Chapter 1

Why the UML?

Let's start off by looking at why you should learn about the Unified Modeling Language (UML).

Why Model Software?

Software just isn't getting any easier to develop, despite the best efforts of computer language inventors, tool developers, and process gurus. Even relatively small systems tend to have large amounts of complexity. This is what leads people to develop **models**, which are simplifications of reality that help them understand the complexity inherent in software.

Of course, a wide variety of models have been in use within various engineering disciplines for a long time. Aerospace engineers rely heavily on models that describe the forces acting on an airplane; electrical engineers use very large models in designing telephone switching systems; civil engineers would be lost without their blueprints. Models taking other forms, such as the simulations used in high finance and the storyboarding that Hollywood directors use, also play very important roles.

The UML has been designed to help the participants in software development efforts build models that will enable the team to visualize the system, specify the structure and behavior of that system, construct the system, and document the decisions made along the way.

Visualization

Models help a software development project team visualize the system they need to build that will satisfy the various requirements imposed by the project's stakeholders. The aphorism "A picture is worth a thousand words" holds especially true when it comes to the work involved in developing software: much of what's interesting about a given system lends itself to visual modeling better than plain text does.

The UML is specifically designed to facilitate communication among the participants in a project. Some aspects of the language are focused on the communication involving customers and developers; others, on that among system architects and database designers; still others, on that among developers working on different pieces of the system. By offering a set of well-defined diagrams, and precise notation to use on those diagrams, the UML gives everyone on the team the ability to understand what's going on with the system at any point in time with minimal risk of misinterpretation.

Specification

To specify a model, in UML terms, means to build it so that it's precise, unambiguous, and complete. Various aspects of the UML address the specification of the many decisions that have to be made as a system evolves.

- Models built early in the project help focus the thought processes of the stakeholders and enable them to explore their options with little risk and at relatively little cost.
- As work proceeds, the initial models get fleshed out as knowledge increases and a greater degree of precision is required. These intermediate models focus on the key concepts of the system and the mechanisms by which those concepts will be embodied.
- UML models with large amounts of detail generally serve as fairly complete descriptions of the most important features of the final system. If the modeling efforts have been rigorous, viewers will be able to trace the elements of the later-stage models to their initial incarnations in the rough sketches.
- UML models can also be constructed from an existing system to assist people in their efforts to maintain and extend the system's functionality.

Construction

Of course, the ultimate goal of a development project is working code. A healthy number of the UML's constructs have direct or indirect relationships with constructs offered by the most popular programming languages, including C++ and Java; other UML elements lend themselves nicely to tasks such as physical database modeling and network layout.

A technique called **forward engineering** involves generating code from models. Visual modeling tools such as Rational Rose make it easy to get code started from UML models. **Reverse engineering**, the construction (or reconstruction) of models from code, can also be very useful. Ideally, a project team will practice **round-trip engineering**, which encompasses both forward engineering and reverse engineering with the goal of keeping models and code in synch to maximum effect.

Documentation

The combination of UML models and the other kinds of work products that come out of a development effort generally forms a solid set of project documentation. For instance, a complete set of use cases (see Chapter 4) can serve as a strong foundation for user guides and related training materials, and the realizations of those use cases, as described in Chapter 5, lend themselves nicely to use by quality assurance people performing white-box testing.

In addition to helping people who weren't involved understand the thought processes underlying the development project, properly produced models and related documents will often be reusable, in part or wholesale, in the context of other projects.

What Makes a Good Software Model?

At the heart of the philosophy that underlies the UML are a set of qualities that, taken together, identify a model as useful and valuable.

- *A good model suits the plan of attack that the team takes toward solving part or all of the problem at hand.* A model that captures essential abstractions and ignores nonessential ones is likely to go a long way toward offering a high degree of enlightenment.

- *A good model allows different viewers to see different levels of detail at different times.* The stakeholders of a project come at a system being developed from varying viewpoints; a good model lends itself to inspection at levels ranging from the executive summary to the gritty low-level details.
- *A good model is connected to reality.* Rigorous development and continuous improvement of a model are likely to result in behavior that closely resembles what the team can expect to see when the real system is unveiled.
- *A good model works well with other models in capturing all of the essential elements of the system.* As discussed later in this chapter, and also in Chapter 2, the UML encourages the creation of several kinds of models that are, for the most part, independent, yet offer ample opportunities to trace the likes of data flows and control flows.

Where Did the UML Come From?

