

CHAPTER

1

What Is UML?

What is the UML? In short, the Unified Modeling Language (UML) provides industry standard mechanisms for visualizing, specifying, constructing, and documenting software systems. However, such a static definition does not convey the tremendous growth of UML and the exciting opportunities of recent advances in the language. UML organizes thought with a dynamic set of features that support continuous evolution. This evolution has seen UML through a number of minor new releases (versions 1.0, 1.1, 1.2, 1.3, and 1.4, referenced as UML 1.x) into a major release (2) designed to extend UML's growth into new areas. The latest UML specification now spans hundreds of pages, with links to other specifications bringing the total into the thousands. The long answer to the question "What is the UML?" is found in the specifications that define the current edition of UML. However, those specifications, freely available for download, do not provide a practical overview of the key features of UML. This book seeks to address this need. This chapter provides some general background on software modeling, the historical roots of UML, and the basic goals of UML 2.

UML provides a language for describing system interactions, supported by a cohesive set of definitions managed by an official group, the Object Management Group (OMG). In this, UML at first resembles a software product more than a language; software versions improve "bugs" in the specification and add new "features" just like UML. However, like a language, UML has evolved dialects beyond the reach of official standards for such needs as data

2 Chapter 1

modeling, business modeling, and real-time development. The designers of UML intentionally gave UML the power for such evolution; UML has extension mechanisms that allow for creative new approaches. The implementation of UML, and especially UML 2, allows flexible language evolution that can keep up with the pace of change in software development. UML seeks to define concepts, to establish notation for communicating those concepts, and to enforce the related grammatical rules for constructing software models, allowing for automated tool support. The goal is to have the flexibility to apply to all software development while providing coherent enough definitions for automation. UML 1.x made a giant step toward this goal, and UML 2 furthers this effort. Put another way, UML provides a set of tools for all those involved in using information technology. This set of tools involves a rich set of graphical elements, including notation for classes, components, nodes, activities, ports, workflow, use cases, objects, states, and the relationships between all these elements.

Just as a dictionary does not explain how to write a book, UML itself offers no process for developing software. Just as a toolbox does not tell you how to use a hammer, UML does not recommend how to use the notation it provides in a model. Using UML to enhance project success requires skill, talent, and creativity. This book explains how to use the UML toolkit to improve software systems. So, this book not only discusses the specific elements of the UML specification, but also shows how to use these to model within a process to develop good software.

The Purpose of Modeling

A model represents highly creative work. There is no final solution and no correct answer that is checked at the end of the work. The model's designers, through iterative work, ensure that their models achieve the goals and the requirements of the project under construction. But a model is not final; it is typically changed and updated throughout a project to reflect new insights and the experiences of the designers. During the modeling, the best solutions are often achieved by allowing a high degree of brainstorming, during which different solutions and views are modeled and tested. By discussing different possibilities, the designers reach a deeper understanding of the system, and can finally create models of the systems that achieve the goals and requirements of the system and its users.

Communicating a model in a clear way to the many people involved in a software project represents a core feature of UML. The term *modeling* can lead to confusion, because it also applies to a number of different levels. Modeling shows not only the detailed specifications of software elements, but also high

level analysis, providing mechanisms to link the different layers of abstraction. A model can reduce complexity by decomposing a system into easy-to-understand elements. In software engineering, modeling starts with a description of a problem, an analysis, and then moves to the proposition of a solution to solve the problem, a design, along with an implementation and a deployment.

Models have driven engineering disciplines for centuries. One assumes the ancient Egyptians worked up a model before starting on a pyramid. Any building project includes plans and drawings that describe the building. This model also provides essential instructions to the workers implementing the design. For a model, the object under development may be a house, a machine, a new department within a company, a software program, or a series of programs. Diagrams specify how you want the finished product to look and to act. Plans for implementation, including estimation of cost and resource allocation, are made based on the information contained in the model. A poor model makes accurate estimation and planning impossible.

During the work of preparing a model, the model's designers must investigate the needs, preferences, structure, and design for the finished product. These needs and preferences, called *requirements*, include areas such as functionality, appearance, performance, and reliability. The designers create a model to communicate the different aspects of the product. Good designers will take advantage of the tools offered by UML to improve their design and get feedback from their clients. In order to allow this feedback loop between those who need the software, those who design the software, those who build the software, and those who deploy the software, a model is often broken down into a number of views, each of which describes a specific aspect of the product under construction.

