CHAPTER 2

Introduction to the UML

Why should I use the UML? What can it contribute to my software development effort?

To effectively utilize any technology, we must understand what it can positively contribute to the software development effort. Therefore, before adopting the Unified Modeling Language (UML), we should answer the following questions:

Why should I use the UML?

What can it contribute to my software development effort?

We begin by formally introducing the UML and defining its intended role in software development, as stated by the UML's original creators. Then we take a brief tour through the history of the UML. To finish, we explore some of the challenges encountered during software development today, and how the UML can help reduce these challenges to manageable tasks.

2.0 The UML Defined

The Unified Modeling Language User Guide states the following:

The UML is a language for

- Visualizing
- Specifying
- Constructing
- Documenting

the artifacts of a software-intensive system. [BOOCH99]

Visual Programming

Don't let this analogy to Java confuse you. The UML is not a visual programming language. The UML enables us to create an expression of our application in the form of a model. We can't express many implementation-specific details in the UML. The UML does map nicely to Java, and this mapping is discussed in Chapter 3. Helping you to model, and ultimately design, more effectively is a goal of this book.

Exploring this definition further enables us to realize the true intent of the UML. As stated previously, the UML is a language. As such, it has an associated set of rules that must be adhered to if we are to create well-formed, precise models, just as Java has a set of syntax rules that must be adhered to in order to create compilable applications. Because it's only a language, and not a methodology, the UML doesn't dictate how we use the models we create nor the order in which we create them. This misconception was one of the greatest ones associated with the UML early in its life. The UML is not a software process.

To further differentiate the UML from a software process, let's consider an analogy. Because Java is a language, the only stipulation to create an application that will compile is to create an application structure that is syntactically correct. On the other hand, Java doesn't dictate how we relate the internal elements of our system, nor does it impose any design restrictions. Therefore, when we create a Java application, we not only must write code that is syntactically correct, but we also must design the application. As discussed in Chapter 1, various principles and patterns can help guide us in creating a more resilient architecture, which is similar to the UML. The only stipulation the UML imposes upon

A History Lesson?

History of any sort is extremely important, and the software industry is no exception. It's important to understand the value of a standardized modeling language, as well as the historical impact other languages have had on the UML. If we ignore this aspect of the UML, we are doomed to repeat the same mistakes in the UML that were made with other modeling languages.

us is to create diagrams that are syntactically correct. How we organize our diagrams and what artifacts to produce is the role of a software process. Different processes will guide us through the creation of a different set of artifacts, which is a major advantage of the UML. Because it's process independent, we can integrate the most desirable pieces into our existing software process.

In Table 2.1, we provide descriptions for the previously listed bulleted points (from *The Unified Modeling Language User Guide*).

Table 2.1 Characteristics of the UML

Term	Description
Visualizing	We're all familiar with the developer who spends time at work struggling with a complex challenge, only to doodle out a solution on a napkin at that evening's happy hour. In fact, most of us have probably done something similar. There's something special about creating a visual representation. It makes it easier to understand and work through the problem. The visual and formal nature of the UML takes this doodling a few steps further. Because the UML is a formal language, with its own set of rules, it enables other developers familiar with the language to more easily interpret our drawings.
Specifying	We must communicate our software system using some common, precise, and unambiguous communication mechanism. Again, the formal nature of the UML facilitates this specification quite nicely.
Constructing	We already know that the UML is a formal language with its own set of syntactical rules. Because of this formality, we can create tools that interpret our models. Obviously, because these tools can interpret our models, they can map the elements to a programming language, such as Java. Many tools do support this forward-engineering concept. In fact, this is one of

(continues)

Table 2.1 Characteristics of the UML *(continued)*

Term	Description
	the advantages of using a formal modeling tool. It enforces the syntactical rules of the UML.
Documenting	The models we create are just one of the artifacts produced throughout the development lifecycle. Using the UML in a consistent fashion produces a set of documentation that can serve as a blueprint of our system.

