04 526146 Ch01.qxd 6/17/03 8:59 AM Page 9

Chapter 1

What's UML About, Alfie?

In This Chapter

- Understanding the basics of UML
- Exploring the *whys* and *whens* of UML diagrams

So you've been hearing a lot about UML, and your friends and colleagues are spending some of their time drawing pictures. And maybe you're ready to start using UML but you want to know what it's all about first. Well, it's about a lot of things, such as better communication, higher productivity, and also about drawing pretty pictures. This chapter introduces you to the basics of UML and how it can help you.

http://www.pbookshop.com

Introducing UML

The first thing you need to know is what the initials *UML* stand for. Don't laugh — lots of people get it wrong, and nothing brands you as a neophyte faster. It's not the *Universal* Modeling Language, as it doesn't intend to model everything (for example, it's not very good for modeling the stock market; otherwise we'd be rich by now). It's also not the Unified Marxist-Leninists, a Nepalese Political party (though we hope you'll never get *that* confused). It is the University of Massachusetts Lowell — but not in this context. *UML* really stands for the *Unified Modeling Language.*

Well, maybe that's not the most important thing to know. Probably just as important is that UML is a standardized modeling language consisting of an integrated set of diagrams, developed to help system and software developers accomplish the following tasks:

- Specification
- Visualization
- Architecture design

04 526146 Ch01.qxd 6/17/03 8:59 AM Page 10

- Construction
- Simulation and Testing
- Documentation

UML was originally developed with the idea of promoting communication and productivity among the developers of object-oriented systems, but the readily apparent power of UML has caused it to make inroads into every type of system and software development.

Appreciating the Power of UML

UML satisfies an important need in software and system development. Modeling — especially modeling in a way that's easily understood — allows the developer to concentrate on the big picture. It helps you see and solve the most important problems now, by preventing you from getting distracted by swarms of details that are better to suppress until later. When you model, you construct an abstraction of an existing real-world system (or of the system you're envisioning), that allows you to ask questions of the model *and get good answers* — all this without the costs of developing the system first.

http://www.pbookshop.com

After you're happy with your work, you can use your models to communicate with others. You may use your models to request constructive criticism and thus improve your work, to teach others, to direct team members' work, or to garner praise and acclamation for your great ideas and pictures. Properly constructed diagrams and models are efficient communication techniques that don't suffer the ambiguity of spoken English, and don't overpower the viewer with overwhelming details.

Abstracting out the essential truth

The technique of making a model of your ideas or the world is a use of *abstraction.* For example, a map is a model of the world — it is not the world in miniature. It's a conventional abstraction that takes a bit of training or practice to recognize how it tracks reality, but you can use this abstraction easily. Similarly, each UML diagram you draw has a relationship to your reality (or your intended reality), and that relationship between model and reality is learned and conventional. And the UML abstractions were developed as conventions to be learned and used easily.

If you think of UML as a map of the world you see — or of a possible world you want — you're not far off. A closer analogy might be that of set of blueprints that show enough details of a building (in a standardized representation with

04 526146 Ch01.qxd 6/17/03 8:59 AM Page 11

lots of specialized symbols and conventions) to convey a clear idea of what the building is supposed to be.

The abstractions of models and diagrams are also useful because they suppress or expose detail as needed. This application of *information hiding* allows you to focus on the areas you need — and hide the areas you don't. For example, you don't want to show trees and cars and people on your map, because such a map would be cumbersome and not very useful. You have to suppress some detail to use it.

You'll find the word *elide* often in texts on UML — every field has its own jargon. Rumor has it that *elide* is a favorite word of Grady Booch, one of the three methodologists responsible for the original development of UML. *Elide* literally means to omit, slur over, strike out, or eliminate. UML uses it to describe the ability of modelers (or their tools) to suppress or hide known information from a diagram to accomplish a goal (such as simplicity or repurposing).