A model not only represents a system or organization, promotes understanding, and enables simulation, but it is also a basic unit of development. It specifies a part of the function, structure, and the behavior of a system. A model is not directly observable by its designer or its user, but can be expressed in terms of diagrammatic notation, text, or a combination of both.

Many models for software engineering are displayed using a graphical language expressed by shapes, symbols, icons, and arrows, and supported by labels. In fact, models are usually described in a visual language, which means that most of the information in the models is expressed by graphical symbols and connections. The old saying that "a picture speaks a thousand words" is also relevant in modeling. Using visual descriptions helps communicate complex relationships; it also makes the practical modeling work easier.

NOTE Not everything is suitable for a visual description, some information in models is best expressed in ordinary text.

4 Chapter 1

In short, usable models are:

- **Accurate.** They must precisely and correctly describe the system they are representing.
- **Understandable.** They must be as simple as possible, but too simple to accurately fulfill their purpose, and must be easy to communicate.
- **Consistent.** Different views must not express things that are in conflict with each other.
- **Modifiable.** They must be easy to change and update.

Software Development, Methods, and Models

Many software projects fail. They can fail in many ways.

- Excessive cost
- Late delivery
- Absolute failure to meet the requirements and needs of customers

Some projects fail in terms of budget, schedule, and functionality, the three major factors to track in managing a software project. Such total failures appear far too often in software development. Effective modeling and sound management practices can help avoid such failures.

Technical advances, such as object-oriented programming, visual programming, and advanced development environments, have helped to increase productivity of coding, but have not solved these problems. Such technical advances have improved efficiency at the lowest level of system development, the programming level, not at the level of the analysis, design, or implementation.

One of the main problems with today's software development is that many projects start programming too soon and concentrate too much effort on writing code. Many managers lack an understanding of the software development process and become anxious when their programming team is not producing code. Programmers also tend to feel more secure when they're programming rather than when they are building abstract models of the system they are to create. In many cases, fundamental industry practices created many of these problems by measuring the performance of a programmer by lines of code rather than by solutions.

The use of modeling in software development is the difference between mature software engineering and hacking up code for a temporary solution. Systems are becoming much larger, integrating many different hardware and software components and machines, and distributed over complex architectures and platforms. The need to integrate extremely complex systems in

distributed environments requires that systems be designed carefully, not coded haphazardly.

Before UML, many approaches, called *methods*, were developed that attempted to inject engineering principles into the craft of software engineering. The most worthwhile engineering techniques are those that can be described both quantitatively and qualitatively, and used consistently and predictably. The method must achieve better results than using no method, must be able to be applied to many different problems, and must be able to be taught to others relatively easily.

The multitude of different methods, each with its own unique notation and tools, left many developers confused and unable to collaborate. The lack of a well-established notation upon which creators of many methods and tools could agree made it more difficult to learn how to use a good method. Furthermore, most of the early object-oriented methods were immature and best suited for small systems with limited functionality. They did not have the capability to scale up to the large systems that are now common. Indirectly, modeling can encourage a designer or software architect to adopt a more disciplined development process. Directly, models serve as a repository of knowledge.

The trends of the software industry still point to the need for creating models of the systems we intend to build. Visual programming is a technique by which programs are constructed by visually manipulating and connecting symbols; so it is that modeling and programming are highly integrated. Systems are becoming larger and are distributed over many computers through client/server architectures (with the Internet as the ultimate client/server architecture). The need to integrate complex systems in distributed environments requires that systems have some common models. Business engineering, where the business processes of a company are modeled and improved, requires computer systems that support these processes to implement the business models. Building models of systems before implementing them is becoming as normal and accepted in the software engineering community as it is in other engineering disciplines.

UML has helped improve modeling, making software engineering more mature as a discipline. Now, those who want to work as a software architect *must* know UML. The Java architecture exam, for example, assumes a knowledge of UML. Such progress, in a little over 60 months since initial adoption, represents a remarkable achievement. It is worth knowing how UML evolved, to understand the core elements of the language. Basically, three prominent methodologists in the information systems and technology industry—Grady Booch, James Rumbaugh, and Ivar Jacobson—cooperated in developing UML. UML combined the most effective diagramming practices applied by software developers over the past four decades. This combination put an end to competing styles that had clashed in the so-called “method wars.”