2.1 Origin

The history of the UML is rich, and while we won't go into detail, it's important to understand its deep roots. The UML's original designers, Grady Booch, Ivar Jacobson, and James Rumbaugh, realized the software industry was saturated with a variety of object-oriented modeling languages and methodologies. Instead of creating another new language, they decided to incorporate the best of the existing languages and methodologies. In October 1995, version 0.8 of the Unified Method was released. After some minor revision, including a change in its name, version 0.9 of the Unified Modeling Language was introduced in June 1996. About this time, the software industry began to take interest in the unification effort, and after some additional revisions, version 1.0 of the UML was submitted to the Object Management Group (OMG) for industry standardization in January 1997. At this point, a Revision Task Force was formed by OMG to serve as the UML's governing body. On November 14, 1997, UML 1.1 was adopted by OMG as the industry standard modeling language. Since that time, the UML has gone through some minor revisions, mostly pertaining to use cases, and in June 1999, UML 1.3 was adopted, which is the version of the UML described in this book.

In Figure 2.1, we see some of the languages and methods that contributed to the UML.

A complete definition of the UML is contained within the "OMG Unified Modeling Language Specification" and can be found on the OMG Web site at www.omg.org. This specification is a robust set of documentation, including but not limited to, the following:

- **UML Summary:** A brief history and introduction to the UML
- **UML Semantics:** The various semantics and rules that compose the UML and contribute to the creation of well-formed models

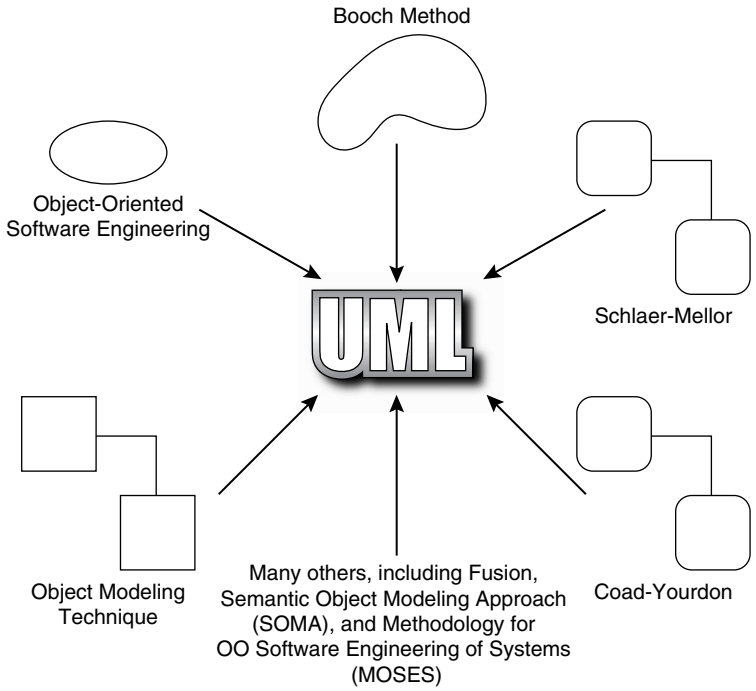

Figure 2.1 Foundations of the UML

- **UML Notation Guide:** Expression of the graphic syntax composing the UML
- **UML Extensions:** Standard UML extension for the unified process and business modeling

While this specification is a complete set of documentation and provides insight to the thought and careful planning that went toward the creation of the UML, it's not exciting reading. It's meant primarily for tools vendors, and a book such as this one provides a more valuable explanation of how the UML can be used most effectively.

2.2 Role of Modeling

To really understand why we would want to adopt any technology, it's first important to understand what problem the technology is trying to solve. Realistically, it does no good to adopt a new technology if it doesn't solve some problem or overcome some challenge. Adopting the UML is no different.