Chapter 2 tells you more about using these concepts of *information hiding* and *abstraction* during development.

http://www.pbookshop.com

Selecting a point of view

UML modeling also supports multiple views of the same system. Just as you can have a political map, a relief map, a road map, and a utility map of the same area to use for different purposes — or different types of architectural diagrams and blueprints to emphasize different aspects of what you're building — you can have many different types of UML diagrams, each of which is a different view that shows different aspects of your system.

UML also allows you to construct a diagram for a specialized view by limiting the diagram elements for a particular purpose at a particular time. For example, you can develop a *class diagram* — the elements of which are relevant things and their relationships to one another — to capture the analysis of the problem that you have to solve, to capture the design of your solution, or to capture the details of your implementation. Depending on your purpose, the relevant things chosen to be diagram elements would vary. During analysis, the elements that you include would be logical concepts from the problem and real world; during design, they would include elements of the design and architectural solution; and during implementation, they would primarily be software classes.

A *use case diagram* normally concentrates on showing the purposes of the system (use cases) and the users (actors). We call a use case diagram that has its individual use cases elided (hidden) a *context diagram,* because it shows the system in its environment (context) of surrounding systems and actors.

04 526146 Ch01.qxd 6/17/03 8:59 AM Page 12

Choosing the Appropriate UML Diagram

UML has many diagrams — more, in fact, than you'll probably need to know. There are at least 13 official diagrams (actually the sum varies every time we count it) and several semiofficial diagrams. Confusion can emerge because UML usually allows you to place elements from one diagram on another if the situation warrants. And the same diagram form, when used for a different purpose, could be considered a different diagram.

In Figure 1-1, we've constructed a UML class diagram that sums up all the major types of UML diagrams (along with their relationships), using the principle of *generalization*, which entails organizing items by similarities to keep the diagram compact. (See Chapter 2 for more information on generalization.)

In Figure 1-1, the triangular arrows point from one diagram type to a more general (or more abstract) diagram type. The lower diagram type is a *kind-of* or *sort-of* the higher diagram type. Thus a `Class Diagram` is a kind of `Structural Diagram`, which is a kind of `Diagram`. The diagram also uses a dashed arrow to indicate a dependency — some diagrams reuse the features of others and depend on their definition. For example, the `Interaction Overview Diagram` depends on (or is derived from) the `Activity Diagram` for much of its notation. To get a line on how you might use UML diagrams, check out the summary in Table 1-1.

http://www.pbookshop.com

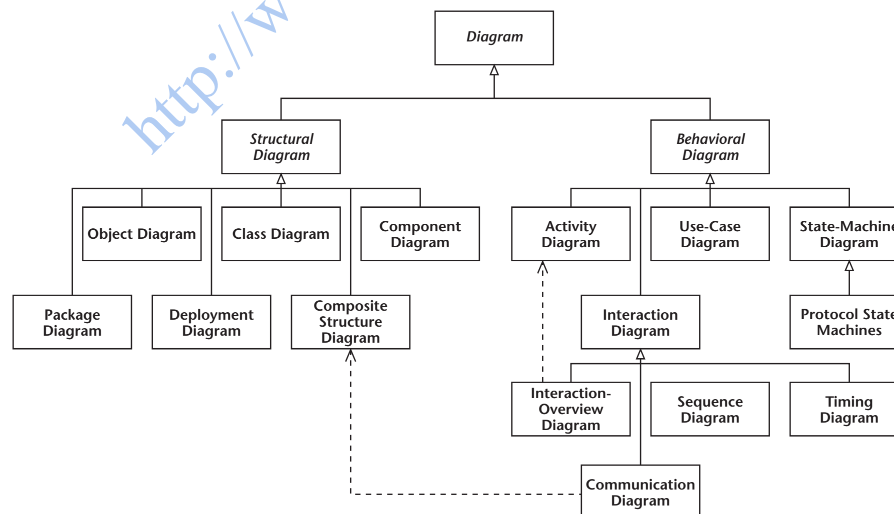

Figure 1-1: A class diagram of UML diagrams.