6 Chapter 1

The Method Wars

Antecedents of object-oriented (OO) modeling languages first appeared in the mid-1970s, and continued during the 1980s as programmers and project managers tried different approaches to analysis and design. Object orientation was initially spawned by the programming language Simula, but it didn't become popular until the late 1980s with the advent of programming languages such as C++ and Smalltalk. When object-oriented programming became a success, the need for methods to support software development followed. More than 50 methods and modeling languages appeared by 1994, and a growing problem became finding a single language to fulfill the needs of many different people and projects.

In the mid-1990s, the modeling languages started to incorporate the best features of previous languages, and few languages gained prominence over the field. These included OOSE (Object-Oriented Software Engineering), OMT-2 (Object Modeling Technique), and Booch'93. OOSE established the use of use cases, making it suited to business engineering and analysis. OMT-2 was strongly oriented toward analysis, and proved to be better at modeling data-intensive information systems. Booch'93 focused on the design and construction phases of projects, and proved especially applicable to engineering applications. Some prominent methods from this era and their contributions are listed as follows:

- **Coad/Yourdon.** This method, also known as OOA/OOD, was one of the first methods used for object-oriented analysis and design. It was simple and easy to learn, and worked well for introducing novices to the concepts and terminology of object-oriented technology. However, the notation and method could not scale to handle anything but very limited systems. It was not heavily used.
- **Booch.** Booch defined the notion that a system is analyzed as a number of views, where each view is described by a number of model diagrams. The method also contained a process by which the system was analyzed from both a macro- and microdevelopment view, and was based on a highly incremental and iterative process. Although it was very strong in architecture, there were those that argued that it didn't nail requirements issues.
- **OMT.** The Object Modeling Technique (OMT) was developed at General Electric by James Rumbaugh and is a straightforward process for performing tests based on a requirements specification. The system is described by a number of models, including the object model, the dynamic model, and the functional model, which complement each other to give the complete description of the system. The OMT method

also contained practical instructions for system design, taking into account concurrency and mapping to relational databases.

- **OOSE/Objectory.** The OOSE and Objectory methods both build on the same basic viewpoint formed by Ivar Jacobson. The OOSE method (I. Jacobson et al., 1992) was Jacobson's own vision of an object-oriented method. The Objectory method was used for building a number of systems from telecommunication systems for Ericsson to financial systems for Wall Street companies. Both methods were based on use cases, which are then implemented in all phases of the development. Objectory was also adapted for business engineering, where the ideas are used to model and improve business processes.
- **Fusion.** This method came from Hewlett-Packard (D. Coleman, 1994) and was called a second-generation method, as it was based on the experiences of many of the initial methods. Fusion enhanced a number of important previous ideas, including techniques for the specification of operations and interactions between objects.

Each of these methods had its own notation (the symbols used to represent object-oriented models), process (which stipulates the activities to perform in different parts of the development), and tools. This made the choice of method a very important decision, and often led to heated debates about which method was "the best," the "most advanced," and "the right" method to use in a specific project. As with any such discussion, there seldom was a good answer, because all the methods had their own strengths and weaknesses. Experienced developers often took one method as a base, and then borrowed good ideas and solutions from others. In practice, the differences between the methods were not significant, and as time passed and the methods developed, they began to integrate the best ideas. Still, the method confusion took up time, prohibited the development of a common notation, and prevented common tool support for visual modeling.

This was recognized by several of the method gurus, who began to seek ways to cooperate. A primary goal of UML was to put an end to the "method wars" within the object-oriented community. In October 1994, Grady Booch and Jim Rumbaugh of the Rational Software Corporation began work on unifying the Booch and OMT methods, resulting in a draft version of the Unified Method in October 1995. OOSE was incorporated into the Unified Method when Rational bought the Swedish company Objectory.