Around 1989, a healthy number of different ways to do object-oriented modeling were making their presence felt. The next several years were a period that's now generally referred to as the "methodology wars," or just "method wars." When the smoke cleared, in the mid-1990s, three of the contestants were seen by most of the object-oriented community as standing head and shoulders above the rest. Their names: Grady Booch, Jim Rumbaugh, and Ivar Jacobson.

Ivar Jacobson's Objectory process came at a system first and foremost from the standpoint of users and the ongoing evolution of their requirements (see Figure 1-1).

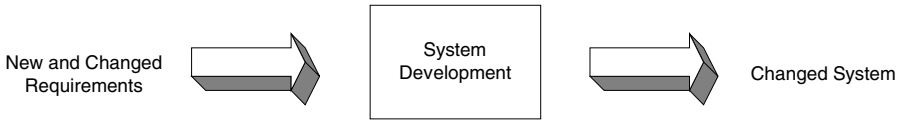

Figure 1-1: Jacobson's View of System Development

Jacobson's *Object-Oriented Software Engineering* (Addison-Wesley, 1992) introduced the OOSE method as a simplified version of Objectory, which he and a number of colleagues developed while building large telecommunications systems for Ericsson in Sweden.

The "white book," as it's commonly referred to, contained the first full-length description of use cases, which immediately became widely used and which now sits at the heart of the UML. Objectory, as expressed in terms of OOSE, also serves as the foundation for the Unified Process, which we'll look at in the next chapter.

Jim Rumbaugh's Object Modeling Technique (OMT) evolved out of extensive work with data-intensive systems, and thus was strongest in the area of modeling the problem space. Figure 1-2 shows the key elements of the first step of the OMT methodology, analysis.

Rumbaugh's *Object-Oriented Modeling and Design* (Prentice Hall, 1991) described, in great detail, how to build three separate models of a system—one each for the static structure, the control structure, and the computation structure—in order to understand the problem before implementing a solution. Much of the notation we'll look at in Chapters 3 and 6 evolved directly from the original OMT notation, and many people around the world still regard the material about mapping the object model (which captures the static structure) to relational databases as definitive nearly a decade after its appearance.

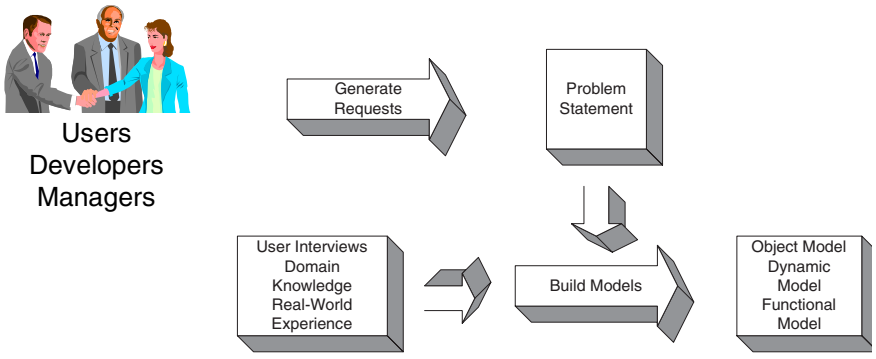

Figure 1-2: Rumbaugh's Elements of Analysis

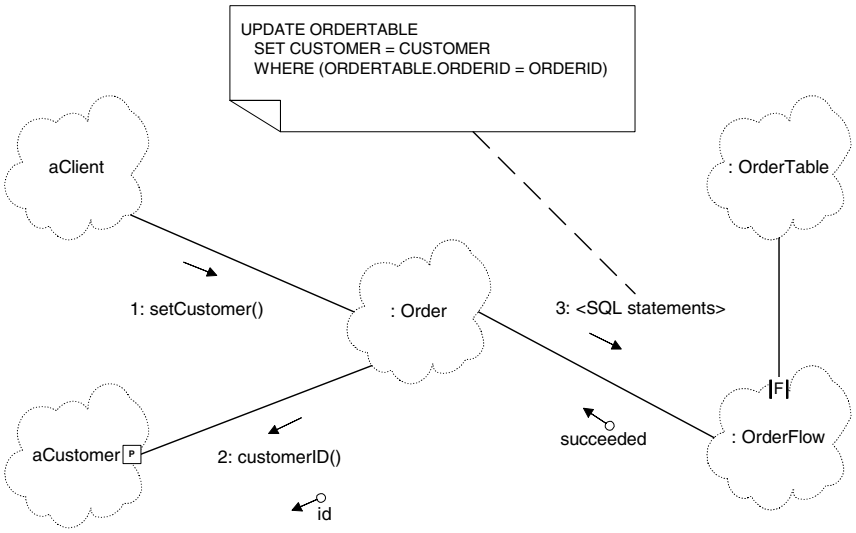

Figure 1-3: Booch's Detailed Design

The strengths of the Booch method (yes, that was the "official" name) were connected with detailed design and coding. Figure 1-3, which models a Structured Query Language (SQL) transaction, is representative.

