 3.2.1 Models: Packages of Views

Models are one type of package explicitly identified in the UML metamodel. Therefore, it is a good example of how to use packages. Conceptually, packages provide a way of organizing models that is analogous to a hierarchic directory structure of folders and files.

With the UML, the process you use (and the toolset) will dictate the structure you use to organize your models. Generally, a single top-level package provides the starting point for a hierarchy of views and models (organized in packages and expressed as diagrams). Rational's Rose CASE tool is organized this way, but its approach is at best a useful convention that reflects the RUP. D'Souza and Wills' book incorporates a more imaginative approach that leverages packaging to the hilt, based on factoring the system in a variety of ways (1998). These can accord with the categories of user, express the different architectural layers in a system, contain the rules that constitute a system's architectural style, hold the patterns that drive the design, and provide the means of organizing work and configuring the work products.

### 3.2.2 Subsystems: Packages of Behavior and Operations

Subsystems are the other type of package explicitly identified in the UML metamodel. Subsystems decompose systems, whereas models partition the logical abstractions that describe a system. Models should be "nearly independent," whereas each subsystem should be independent and complete—there should be a minimum of coupling between the parts of different subsystems.

The notion of subsystems that has been incorporated in the 1.3 version of the *UML Specification* has an enhanced flavor that hints at a cross between model and class. A subsystem is no longer a collection of modules or classes as it was for Grady Booch in *Object-Oriented Analysis and Design with Applications* (1994, 221) or Ivar Jacobson in *Object-Oriented Software Engineering* (1994, 148). Now, a *subsystem* is a means of representing "a behavioral unit in the physical system" and a collection of model elements. It is fundamentally a way of partitioning the system-to-be-implemented, and so is more physical than logical or conceptual. Although it is a chunking mechanism like a model, at its simplest it is meant to address the organization of executables and production artifacts rather than ideas.

A model can be divided into subsystems; subsystems can include one or more models. Conceptually, the whole physical system is represented at the top level by a single model. This model can then be subdivided as needed to suit the purposes of the modeling exercise. Ordinary subordinate models that are packaged as models are logical conveniences. Subsystems are miniature systems themselves, ones that do not overlap. In the course of development, as the focus of modeling shifts towards construction and the physical architecture becomes stabilized, subsystems provide a way of organizing work products that can be translated into real physical implementations.

A subsystem inherits all the naming, access, and dependency properties of a package. However, like a class and unlike a model, a subsystem provides interfaces and has operations. Its contents can be separated into specification and realization subsets:

- *Specification*—Operations and/or features, together with such elements as use cases and state machines

- *Realization*—Identifies those elements that physically implement the specification

Subsystems are one of those areas in the UML that are notably inconsistent and muddy. For example, on page 2-180, the *UML Specification* declares that a subsystem "may or may not be instantiable." On page 2-181, however, it says that subsystems "cannot be instantiated" while declaring that an instantiable subsystem has some of the semantics of a composite class, whereas one that is not instantiable has the semantics of a package (Rational Software Corporation 1999).

Subsystems are probably going to become much clearer when the next major revision happens. Meanwhile, the importance of subsystems is very much a matter of the process that you're using. The RUP relies on subsystems heavily; for D'Souza and Wills' book, they're mostly a convenient way to package infrastructural services.

### 3.2.3 Frameworks: Packages of Patterns

A framework can also be expressed by using a package, one "consisting mainly of patterns, where patterns are defined as template collaborations" (Rational Software Corporation 1999, 2-174). The generic package acts as a template with some elements parameterized (that is, acting as placeholders for elements in what amounts to a software pattern) such as the Gang of Four patterns or Martin Fowler's analysis patterns.

Although not high-profile members of the UML (that is, they don't have a specification as a separate ModelElement, such as Model and Subsystem), *frameworks* are the third legitimate use for packages.

D'Souza and Wills' book uses frameworks as a way of providing a consistent mechanism for organizing and reusing model elements across all stages of development (1998). An implementation model can be composed of code frameworks that reflect the frameworks that emerged in the analysis model and were refined in the design model. Seeing components as collaborations and systems as collaborations between components makes the use of patterns that are instantiated as frameworks an effective alternative to the models-and-subsystems approach of traditional object-oriented design.