Earlier in 1995, Ivar Jacobson, the Chief Technology Officer of Objectory, and Richard Soley, Chief Technology Officer of the Object Management Group (OMG), decided to work together to achieve an independent, open standard in the methods marketplace. OMG hosted a meeting attended by most of the major methodologists at the time, with the one goal of achieving an industry

8 Chapter 1

standard modeling language. Jacobson joined Booch and Rumbaugh's unification work and the three became known as the "Three Amigos." The three then recognized that their work was more suited to creating a standard modeling language than a method and began work on the Unified Modeling Language. They submitted a number of preliminary versions of UML to the object-oriented community and received more ideas and suggestions to improve the language. Version 1.0 of the Unified Modeling Language was released in January 1997.

Even though the main parts of UML are based on the Booch, OMT, and OOSE methods, these designers also included concepts from other methods, for example, the work of David Harel on state charts adopted in the UML state machines, the parts of the Fusion notation for numbering operations included in collaboration diagrams, the responsibilities and collaborations from Wirfs-Brock, and the work of Gamma-Helm-Johnson-Vlissides on patterns and how to document them that inspired details of class diagrams. UML 2 has been additionally improved by the work of a number of designers with practical experience. The ability of UML to incorporate so many different thinkers into a framework of cooperation and improvement helps to explain the language's success.

The goals of UML, as stated by the designers, are:

- To model systems (and not just software) using object-oriented concepts
- To establish an explicit coupling to conceptual as well as executable artifacts
- To address the issues of scale inherent in complex, mission-critical systems
- To create a modeling language usable by both human beings and machines

Acceptance of UML

To establish UML, the designers realized that the language had to be made available to everyone. Therefore, the language is nonproprietary and open to all. Companies are free to use it with their own methods. Tool vendors are free to create tools for it, and authors are encouraged to write books about it. During 1996, a number of organizations joined to form the UML Partners consortium. These companies included the Digital Equipment Corporation, Hewlett-Packard, I-Logix, Intellicorp, IBM, ICON Computing, MCI Systemhouse, Microsoft, Oracle, Texas Instruments, Unisys, and Rational. These organizations saw UML as strategic to their businesses and contributed to the definition of UML. The companies also supported the proposal to adopt UML as an Object Management Group (OMG) standard. This made UML independent of any one

company's business plan, making it more likely to act as a standard throughout the industry. This approach brought peace and ended the wasteful method wars. For UML 2, this list of companies has expanded significantly. At this point, UML has far outgrown its initial founders, and it will outlast Rational as an independent company, now that IBM has purchased Rational.

The Object Management Group

The Object Management Group was founded in 1989 by 11 member companies, including 3Com Corporation; American Airlines; Canon, Inc.; Data General; Hewlett-Packard; Philips Telecommunications N.V.; Sun Microsystems; and Unisys Corporation, and now has over 800 members. The OMG is a not-for-profit corporation that is dedicated to establishing a component-based software industry through vendor independent specifications. Its goal is to produce industry guidelines and specifications in order to provide a common framework for application development.

The OMG does not produce software; it produces and then distributes specifications that are developed by an open community of contributors. Members can submit technology or concepts for review, contribute to open discussions, provide commentary, and review submissions from other members. Members of the organization then vote on contributions, adoptions, and versions of specifications in Task Forces and Technology Committees.

Unified Modeling Language Elements

With OMG support, the UML is now very well documented with a formal specification of the semantics of the language. Chapter 2 provides a high-level overview and is followed by chapters that look at these features more closely. At a high level, the details of UML are as follows:

- All UML diagrams describe object-oriented information.
- Class and object diagrams illustrate a system's static structure and the relationship between different objects.
- Interaction diagrams, state machines, and activity diagrams, show the dynamic behavior of objects, as well as messages between objects.
- Use-case and activity diagrams show system requirements and process workflows.
- The composite structure diagram shows the collaborating features as they combine at run time to implement a specific UML element.
- Deployment diagrams help with the deployment of the software in a specific environment.

10 Chapter 1

These features of UML enable business analysts and software architects to collaborate on a design, and specify, construct, and document applications in a standard way. Modelers can make sure that requirements are effectively captured, business problems are addressed, and the solutions are workable and practical.

Because no notation can cover every possible type of information, UML also supports the use of comments and defines their notation. One can also extend UML for specialized environments. UML 2 encourages a modeler to develop a Platform Specific Model tuned to the deployment environment. However, these language elements are not enough; they must be used within a sound process to take advantage of their main features to enhance communication.