Grady Booch's *Object-Oriented Design with Applications* (subsequently retitled *Object-Oriented Analysis and Design with Applications* [Addison-Wesley, 1994]) became a widely used college textbook. Its combination of rigorous attention to the details associated with building software systems and musings about architecture, philosophy, and cognitive science established Booch as perhaps *the* most influential figure in the object-oriented community.

The three approaches had enough common ground that mixing and matching became inevitable.

The initial seed of the Unified Method came in late 1994, when Rumbaugh left General Electric to join Booch at Rational. The company made the first version of the method public a year later, not long before Rational bought Jacobson's company and brought him into the fold. There followed the 0.9 Unified Method documentation, and then version 1.0 of the Unified Modeling Language.

Version 1.0 was what Rational submitted to the Object Management Group (OMG), the body that serves to define standards across many areas of computer science. After some negotiating with the OMG and the most prominent competitors in the race to standardization, UML 1.1 became the standard object-oriented modeling language in November of 1997. After a brief interlude during which a 1.2 version was in place, the OMG released version 1.3 of the UML, which is the version this book addresses.

Views of a System

The **architecture** of a system is the fundamental organization of the system as a whole. A system's architecture has a number of aspects, including static elements, dynamic elements, how those elements work together, and the overall architectural style that guides the organization of the system. Architecture also addresses issues such as performance, scalability, reuse, and economic and technological constraints.

Each of the various stakeholders in a software development project comes to the table with a different agenda. In turn, each stakeholder looks at the system from a different angle, and those angles are likely to change as the system evolves. The UML captures these angles as a set of five interlocking **views**. Each view reveals a particular set of aspects of the system from a given perspective, and hides, in effect, other aspects that are not of concern to the viewer. Figure 1-4 shows the five views of a system's architecture that the UML defines.

The **use case view** focuses on the scenarios executed by human users and also external systems. The contents of this view express what the new system will do without specifying how it will do those things. The next chapter takes a closer look at how this view influences the other views. Chapter 4 describes the key elements of the use case view.

The **design view** focuses on the things that form the vocabulary of the problem that the system is trying to solve and the elements of the solution to that problem. This view encompasses both static, or structural, aspects of the system and dynamic, or behavioral, aspects. Chapters 3 and 6 discuss the static/structural aspects of this view; Chapters 5, 7, 8, and 9 describe the dynamic/behavioral aspects.

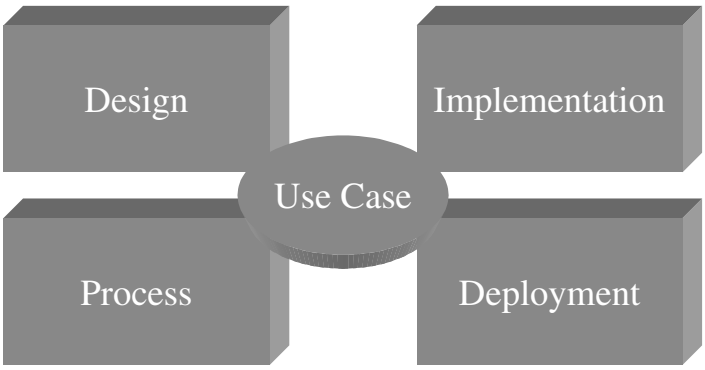

Figure 1-4: Five Views of a System

The **process view** focuses on those aspects of the system that involve timing and the flow of control. Elements of this view also address issues such as performance, scalability, and throughput. Chapters 5 and 7 contain material relevant to this view.

The **implementation view** focuses on the things that the project team assembles to form the physical system. This includes the source code, executable code, physical databases, and associated documentation, among other elements. Chapter 10 discusses these elements.

The **deployment view** focuses on the geographic distribution of the various software elements on hardware and the other physical elements that constitute the system. Chapter 10 also describes this view.