In the same way that packages have evolved from groupings of classes, in D'Souza and Wills' book, a framework provides a more refined means of using patterns, especially as a way of enhancing reuse. Although collaboration remains the core of patterns and frameworks in D'Souza and Wills' book, catalysis frameworks go beyond the simplistic understanding of patterns as a collaboration between objects that underlies the way patterns are currently embodied in the UML (1998).

## 3.3 Extensions

One of the key capabilities of the UML is that it can be customized and tailored as needed with a number of *extension mechanisms*. Extensions aren't just a means of localizing the UML to provide a dialect for a project or organization. They also provide a means of cautiously evolving the language.

Extensions can be added to the language as a whole as an informal solution to a general need, and later (possibly) being translated into a full and distinctive member of the language. For example, Framework is currently handled by means of an extension, rather than having the full blessing associated with being a ModelElement, the way Model and Subsystem are handled.

There are three extension mechanisms currently available, and one way of packaging extended dialects of the UML in standardized form. The following is a basic introduction.

### 3.3.1 Tagged Values

*Tagged values* are properties of elements that are explicitly defined. They are information about the model or model element, not the system itself. The tag and its associated value are defined using the following form:

```
{tag = "value"}
```

For example:

```
{modeler = "L. DaVinci"}
```

with braces surrounding the tagged value. The tagged value can be placed inside a container or in close proximity to the model element being tagged. There are some predefined tags in the UML, such as *invariant*, *precondition*, and *postcondition*, which can be used (for example) to support the use of contracts.

### 3.3.2 Constraints

*Constraints* are semantic restrictions on a model element—essentially, rules or conditional statements. They can be user-defined or part of the UML generally. Like tagged values, they appear within braces, typically contained within the model element they relate to, or appear close by. For example:

```
{age >= 21}
```

There are a substantial number of UML-provided constraints.

### 3.3.3 Stereotypes

*Stereotypes* are the most powerful and used of the extension mechanisms. They provide the means to specialize existing ModelElements—in effect, by creating new ones. This way, limitations or special needs can be addressed without having to get an official change to the UML itself (although, as with tagged values and constraints, the UML includes a number of official stereotypes, and local stereotype generation needs to be handled with caution).

Currently, for example, *framework* is a stereotype of package, and *actor* is a stereotype of class. A stereotype is expressed by adding the name of the stereotype to the model element being extended. The name is enclosed in guillemets. Therefore, the framework looks like [Figure 3.1](#).

#### **Figure 3.1**

A package stereotyped as a framework.

### 3.3.4 Profiles

The UML provides a mechanism called a *profile* to act as a way of packaging a predefined set of stereotypes, tagged values, constraints, and notation icons to customize the UML for a specific domain or process. Because they are extensions, profiles don't alter the basic UML. Instead, they are intended to help in the creation and management of dialects that can be more or less local.

Two samples are included in the 1.3 Specification; one for the RUP (referred to as the Unified Process), and Business Modeling a la the Three Amigos: Ivar Jacobson, Grady Booch, and James Rumbaugh. Others have been proposed, notably for workflow and process modeling, and to handle persistence.

Although not intended this way, an organization or process profile might also be a way to document which of the UML's copious supply of formal constraints, tags, stereotypes, and keywords are legal—providing a filtering capability as well as an extending capability. A profile would then be included as part of the documentation of the architectural style for a development effort.

## 3.4 Symbols

The authors of the UML have made an effort to avoid overwhelming the user with too many distinctive symbols, and have also tried to provide notational similarities among conceptually related symbols (packages and classes, for example).

UML symbols can have content or just be iconic. Actually, the UML distinguishes between what it calls, awkwardly, two-dimensional symbols and icons:

- *Icons*—Fixed in size and shape, icons do not have to be attached to paths (but can be *terminators*: icons on the end of a path that qualify the meaning of the path symbol). They may be located within symbols.
- *Two-dimensional symbols*—Having the capability to shrink or grow, two-dimensional symbols can have compartments as well. Some two-dimensional symbols are also graphs—they contain nodes connected by paths.