Methods and Modeling Languages

There are important differences between a method and a modeling language. A *method*, also called a methodology, is an explicit way of structuring one's thinking and actions. It consists of a process, a standard vocabulary, and a set of rules and guidelines. It tells the user what to do, how to do it, when to do it, and why it is done. The method defines a set of activities that will accomplish the goals of the project, including the purpose of each specific activity, and what resulted from that activity. Examples include the Unified Process, Shlaer-Mellor, CRC (Class, Responsibilities, and Collaborators), and Extreme Programming. A mature and effective methodology for object-oriented development will encompass many different elements:

- A full life-cycle process
- A language defining concepts and models that are consistent, including notation for how ideas are presented
- Defined roles with prescribed responsibilities
- Rules and guidelines that define activities and how they are performed, including the work products and deliverables they produce
- Defined analysis, design, development, and test strategies

Methods contain models, and these models are used to describe something and to communicate the results of the use of a method. The main difference between a method and a modeling language is that the modeling language lacks a process or the instructions for what to do, how to do it, when to do it, and why it is done. Though UML standardizes models and has incorporated elements gathered from many different methods, the development approaches that use UML are as diverse as the environments in which it is used. It can

provide the underlying notation and language of the process. Still, different projects will emphasize different diagrams and extensions to UML. A project that demands accurate real-time execution will have different modeling needs than a project that provides reports at predefined intervals, for example. The real-time project will likely use extension mechanisms found in profiles for the advanced and specialized concepts needed in the project.

A model is expressed in a *modeling language*. A modeling language consists of notation—the symbols used in the models—and a set of rules directing how to use it. The rules are syntactic, semantic, and pragmatic.

- Syntax tells us how the symbols should look and how the symbols in the modeling language should be combined. The syntax can be compared to words in natural language; it is important to know how to spell them correctly and how to put different words together to form a sentence.
- Semantic rules explain what each symbol means and how it should be interpreted, either by itself or in the context of other symbols.
- The pragmatic rules define the intentions of the symbols through which the purpose of a model is achieved and becomes understandable for others. This corresponds in natural language to the rules for constructing sentences that are clear and understandable. To use a modeling language well, it is necessary to learn all of these rules. Fortunately, UML is a lot easier to comprehend than a natural language. Though naturally, even when the language is mastered there is no guarantee that the models produced will be effective.

Object-Oriented Software Development

UML comes from an object-oriented background. One answer to the question “What is the UML?” could be the following: “It is an object-oriented modeling language for modern software systems.” As an object-oriented modeling language, all the elements and diagrams in UML are based on the object-oriented paradigm. Software development that is object-oriented depicts the real world and solves problems through the interpretation of “objects”, digitally mimicking and representing the tangible elements of a system. The object-oriented approach considers a system as a dynamic entity comprising components, which can really only be defined with respect to one another. What a system is and does can be described only in terms of its components and how they interact.

12 Chapter 1

Concepts of Object Orientation

The principal concepts of object orientation are as follows.

- Object orientation is a technology for producing models that reflect a domain, such as a business domain or a machine domain, in a natural way, using the terminology of the domain.
- Object-oriented software development has five underlying concepts: objects, messages, classes, inheritance, and polymorphism. Software objects have a state, a behavior, and an identity. The behavior can depend upon the state, and the state may be modified by the behavior. Messages that provide the communication between the objects of a system, and between systems themselves, have five main categories: constructors, destructors, selectors, modifiers, and iterators.
- Every object is a real-world “instance” of a class, which is a type of template used to define the characteristics of an object. Each object has a name, attributes, and operations. Classes are said to have an association if the objects they instantiate are linked or related.
- Object-oriented models, when constructed correctly, are easy to communicate, change, expand, validate, and verify.
- When done correctly, systems built using object-oriented technology are flexible in response to change, have well-defined architectures, and provide the opportunity to create and implement reusable components. The requirements of the system are traceable to the code of the system.
- Object-oriented models are conveniently implemented in software using object-oriented programming languages. Using programming languages that are not object-oriented to implement object-oriented systems is not recommended. However, it’s important to realize that object-oriented software engineering is much more than just a couple of mechanisms in a programming language.
- Object orientation is not just a theory, but a well-proven technology used in a large number of projects and for building many different types of systems. The field still lacks standardization to show the way to an industrialization of object technology. The work provided by OMG strives to achieve such standardization.
- Object orientation requires a method that integrates a development process and a modeling language with suitable construction techniques and tools.