2.2.1 The Challenge

A number of challenges are associated with software development. However, none presents more obstacles than that of changing requirements. In Chapter 1, we learned that it costs twice as much to maintain a system as it costs to develop it, and that during development, only 15 percent of our time is actually devoted to programming. These statistics are proof that a system is never really complete until it has been removed from production and is no longer functioning. Until that point, time and money are spent adding new features, fixing bugs, and even improving its internal structure. This entire maintenance effort is typically the result of growing and changing requirements. If requirements were static, once these requirements were realized in a functioning system, the system would be complete and require no more maintenance, ultimately incurring no additional cost. So the real challenge in software development is dealing with these changing requirements. Therefore, we need a technology that helps us deal effectively with changing requirements.

Before we explore how we can more effectively manage a dynamic set of requirements, let's first review why changing requirements are so frustrating to deal with. In Chapter 1, we briefly discussed the architecture paradox. This paradox recognizes that a system has the competing goals of survivability and evolvability. In order for a system to survive, it must meet requirements, but as requirements change, the system must evolve. Failure to evolve to these changes means the system loses its survivability. Therefore, our system's archi-

Maintenance versus Reuse

The object-oriented paradigm carries with it the promise of reuse. However, reuse may not be the true benefit of object orientation. If the main obstacle in software development is accommodating changing requirements, we should be in search of a technology that allows our systems to adapt more flexibly to change. Fortunately, object orientation carries with it this benefit at least as much as, if not more than, the benefit of reuse. In Chapter 1, we briefly discussed this issue. We offer a gentle reminder here because of the emphasis that we'll place on architecting resilient systems that are easier to maintain versus attempting to architect systems that realize a high degree of reuse, at least initially. Stressing resiliency will continue to be a key aspect of our discussions as we progress through the remainder of this book.

tecture directly impacts the ability of a system to evolve gracefully. This is the same conclusion reached in Chapter 1, and it now becomes clear that the following holds true:

> *The cost to maintain a system is directly related to the resiliency of the system's architecture.*

Therefore, the challenge in developing software is to develop software that is well architected.

2.2.2 Complexity of Architecture

Developing software is an inherently complex process. In *The Mythical Man-Month,* Frederick Brooks cites two complexities associated with software development. He categorizes them as *essential complexities* and *accidental complexities* [BROOKS95]. Essential complexities are those difficulties that are inherent in the nature of software, whereas accidental complexities are difficulties that attend the production of software but can be eliminated.

There have been many attempts to eliminate accidental complexity. Examples of accidental complexity include a mismatch of tools or paradigms, a lack of formal methodologies or models, and awkward programming languages. Years ago, we wrote software using assembly language, which was a time-consuming process. Programming languages have evolved since then, and today we are able to focus more on our problem domain and spend less time struggling with various technical issues such as performance and hardware constraints. With each of these advances, certain accidental complexities are reduced. These advances, however, introduce new accidental complexities, such as those associated with mapping an object-oriented system to a relational database for persistent storage, known as the *object/relational impedance mismatch*.

The more interesting of these complexities, essential complexity, cannot be eliminated and is inherent in the nature of software. It's fairly common to see comparisons between software engineering and other engineering disciplines. The electronics industry has a set of canned components that can be used in many different types of electronics, which carries with it obvious advantages. Why do we not have similar components in the software industry? Only recently has this concept of component-based development been gaining steam. The problem is that certain properties of software development aren't found in many other engineering disciplines. First, software is invisible. Obvious contradictions aren't easily caught because software has no geometric representation as electronic components do. No dictating physics are associated with software. When designing an electric circuit, or building a skyscraper,

mathematical limitations are placed on what we can do. Software typically doesn't have this same set of restraints. Because of its invisibility, software often is seen as infinitely changeable. It would be absurd to consider changing the structure of a 100-story high-rise once it's 90 percent complete. This doesn't always hold true with software. Therefore, we need to create software systems that are infinitely malleable. Put simply, people and businesses change, and demand that the software do so as well. Therefore, the fashioning of these complex, yet malleable, structures in a programming language can be a difficult and challenging task. This task is the complex essence of software engineering, and when we're able to reduce this essential complexity, we'll have taken a small step forward.