### 3.4.1 Actor

An *actor* is something or someone outside the system that interacts directly with it—typically, a user of the system or another system/subsystem (see [Figure 3.2](#)). An actor participates in use cases and may be represented in other types of diagrams anywhere in the system model.

#### **Figure 3.2**

Actor.

### 3.4.2 Use Case/Collaboration

A *use case* is a sequence of interactions by an actor with the system, which yields observable value to the actor (see [Figure 3.3](#)). By the way, this is not the UML definition, which is deplorably fussy and technical, and seems to be an effort to make use cases into a type of class and what the UML calls a "behavioral thing."

**Figure 3.3**

Use case.

A *collaboration* is a collection of objects that interact to implement behavior (see [Figure 3.4](#)). Typically, a collaboration can be used to specify the realization of a use case or an operation. A collaboration can also be used to specify a software pattern, and a *parameterized* collaboration (that is, one with abstract participants that are replaced when the pattern is used) can specify an architectural pattern.

**Figure 3.4**

Collaboration.

### 3.4.3 Class/Object/Type/Active Class

A *class* is an abstraction of a set of possible objects that share the same attributes, operations, methods, relationships, and semantics (see [Figure 3.5](#)). A class may use a set of interfaces to specify collections of operations it provides to its environment. A *type* is a representation of a collection of objects without specifying the physical implementation as a class. Class and type use the same symbol in the UML.

**Figure 3.5**

Class.

An *object* is an instance of a class or an example of a type—the same symbol as class, but with the name underlined (see [Figure 3.6](#)).

**Figure 3.6**

Object.

An *active class* is a set of objects, each of which owns a thread of control (that is, it can initiate, control, and terminate a process or thread). (See [Figure 3.7](#).)

**Figure 3.7**

Active class.

### 3.4.4 Interface

An *interface* describes the visible operations of a class, component, or package (see [Figure 3.8](#)). It defines the services available from the implementing element.

**Figure 3.8**

Interface.

### 3.4.5 Component

A *component* is a physical, replaceable part of a system that packages implementation and provides the realization of a set of interfaces (see [Figure 3.9](#)). A component represents a physical piece of implementation of a system, including software code (source, binary, or executable) or equivalents

such as scripts or command files (Rational Software Corporation 1999, B-5).

### **Figure 3.9**

Component.

## **3.4.6 Node**

A *node* represents a processing resource that exists at run time, with at least a memory and often processing capability as well (see [Figure 3.10](#)). Nodes comprise computing devices and any other physical resources used in a system, such as people or machines.

### **Figure 3.10**

Node.

## **3.4.7 Package**

A *package* (see [Figure 3.11](#)) is a UML container used to organize model elements (refer to Section 3.2, "[Packages](#)," earlier in this chapter).

### **Figure 3.11**

Package.

## **3.4.8 State**

A *state* is the condition, status, or situation of an object as part of its lifecycle and/or as the result of an interaction (see [Figure 3.12](#)). A state may also be used to model an ongoing activity.

### **Figure 3.12**

State.

## **3.4.9 Note**

A *note* is used to provide explanatory text, such as comments in a model (see [Figure 3.13](#)). A note can also be used in specific ways, depending on the modeling dialect you adopt (requirements are a good example).

### **Figure 3.13**

Note.

## **3.5 Lines**

In the UML, *lines* are used to express messages (dynamic connections between model elements), "links" (relationships between model elements—the term *link* also has a formal meaning within the UML), and interactions. Generally, messages don't appear in structural models; links don't appear in dynamic models. But this, too, can be varied within a dialect. A basic set of line-based UML notation is explained in the following sections.

---

**Note** - Remember (as I mentioned when discussing visual relationships in Section 3.1.2, "Diagrams"), lines must be terminated in some fashion in the UML, either with a model element graphic or an icon.

---

### 3.5.1 Messages

*Messages* are used in interactions between model elements in *dynamic models*, those that represent the behavior in a system. Messages convey information between objects, for example, and trigger activities. There are four types of messages (as shown in Figure 3.14):

- *Simple message*—Control is passed from one object to another without providing details.
- *Synchronous message*—The sending object pauses to wait for results.
- *Asynchronous message*—The sending object does not pause to wait for results.
- *Return message*—This message indicates a return from a procedure call.