In UML, objects and classes are modeled by object and class diagrams. The interaction between objects is represented by communication or sequence diagrams. Communication diagrams more effectively display the physical layout

of objects within a system, while sequence diagrams show the interactions between objects across time. Classes may be placed into hierarchies by means of generalization and specialization relationships, which are usually implemented using inheritance. UML perfectly complements the object-oriented development philosophy, as it was designed with object orientation as its foundation.

Business Engineering

The use of object-oriented modeling for the purpose of business engineering has generated a lot of interest. Object-oriented models have proven to be an excellent method for modeling the business processes in a company. A business process provides some value to the customer of the business (or perhaps the customer's customer). When a company uses techniques such as Business Process Reengineering (BPR) or Total Quality Management (TQM), the processes are analyzed, improved, and implemented in the company. Using an object-oriented modeling language to model and document the processes also makes it easier to use these models when building the information systems in the company.

CROSS-REFERENCE For more on business engineering, see the book **Business Modeling with UML: Business Patterns at Work** by Magnus Penker and Hans-Erik Eriksson (Wiley 2000).

UML 2 offers a number of new features designed to make it easier to move from business process modeling to software development. Activity diagrams and state machines include more features for accurate system description and flow control. In addition, a first-class extension mechanism with profiles makes it easier to tailor UML in a standard way to support specific domains.

Disciplines of System Development

There are numerous methods and processes applied to develop systems. Disregarding particular phases, milestones, or artifacts created, you can consider that a set of disciplines will always be applied. These disciplines are:

- Requirements
- Analysis
- Design
- Implementation
- Test

14 Chapter 1

No matter when the disciplines are applied in an end-to-end process, no matter how much or little emphasis each discipline might receive in a process, one must work on the system from each of these perspectives.

CROSS-REFERENCE Chapter 10 describes a process for using UML. In the next sections, we will just introduce how UML supports each of these disciplines.

Requirements

UML has use cases to capture the requirements of the customer. Through use-case modeling, the external actors that have interest in the system are modeled along with the functionality they require from the system (the use cases). The actors and use cases are modeled with relationships and have communication associations with each other or are broken down into hierarchies. The actors and use cases are described in a UML use-case diagram. Each use case is described in text, and that specifies the requirements of the customer, what he or she expects of the system, without considering how the functionality will be implemented. Extra detail of the system can be described using behavioral diagrams, such as state machines and activity diagrams, plus structural diagrams, such as high-level class diagrams showing relationships among business entities. The requirements discipline can also be exercised with business processes, not only for software systems.

Analysis

Analysis is concerned with the primary abstractions (classes and objects) and mechanisms that are present in the problem domain. The classes that model these are identified, along with their relationships to each other, and described in a UML class diagram. Collaborations between classes necessary to perform the use cases are also described, via any of the dynamic models in UML. In the analysis, only classes that are in the problem domain (real-world concepts) are modeled—not technical classes that define details and solutions in the software system, such as classes for user interface, databases, communication, concurrency, and so on.

Design

In design, the result of the analysis is expanded into a technical solution. New classes are added to provide the technical infrastructure: the user interface, database handling to store objects in a database, communication with other systems, interfacing to devices in the system, and others. The domain problem classes from the analysis are “embedded” into this technical infrastructure,

making it possible to change both the problem domain and the infrastructure. The design results in detailed specifications for the implementation activities.

Implementation

In implementation, the classes from the design phase are converted to actual code in an object-oriented programming language (using a procedural language is *not* recommended). Depending on the capability of the language used, this can be either a difficult or an easy task.

When creating analysis and design models in UML, it is best to avoid trying to mentally translate the models into code. The models are a means to understand and structure a system; thus, jumping to early conclusions about the code can be counterproductive to creating simple and correct models. The programming is a separate activity, during which the models are converted into code.

Test

A system is normally exercised via unit tests, integration tests, system tests, and acceptance tests.

- The *unit tests* are of individual classes or a group of classes and are typically performed by the programmer.
- The *integration test* integrates components and classes in order to verify that they cooperate as specified.
- The *system test* views the system as a “black box” and validates the end functionality of the system expected by an end user.
- The *acceptance test* is conducted by the customer to verify that the system satisfies the requirements. It is similar to the system test.