2.2.3 The Remedy

Because we can't eliminate essential complexity, we can hope only to reduce it. The UML, and visual modeling, is a tool that can be used to help reduce essential complexity. By creating visual representations of our software system, it's easier to identify contradictions that may have been previously overlooked. Thus, visual modeling enables us to create systems that are flexible enough to achieve a higher degree of resiliency. While it isn't necessarily true that we wouldn't be able to create a flexible design without modeling, creating a visual representation of a system can help all individuals involved to more fully understand that system. Consequently, each of these individuals will see a common system with little ambiguity. This common perspective helps increase communication among our team members and also helps each team member more fully understand the system. If we better understand something, we can work with it more effectively.

Though we may never be able to accommodate every single change scenario, working toward this goal gives our system the extra degree of flexibility that it might need to survive longer and grow into the future. In addition to visual modeling, associated principles, patterns, and other proven best practices just might give us what we need to turn adequate designs into great designs.

2.3 Benefits

At this point, we should have a much clearer understanding of what the UML is and what it is not. We've also probably begun to formulate our own theories as to why we would want to model, and what some of the benefits of doing so are. In this section, we discuss the obvious, and the not-so-obvious, benefits of the UML.

Modeling as an Activity

While we often discuss modeling as if it were an activity, we must keep in mind that we shouldn't treat modeling as a formal stage in the software development lifecycle. Nor should we be led to believe at any point in time that creating a model adds value to our system. The only value that can be added to our system comes in the form of source code that is error free and functionally correct. In fact, while we use the term *modeling*, we model primarily to produce higher-quality designs. However, modeling does contribute to the creation of a successful system by helping us manage complexity and solve difficult challenges. Consequently, modeling is an omnipresent activity performed throughout all stages of the software development lifecycle. We model because it helps us analyze problems and design more effective solutions that can be communicated effectively.

One of the greatest benefits of using the UML is that it facilitates a common, precise, unambiguous, and unified communication mechanism. While other modeling languages may claim to provide the first of these three benefits, only the UML can lay claim to unification. It's the industry standard modeling language. For those of us who continue to take advantage of the benefits provided from other traditional languages, a transition to the UML provides the added benefit of unification.

Modeling also enables us to create simplified representations of our systems. By modeling at different levels of abstraction, we can communicate a different intent by creating different models. We have to be cautious here. If we model at too high a level of abstraction, our model loses the value of its original intent. Regardless, the ability to communicate bits and pieces of the overall system contributes to a more manageable understanding of our system. We'll continue to explore this concept as we progress throughout this book.

As we've seen, developing software is an inherently complex process. Just like the developer who doodles on a napkin to gain further insight into the problem at hand, modeling serves as an excellent problem-solving mechanism. Reaching resolution on our most complex problems in visual form prior to coding contributes to a more resilient system. Similar to problem solving, modeling can help us validate our theories. When confronted with a problem, it's highly likely that we will consider multiple solutions. Modeling can validate that we've chosen the solution that is most viable considering the context of the problem.

While we discuss the UML throughout this book in the context of developing Java applications, the UML is language independent. In Chapter 3, we define the mappings from the UML to Java. Mappings such as these exist for any other object-oriented programming language.

2.4 Conclusion

Modeling has taken a strong foothold in software development and is undoubtedly here to stay. The UML was created to provide the software development community with a standard, unified modeling language. The UML has a rich history and was created by some of the most well-respected methodologists in the software industry.

While modeling is not the silver bullet for software development, it has numerous benefits, which include enhanced communication, validation of our design theories, and helping to simplify an inherently difficult process. By taking advantage of these benefits, we can produce systems that will survive as the demands of our users and business change. In some regards, all projects utilize a certain degree of modeling. The choice to use a formal language such as the UML should be given serious consideration. The result can be a much more resilient design, which can reduce the cost associated with the maintenance effort.