#### **Figure 3.14**

The four types of messages.

### 3.5.2 Relationships in General

Relationships are used in *structural models* to show semantic connections between model elements.

A *dependency* is what *The Unified Modeling Language User Guide* calls a "using" relationship (Jacobson, Booch, and Rumbaugh 1998a), one in which the connection between two things means that if one changes, it affects the other (see Figure 3.15). Dependencies can be used to identify connections between a variety of model elements, packages being a notable example. These are unidirectional relationships.

#### **Figure 3.15**

Dependency.

A *generalization* is a relation between two elements, in which one is a more general form of the other (see Figure 3.16). Class inheritance is represented this way, but generalization can be used more generally. Again, packages are an example.

#### **Figure 3.16**

Generalization.

An *association* is what the UML calls a *structural relationship*, mapping one object to another set of objects (see Figure 3.17). It is also used to identify the communication path between an actor and a use case. *The Unified Modeling Language Reference Manual* describes associations as "the glue that ties systems together" (Jacobson, Booch, and Rumbaugh 1998b, 47). In the UML, an association has a navigational sense to it as well (you can get there from here, from one object to another), which can

cause heartburn among some object methodology purists.

### **Figure 3.17**

Association.

A *realization* is a type of dependency relationship that identifies a *contractual link* between elements—a realizing element. For example, a class implements the behaviors in a specifying element; in this case, it is an interface (see [Figure 3.18](#)). A realization also links use cases and collaborations. (See Section 3.5.4, "[Relationships: Some Uses of Dependency.](#)")

### **Figure 3.18**

Realization.

## **3.5.3 Relationships: Some Types of Associations**

The UML considers aggregations and composites to be special forms of association with distinctive notations. Needless to say, the semantics of aggregates and composites is subject to much debate.

A *qualified association* is a plain association with an indication of what information to use when identifying a target object in the set of associated objects. In [Figure 3.19](#), bank account # is used to identify the customer it belongs to.

### **Figure 3.19**

Qualified association.

An *aggregation* represents a whole-part relationship. This contrasts with a plain association, which shows a relationship among/between peers, depending on the number (see [Figure 3.20](#)). In an aggregation, one element is the *whole* and the other(s) are the *parts*.

### **Figure 3.20**

Aggregation.

A *composition* is an aggregation that has strong ownership of its parts. Therefore, if the whole element disappears, the parts do, too (see [Figure 3.21](#)).

### **Figure 3.21**

Composition.

## **3.5.4 Relationships: Some Uses of Dependency**

Dependency relationships are frequently stereotyped in the UML to support the needs of particular types of diagrams or model elements. The following are some examples:

*Extends* provides a way of handling behavior that is optional in a use case (see [Figure 3.22](#)). The optional behavior is packaged in an extending use case and connected via an <<extends>> dependency.

### **Figure 3.22**

Extends dependency.

*Includes* provides a way of handling behavior that is common to a number of use cases (see [Figure 3.23](#)). The optional behavior is factored out, packaged in an included use case, and connected via an <<includes>> dependency.

**Figure 3.23**

Includes dependency.

*Imports* is a dependency between packages (see [Figure 3.24](#)). A receiving package can access the publicly visible elements of the package being imported.

**Figure 3.24**

Imports dependency.

### 3.5.5 Abstraction: Other Uses of Dependency

The UML includes a variety of explicit expressions of abstraction:

*Refinement* indicates a dependency between two elements at different levels of abstraction (see [Figure 3.25](#)). An example of this is related elements in different types of models, such as an analysis model and a design model, which are very important in D'Souza and Wills' book, but are hardly mentioned in the RUP.

**Figure 3.25**

Refinement.

*Realization* was introduced earlier as a full-blown significant relationship meriting its own line symbol. It can also be indicated by using a stereotype (see [Figure 3.26](#)). The *UML Specification* says, "Realization can be used to model stepwise refinement, optimizations, transformations, templates, model synthesis, framework composition, and so on" (Rational Software Corporation 1999).

**Figure 3.26**

Realization.