04 526146 Ch01.qxd 6/17/03 8:59 AM Page 13

Slicing and dicing UML diagrams

There are many ways of organizing the UML diagrams to help you understand how you may best use them. The diagram in Figure 1-1 uses the technique of organization by *generalization* (moving up a hierarchy of abstraction) and *specialization* (moving down the same hierarchy in the direction of concrete detail). (See Chapter 6 for more on generalization and specialization.) In Figure 1-1, each diagram is a subtype of (or special kind of) the diagram it points to. So — moving in the direction of increasing abstraction — you can consider a communication diagram from two distinct angles:

- It's a type of interaction diagram, which is a type of behavioral diagram, which is a type of diagram.
- It's derived from a composite structure diagram, which is a kind of structural diagram, which is a type of diagram.

After you get some practice at creating and shaping UML diagrams, it's almost second nature to determine which of these perspectives best fits your purpose.

This general arrangement of diagrams that we used in our Figure 1-1 is essentially the same as the UML standard uses to explain and catalog UML diagrams — separating the diagrams into *structural diagrams* and *behavioral diagrams*. This is a useful broad categorization of the diagrams, and is reflected in the categorizations in Table 1-1:

- **Structural diagrams:** You use structural diagrams to show the building blocks of your system — features that don't change with time. These diagrams answer the question, *What's there?*
- **Behavioral diagrams:** You use behavioral diagrams to show how your system responds to requests or otherwise evolves over time.
- **Interaction diagrams:** An interaction diagram is actually a type of behavioral diagram. You use interaction diagrams to depict the exchange of messages within a *collaboration* (a group of cooperating objects) en route to accomplishing its goal.

Table 1-1 UML 2 Diagrams and Some of Their Uses

Category	*Type of Diagram*	*Purpose*	*Where to Find More Information*
Structural diagram	Class diagram	Use to show real-world entities, elements of analysis and design, or implementation classes and their relationships	Chapter 7

(continued)

http://www.pbookshop.com

04 526146 Ch01.qxd 6/17/03 8:59 AM Page 14

Table 1-1 *(continued)*

Category	*Type of Diagram*	*Purpose*	*Where to Find More Information*
Structural diagram	Object diagram	Use to show a specific or illustrative example of objects and their links. Often used to indicate the conditions for an event, such as a test or an operation call	Chapter 7
Structural diagram	Composite structure diagram	Use to show the how something is made. Especially useful in complex structures-of-structures or component-based design	Chapter 5
Structural diagram	Deployment diagram	Use to show the run-time architecture of the system, the hardware platforms, software artifacts (deliverable or running software items), and software environments (like operating systems and virtual machines)	Chapter 19
Structural diagram	Component diagram	Use to show organization and relationships among the system deliverables	Chapter 19
Structural diagram	Package diagram	Use to organize model elements and show dependencies among them	Chapter 7
Behavioral diagram	Activity diagram	Use to the show data flow and/ or the control flow of a behavior. Captures workflow among cooperating objects	Chapter 18
Behavioral diagram	Use case diagram	Use to show the services that actors can request from a system	Chapter 8
Behavioral diagram	State machine diagram / Protocol state machine diagram	Use to show the life cycle of a particular object, or the sequences an object goes through or that an interface must support	Chapter 18
Interaction diagram	Overview diagram	Use to show many different interaction scenarios (sequences of behavior) for the same collaboration (a set of elements working together to accomplish a goal)	Chapter 13

http://www.pbookshop.com

04 526146 Ch01.qxd 6/17/03 8:59 AM Page 15

Category	*Type of Diagram*	*Purpose*	*Where to Find More Information*
Interaction diagram	Sequence diagram	Use to focus on message exchange between a group of objects and the order of the messages	Chapter 13
Interaction diagram	Communication diagram	Use to focus on the messages between a group of objects and the underlying relationship of the objects	Chapter 14
Interaction diagram	Timing diagram	Use to show changes and their relationship to clock times in real-time or embedded systems work	Rarely used, so we refer you to the UML specification

http://www.pbookshop.com

Because UML is very flexible, you're likely to see various other ways of categorizing the diagrams. The following three categories are popular:

- **Static diagrams:** These show the static features of the system. This category is similar to that of structural diagrams.
- **Dynamic diagrams:** These show how your system evolves over time. This category covers the UML state-machine diagrams and timing diagrams.
- **Functional diagrams:** These show the details of behaviors and algorithms — how your system accomplishes the behaviors requested of it. This category includes use-case, interaction, and activity diagrams.

You can employ UML diagrams to show different information at different times or for different purposes. There are many modeling frameworks, such as Zachman or DODAF (Department of Defense's Architecture Framework) that help system developers organize and communicate different aspects of their system. A simple framework for organizing your ideas that is widely useful is the following approach to answering the standard questions about the system:

- **Who uses the system?** Show the actors (the users of the system) on their use case diagrams (showing the purposes of the system).
- **What is the system made of?** Draw class diagrams to show the logical structure and component diagrams to show the physical structure.
- **Where are the components located in the system?** Indicate your plans for where your components will live and run on your deployment diagrams.
- **When do important events happen in the system?** Show what causes your objects to react and do their work with state diagrams and interaction diagrams.

04 526146 Ch01.qxd 6/17/03 8:59 AM Page 16

- **Why is this system doing the things it does?** Identify the goals of the users of your system and capture them in use cases, the UML construct just for this purpose.
- **How is this system going to work?** Show the parts on composite structure diagrams and use communication diagrams to show the interactions at a level sufficient for detailed design and implementation.

Automating with Model-Driven Architecture (MDA)

Model-driven architecture (MDA) is new way to develop highly automated systems. As UML tools become more powerful, they make automation a real possibility much earlier in the process of generating a system. The roles of designer and implementer start to converge. UML provides you with the keys to steer your systems and software development toward new horizons utilizing model-driven architectures.

In the past, after the designer decides what the system would look like — trading off the design approach qualities such as performance, reliability, stability, user-friendliness — the designer would hand the models off to the developer to implement. Much of that implementation is difficult, and often repetitious. As one part of an MDA approach to a project, UML articulates the designer's choices in a way that can be directly input into system generation. The mechanical application of infrastructure, database, user interface, and middleware interfaces (such as COM, CORBA, .NET) can now be automated.

http://www.pbookshop.com

Because UML 2 works for high-level generalization or for showing brass-tacks detail, you can use it to help generate high-quality, nearly complete implementations (code, database, user-interface, and so on) *from the models.*

In MDA, the Development Team is responsible for analysis, requirements, architecture, and design, producing several models leading up to a complete, but Platform-Independent Model (PIM). Then UML and MDA tools can generate a Platform-Specific Model (PSM) based on the architecture chosen and (after some tweaking) produce the complete application.

This approach promises to free the development team from specific middleware or platform vendors. When a new architecture paradigm appears — and it will — the team can adopt it without going back to Square One for a complete redevelopment effort. The combination of UML and MDA also promises to free development teams from much of the coding work. Although the required UML models are much more specific than most organizations are used to, their use will change the way developers make systems.

With the advent of MDA and its allied technologies, UML becomes a sort of executable blueprint — the descriptions, instructions, and the code for your system in one package. Remember it all begins with UML.