The different test teams use different UML diagrams as the basis for their work: unit tests use class diagrams and class specifications, integration tests typically use component diagrams and collaboration diagrams, and the system tests implement use-case and activity diagrams to verify that the system behaves as initially defined in these diagrams.

Relevant Changes in UML 2

The initial versions of UML blended a number of different modeling methods into a standard, freeing software engineers to focus on software and allowing tool vendors to provide automated support for software development. More importantly, UML improved software engineering by supporting effective management of information and communication about system elements. The model took center stage. Such success created an appetite for more. Software

16 Chapter 1

engineers now want additional features for describing behavior, portraying alternate paths, supporting component development, and showing the complex distributed deployments that mark enterprise applications. Tool vendors want an unambiguous specification so that all tools claiming to implement UML actually do. Further, the worlds of automated support tools and software modeling continue to grow closer. Some look for an architecture driven by models with enough technical sophistication to generate more code automatically and with enough flexibility to evolve with continuous change. UML 2 seeks to deliver these additional features, and more.

The OMG set out well-defined goals for UML 2 in a set of requirements that reflected issues found by users of UML 1.x. UML 2 supports the initiative for a model-driven architecture by providing the stable technical infrastructure that allows for additional automation of the software development process. From the requirements specification, four goals have emerged as prominent drivers in the evolution of UML:

- Make the language for modeling software entities more executable.
- Provide more robust mechanisms for modeling workflow and actions.
- Create a standard for communication between different tools.
- Reconcile UML within a standard modeling framework.

UML 2 has four separate specifications to address these main goals. Of these, the superstructure provides the main specification for UML elements and diagrams. Most of the information relevant to this book comes from the superstructure specification. That said, the other specifications help make UML easier to implement with support tools. They are of more interest to tool vendors and those with a passion for modeling, but merit brief mention here to explain the increasing complexity of UML behind the notation definition. Indeed, the OMG wanted to clearly separate the definition of the UML notation from the metamodel semantics.

The infrastructure specification provides a clearer mapping between UML and a general framework for all modeling from the OMG, the Meta Object Facility (MOF). This will allow the production of additional types of modeling languages that will all use the same basic elements. Additionally, this makes it easier for UML to rely on Extensible Markup Language (XML) Metadata Interchange (XMI) to define the rules for validating models and moving them between tools. While UML 1.x enhanced communication between people with visual diagram, UML 2 will enhance automated communication with the ability to express UML models in XML. If UML 1.0 attempted to end the method wars among those in object-oriented design, UML 2 seeks to be part of a system that allows modeling of all information technology systems and communication within a complex enterprise, a very lofty goal.

The Object Constraint Language (OCL) defines instructions for specifying constraints and actions in a model. UML diagrams, however, do not require the use of OCL. The Diagram Interchange Model is a brief statement that specifies what you would anticipate: rules for diagram interchange.

UML 2 seeks to build on UML 1.x, not replace elements for no reason. OMG specified that changes should have as little impact as possible to users of current languages. Any change requires a clear statement of how this helps UML attain the specified goals. We can summarize the requirement's specification goals for the superstructure as follows.

- Support component-based development, including full definition for plug substitutability.
- Allow fuller modeling of the component execution context, especially needed with the proliferation of middleware and Web applications.
- Provide standard profiles or extensions for important languages.
- Support automated implementation of common interaction patterns.
- Enhance UML for runtime architecture, including support for the modeling of internal structure, especially communication paths, as well as the description of the dynamic behavior of that internal structure.
- Improve state machines, with clearer specifications for links to behavior and rules for specialization.
- Improve activity graphs with better control and data flow features.
- Support the composition of interaction mechanisms; define sequences of interactions, alternate interactions, and parallel execution.

Summary

UML 2 has a number of changes to diagrams that reflect behavior and deployment. New features, such as the port, explicitly support component-based development. Indeed, each of the changes in UML 2 in some way furthers these stated high-level goals discussed in the preceding section. Readers will find a more detailed summary of these changes at the end of each modeling chapter. The next chapter, Chapter 2, provides a high-level overview of the main elements in UML as background for the more detailed chapters on the specific types of modeling. The emphasis remains on understanding how to use the basic tools of UML, giving the reader an effective toolkit for software success.