*Derivation* indicates that an instance of an element (the derived element) can be computed from the other element in the relationship (see [Figure 3.27](#)).

**Figure 3.27**

Derivation

A *trace* relationship shows connections between model elements or sets of model elements representing the same concept in different models (see [Figure 3.28](#)). Traces are mainly used for tracking requirements and changes across models.

**Figure 3.28**

Trace.

## 3.6 Diagrams

UML diagrams are where it all comes together. As I already mentioned, in the UML there is no formal way of bounding or containing a diagram (that is, no notation for the diagram itself), so there are no relationships between diagrams. Instead, diagrams are the graphical presentation vehicles for aspects of a model. They don't stand alone, but are meant to be part of the textual narrative that provides the model specification.

The UML specifically includes nine different diagrams in its documentation, which I'll discuss briefly here. However, these diagram types are process-dependent and suggestive, rather than prescriptive. The UML allows the modeler to combine any and all elements into diagrams, depending on the modeling needs at hand. In practice, of course, only certain combinations of elements are sensible, and these nine types are pretty much the "canonical" ones for object-oriented programming.

However, all of this really needs to be considered in the context of the process and the tool being used. Although the RUP includes all of these diagrams, D'Souza and Wills' book (for example) doesn't.

Detailing these diagrams and the alternatives can be a book by itself. Most of the literature on the UML focuses on these diagrams, so there are already many good books to choose from. Because this book is as process-independent as the UML, I'll leave detailed discussions of these diagrams to the many other more programming-oriented, books out there and to come.

### 3.6.1 Class Diagram

For traditional object-oriented development, the class diagram is the keystone for the system model. In UML terms, it is a view of the static structural model. It is *not* a formal partitioning of the model; therefore, individual class diagrams "do not represent divisions in the underlying model" (Rational Software Corporation 1999, 3-33).

Aside from classes, a class diagram can also contain interfaces, packages, relationships, and instances (such as objects and links). The *UML Specification* suggests that "a better name would be *static structural diagram*," but bows to tradition and brevity in sticking with *class diagram* (1999).

The UML recognizes the distinction between *class* and *type*; the nature of what is meant by *class* changes conceptually, depending on the development stage that the modeling effort is in. For those interested, Martin Fowler, Kendall Scott, and Ivar Jacobson do an admirable job of capturing and explaining the modeling nuances of class, type and object in *UML Distilled: Applying the Standard Object Modeling Language* (Fowler, Scott, and Jacobson 1997).

### 3.6.2 Use Case Diagram

For use case-driven development, the *use case diagram* is the keystone of the modeling effort. Use case diagrams show actors and use cases, together with their relationships. These include the following:

- *Associations* between the actors and the use cases
- *Generalizations* between the actors

- *Generalizations, extends, and includes* relationships among the use cases

The use cases may be enclosed by a rectangle to show the boundary of the containing system, and so on. As will be evident in the UML patterns, use case diagrams by themselves are essentially trivial. The real substance of a use case is in the text (narrative).

A use case captures the significant parts of an interaction with the system or some part of the system by the actor(s) defining the scope, context, and requirements. Use cases can be used in a variety of ways throughout the development effort (for example, as the basis for setting up the test environment and as a starting point for a user manual).

The following is an example of a basic use case, which describes making a phone call, in text format first and then the diagram (see [Figure 3.29](#)). The caller initiates the use case, and the receiver is a secondary actor. The system that the caller and actor interact with is, of course, the phone system. The alternate flows aren't completely described, but indicate the kind of conditions that alternate flows handle.

*Actors:*

Caller Receiver

*Normal Flow:*

The use case begins when Caller picks up the handset of telephone.

Caller listens for dial tone.

(Alternate flow: no dial tone)

Caller dials a phone number.

Phone System rings Receiver's phone.

(Alternate flow: Wrong number)

Receiver answers.

(Alternate flow: no answer)

Caller conducts conversation.

The use case ends when Caller hangs up phone.

*Alternate Flows:*

No dial tone

Wrong number

No answer

### **Figure 3.29**

Telephone call use case diagram.

## **3.6.3 Interaction Diagrams**

Interaction diagrams are used in the dynamic modeling of the system. There are two kinds: sequence diagrams and collaboration diagrams.