04 526146 Ch01.qxd 6/17/03 8:59 AM Page 17

Identifying Who Needs UML

Broadly speaking, UML users fall into three broad categories:

- **Modelers:** Modelers try to describe the world as they see it — either the world as is, whether it's a system, a domain, an application, or a world they imagine to come. If you want to document a particular aspect of some system, then you're acting as a modeler — and UML is for you.
- **Designers:** Designers try to explore possible solutions, to compare, to trade off different aspects, or to communicate approaches to garner (constructive) criticism. If you want to investigate a possible tactic or solution, then you're acting as a designer — and UML is for you.
- **Implementers:** Implementers construct solutions using UML as part of (or as the entire) implementation approach. Many UML tools can now generate definitions for classes or databases, as well as application code, user interfaces, or middleware calls. If you're attempting to get your tool to understand your definitions, then you're an Implementer — and (you guessed it) UML is for you.

http://www.pbookshop.com

To understand how you can benefit from UML, it will help to know how and why it was developed. It's based on successful and working techniques proposed by groups of Software Technology Vendors before the Object Management Group, and voted upon by the members.

Dispelling Misconceptions about UML

Many developers have several misconceptions about UML. Perhaps you do too, but after reading this book, you'll have the misconceptions dispelled:

- **UML is *not* proprietary.** Perhaps UML was originally conceived by Rational Software, but now it's owned by OMG, and is open to all. Many companies and individuals worked hard to produce UML 2. Good and useful information on UML is available from many sources (especially this book).
- **UML is *not* a process or method.** UML encourages the use of modern object-oriented techniques and iterative life cycles. It is compatible with both predictive and agile control approaches. However, despite the similarity of names, there is no requirement to use any particular "Unified Process" — and (depending on your needs) you may find such stuff inappropriate anyway. Most organizations need extensive tailoring of existing methods before they can produce suitable approaches for their culture and problems.
- **UML is *not* difficult.** UML is big, but you don't need to use or understand it all. You are able to select the appropriate diagrams for you

04 526146 Ch01.qxd 6/17/03 8:59 AM Page 18

needs and the level of detail based on you target audience. You'll need some training and this book (of course), but UML is easy to use in practice.

- **UML is *not* time-consuming.** Properly used, UML cuts total development time and expenses as it decreases communication costs and increases understanding, productivity, and quality.

The evolution of UML

In the B.U. days (that's Before UML), all was chaos, because object-oriented developers did not understand each other's speech. There were over 50 different object-oriented graphical notations available (I actually counted), some of them even useful, some even had tool support. This confusion, interfered with adoption of object-oriented techniques, as companies and individuals were reluctant to invest in training or tools in such a confusing field.

Still the competition of ideas and symbols did cause things to improve. Some techniques were clearly more suited to the types of software problems that people were having. Methodologists started to adopt their competitors' useful notation. Eventually some market leaders stood out.

In October 1994, Jim Rumbaugh of the Object Modeling Technique (OMT) and Grady Booch of the Booch Method started to work together on unifying their approach. Within a year, Ivar Jacobson (of the Objectory Method), joined the team. Together, these three leading methodologists joined forces at Rational Software, became known as the Three Amigos, and were the leading forces behind the original UML. Jim Rumbaugh was the contributor behind much of the analysis power of UML and most of its notational form. Grady Booch was the force behind the design detail capabilities of UML. Ivar Jacobson led the effort to make UML suitable for business modeling and tying system development to use cases.

The Three Amigos were faced with the enormous job of bringing order and consensus to the Babel of notation and needed input from the other leading methodologist about what works and what doesn't. They enlisted the help of the Object Management Group (OMG), a consortium of over 800 companies dedicated to developing vendor-independent specifications for the software industry. OMG opened the development of UML to competitive proposals. After much debate, politics, and bargaining, a consensus on a set of notation selected from the best of the working notation used successfully in the field, was adopted by OMG in November 1997.

Since 1997, the UML Revision Task Force (RTF) of OMG — on which one of your authors (okay, it was Michael) served — has updated UML several times. Each revision tweaked the UML standard to improve internal consistency, to incorporate lessons learned from the UML users and tool vendors, or to make it compatible with ongoing standards efforts. However, it became clear by 2000 that new development environments (such as Java), development approaches (such as component-based development), and tool capabilities (such more complete code generation) were difficult to incorporate into UML without a more systematic change to UML. This effort leads us to UML 2, which was approved in 2003.

http://www.pbookshop.com