### **Sequence Diagram**

A *sequence diagram* shows an interaction arranged in time sequence: the objects (*not* classes) and the messages that pass between them when an interaction occurs. These are what Ivar Jacobson used to call interaction diagrams.

A sequence diagram has a list of participating objects across its top, shown as rectangles. Each object rectangle contains at least a name, always underlined to indicate that the rectangle is an object and not a class. Below each object rectangle, shown with a dotted line, is the *object lifeline*, the time-ordered visual framework for message exchanges between the objects (and with the system). A narrow vertical rectangle called the *activation* represents the period of time an object is actually performing an action (directly or through an intermediary, such as another object). Object messages appear as arrows with a text description. [Figure 3.30](#) shows the basic elements of a sequence diagram.

### **Figure 3.30**

Simple sequence diagram.

### **Collaboration Diagram**

A *collaboration diagram* also shows the passing of messages between objects, but focuses on the objects and messages and their order instead of the time sequence. The sequence of interactions and the concurrent threads are identified using sequence numbers. A collaboration diagram shows an interaction organized around the roles in the interaction and their links to each other, and shows the relationships among the objects playing the different roles. The *UML Specification* suggests that collaboration diagrams are better for real-time specifications and for complex scenarios than sequence diagrams.

[Figure 3.31](#) shows the telephone call example expressed as a collaboration diagram.

### **Figure 3.31**

Collaboration diagram showing static structure of objects and messages flowing between them.

## **3.6.4 State Diagrams**

*State diagrams* show the states, events, transitions, and activities for the system, depicted as *state machines*. They're part of dynamic modeling rather than structural modeling. They are used to describe the behavior of a model element such as an object or an interaction, describing "possible sequences of states and actions through which the element can proceed during its lifetime as a result

of reacting to discrete events (for example, signals, operation invocations)" (Rational Software Corporation 1999, 3-131).

The official name in the UML is *statechart diagram*, but *state diagram* pops up all over the place in the *UML Specification* and in much of the literature. The term *statechart* reflects its origins in Harel Statecharts, a long-used tool for modeling real-time systems, which the UML has modified to fit object-oriented development.

A state diagram is typically used to model the behavior of an object that needs a complete description of its discrete states. Many objects in standard business systems may not have significant states, in which case they will not need state diagrams. (On the other hand, for those developing real-time systems, a discussion of state diagrams and state modeling can be a book by itself.) A state diagram is typically associated with one and only one class, and lists all the states that a particular object can have during system operation.

Each rounded rectangle represents one state that an object can be in. A state diagram must represent all the states in which an object can find itself, as well as define an initial state the object will be in. An *initial state* is shown as a filled-in black circle. The *final state* of an object is not required if the object has no final state, such as in a system that is always running.

An object passes from one state to another following a *transition* triggered by an event within the system (typically, a method invocation). A transition is shown by drawing an arrow from one state to another and is associated with an event that triggers the transition.

[Figure 3.32](#) shows the state diagram for a light switch that shows both transition lines and events.

### **Figure 3.32**

State diagram showing the two states of a light switch.

## **3.6.5 Activity Diagrams**

An *activity diagram* is a "special case of a state diagram in which all (or at least most) of the states are action or subactivity states and in which all (or at least most) of the transitions are triggered by completion of the actions or subactivities in the source states" (Rational Software Corporation 1999, 3-151). In fact, they're basically sophisticated versions of flowcharts. They're intended to cover workflows and processes, and have much of the flavor of flowcharts without the negative "baggage." Their depiction as versions of state diagrams is subject to much criticism, but at least it helps to make them a consistent member of the UML family.

An activity diagram is attached to a class (which includes a use case for the UML), to a package, or to the implementation of an operation. The *UML Specification* says that they "focus on flows driven by internal processing (as opposed to external events)" (1999, 3-151). It recommends using them where the events involved represent the completion of internally generated actions (for procedural flow of control).

In an activity diagram, *swimlanes* are used to package the organizational boundaries within an activity model: they are used to show *who* is doing *what* in an activity model.

When it is necessary to indicate that two or more actions occur in parallel, a line called a

*synchronization bar* shows where each thread in the parallel actions halts, waiting until the other threads reach the same point.

[Figure 3.33](#) shows an activity model for the workflow of an order system. Organized around the responsibilities of the customer, sales, and the warehouse, swimlanes are the vertical lines partitioning the diagram and identifying the responsibilities for each activity.

### **Figure 3.33**

An order process with three swimlanes, each showing the responsibility of the business unit during the process.

## **3.6.6 Implementation Diagrams**

*Implementation diagrams* model implementation artifacts and considerations, including how the source code is structured and how (and where) the executables are made available. The *UML Specification* suggests that they also can be applied in a broader sense to business modeling. *Components* are the business procedures and documents, and the *run-time structure* is the organization units and resources (human and other) of the business (1999, 3-165). There are two types of implementation diagrams: component diagrams and deployment diagrams.

### **Component Diagrams**

*Component diagrams* show the structure of the code (see [Figure 3.34](#)). According to the *UML Specification*, they are "the dependencies among software components, including source code components, binary code components, and executable components" (1999). The nuances of *component* are such that it seems to be as malleable a term as *architecture* and *abstraction*.

### **Figure 3.34**

Component diagram showing the dependencies between components of a payment-processing system.

### **Deployment Diagrams**

*Deployment diagrams* show the structure of the runtime system (see [Figure 3.35](#)), including the "configuration of runtime processing elements and the software components, processes, and objects that live on them" (Rational Software Corporation 1999, 3-166). For business modeling, the *UML Specification* says runtime elements include workers and organizational units, and the software components include procedures and documents used by the workers and organizational units.

### **Figure 3.35**

Deployment diagram showing run-time nodes and components for a browser-based three-tier account management system.

## **3.7 Further Reading**

Martin Fowler, Kendall Scott, and Ivar Jacobson's book, *UML Distilled: Applying the Standard Object Modeling Language* (1997), is still the best practical introduction to the UML, despite having been pretty much the first out of the chute. It shows its age in spots, but combines modeling and

programming concerns and cuts the huge beast down to a digestible size.

All of the UML books by the Three Amigos are useful, although with varying degrees of often-unstated bias towards the RUP. The *Unified Modeling Language User Guide* seems badly organized when you try to actually use it, but has a wealth of real-life, useful suggestions and ideas, a number of which provided the starting point for patterns in this book (Jacobson, Booch, and Rumbaugh 1999a). The *Unified Modeling Language Reference Manual* is occasionally idiosyncratic in content, at least when compared with the *UML Specification, Version 1.3*, but it is the most up-to-date and least influenced by the RUP (Jacobson, Booch, and Rumbaugh 1998b). With its encyclopedia/dictionary-type approach, it sometimes provides the only way to get the right information on nuances of current meaning that end-run some of the inconsistencies in the *UML Specification*. *The Unified Software Development Process* is the best of the bunch, but, of course, is even more biased toward Rational's process than the other two... but at least it's meant to be (Jacobson, Booch, and Rumbaugh 1999a).

*Objects, Components and Frameworks with UML: The Catalysis Approach* by Desmond D'Souza and Alan Wills (1998) is the best antidote to the bias towards the RUP that shows up not just in the Three Amigos books, but in almost all the UML books. The first chapter is almost impenetrable, but gradually you get the sense of the catalysis approach and why it (and component-based development) really is different from traditional object-oriented thinking.

One of any number of short articles by D'Souza is probably required reading as a preamble to undertaking the book, though. Alternatively, try the patterns first and then read the rest of the text. I've reworked a number of the patterns from catalysis in this book because they express real best practices from mainstream modeling. Although the rest of the book requires a real paradigm shift in thinking to absorb properly, the patterns are good fundamental ideas expressed simply and well.

Finally, the *UML Specification, Version 1.3* (released in July 1999) is a necessary part of the library, although not one meant to be read. Because of the changes that have been made and some that are presaged, the 1.3 version needs to be available for occasional consultation. The language is frequently impossible, the odd inconsistencies are frustrating, but it *is* the source of final judgment. And it's free for downloading from the OMG and Rational Software Web sites.

© Copyright Pearson Education. All rights reserved.