Object-oriented methodology is a set of methods, models, and rules for developing systems. Modeling is the process of describing an existing or proposed system. It can be used during any phase of the software life cycle. A model is an abstraction of a phenomenon for the purpose of understanding it. Since a model excludes unnecessary details; it is easier to manipulate than the real object. Modeling provides a means for communicating ideas in an easy to understand and unambiguous form while also accommodating a system's complexity. In this part we will look at Object-Oriented Methodologies in Chapter 4, and Unified Modeling Language in Chapter 5.
Anyone who observes software development cannot but be impressed by its repetitive nature. Over and over again, programmers weave a number of basic patterns: sorting, searching, reading, writing, comparing, traversing, allocating, synchronizing, and so forth. Experienced programmers know the feeling of déjà vu so characteristic of their trade [18].

**Chapter Objectives**

You should be able to define and understand
- Object-oriented methodologies.
  - The Rumbaugh et al. OMT.
  - The Booch methodology.
  - Jacobson’s methodologies.
- Patterns.
- Frameworks.
- Unified approach (UA).

This chapter studies some of the well-known object-oriented methodologies and emerging techniques such as use of patterns and frameworks. The chapter concludes with the unified approach (UA), which is a combination of the best practices and methodologies described in this chapter. The UA is a conceptual model used in this book for studying object-oriented concepts and system development.

### 4.1 INTRODUCTION: TOWARD UNIFICATION—TOO MANY METHODOLOGIES

In the 1980s, many methodologists were wondering how analysis and design methods and processes would fit into an object-oriented world. Object-oriented methods suddenly had become very popular, and it was apparent that the techniques to help people execute good analysis and design were just as important as the object-oriented concept itself.
To get a feel for object-oriented methodologies, let us look at some of the methods developed in the 1980s and 1990s. This list by no means is complete [14].

- 1991. Jim Rumbaugh led a team at the research labs of General Electric to develop the object modeling technique (OMT) [19].

These methodologies and many other forms of notational language provided system designers and architects many choices but created a very split, competitive, and confusing environment. Most of the methods were very similar but contained a number of often annoying minor differences, and each had a group of practitioners that liked its ideas. The same basic concepts appeared in very different notations, which caused confusion among the users [14].

The trend in object-oriented methodologies, sometimes called second-generation object-oriented methods, has been toward combining the best aspects of the most popular methods instead of coming out with new methodologies, which was the tendency in first-generation object-oriented methods. In the next section, to give you a taste of object-oriented methodologies, we will look at some of the most popular ones.

### 4.2 Survey of Some of the Object-Oriented Methodologies

Many methodologies are available to choose from for system development. Each methodology is based on modeling the business problem and implementing the application in an object-oriented fashion; the differences lie primarily in the documentation of information and modeling notations and language. An application can be implemented in many ways to meet the same requirements and provide the same functionality. The largest noticeable differences will be in the trade-offs and detailed design decisions made. Two people using the same methodology may produce application designs that look radically different. This does not necessarily mean that one is right and one is wrong, just that they are different. In the following sections, we look at the methodologies and their modeling notations developed by Rumbaugh et al., Booch, and Jacobson which are the origins of the Unified Modeling Language (UML).

Each method has its strengths. The Rumbaugh et al. method is well-suited for
describing the object model or the static structure of the system. The Jacobson et al. method is good for producing user-driven analysis models. The Booch method produces detailed object-oriented design models.

4.3 RUMBAUGH ET AL.'S OBJECT MODELING TECHNIQUE

The object modeling technique (OMT) presented by Jim Rumbaugh and his coworkers describes a method for the analysis, design, and implementation of a system using an object-oriented technique. OMT is a fast, intuitive approach for identifying and modeling all the objects making up a system. Details such as class attributes, method, inheritance, and association also can be expressed easily. The dynamic behavior of objects within a system can be described using the OMT dynamic model. This model lets you specify detailed state transitions and their descriptions within a system. Finally, a process description and consumer-producer relationships can be expressed using OMT's functional model. OMT consists of four phases, which can be performed iteratively:

1. **Analysis.** The results are objects and dynamic and functional models.
2. **System design.** The results are a structure of the basic architecture of the system along with high-level strategy decisions.
3. **Object design.** This phase produces a design document, consisting of detailed objects static, dynamic, and functional models.
4. **Implementation.** This activity produces reusable, extendible, and robust code.

OMT separates modeling into three different parts:

1. An **object model**, presented by the object model and the data dictionary.
2. A **dynamic model**, presented by the state diagrams and event flow diagrams.

4.3.1 The Object Model

The object model describes the structure of objects in a system: their identity, relationships to other objects, attributes, and operations. The object model is represented graphically with an object diagram (see Figure 4–1). The object diagram contains classes interconnected by association lines. Each class represents a set of individual objects. The association lines establish relationships among the classes. Each association line represents a set of links from the objects of one class to the objects of another class.

4.3.2 The OMT Dynamic Model

OMT provides a detailed and comprehensive dynamic model, in addition to letting you depict states, transitions, events, and actions. The OMT state transition diagram is a network of states and events (see Figure 4–2). Each state receives one or more events, at which time it makes the transition to the next state. The next state depends on the current state as well as the events.
4.3.3 The OMT Functional Model

The OMT data flow diagram (DFD) shows the flow of data between different processes in a business. An OMT DFD provides a simple and intuitive method for describing business processes without focusing on the details of computer systems [3].

Data flow diagrams use four primary symbols:

1. The process is any function being performed; for example, verify Password or PIN in the ATM system (see Figure 4–3).
2. The data flow shows the direction of data element movement; for example, PIN code.
3. The data store is a location where data are stored; for example, account is a data store in the ATM example.
4. An external entity is a source or destination of a data element; for example, the ATM card reader.

Overall, the Rumbaugh et al. OMT methodology provides one of the strongest tool sets for the analysis and design of object-oriented systems.
No account has been selected

Nothing is selected → Account has been selected

Selected checking or savings account → Select checking account

Select transaction type (withdraw, deposit, transfer) → Enter the amount

Enter the amount → Confirmation

**FIGURE 4-2**
State transition diagram for the bank application user interface. The round boxes represent states and the arrows represent transitions.

### 4.4 THE BOOCH METHODOLOGY

The Booch methodology is a widely used object-oriented method that helps you design your system using the object paradigm. It covers the analysis and design phases of an object-oriented system. Booch sometimes is criticized for his large set of symbols. Even though Booch defines a lot of symbols to document almost every design decision, if you work with his method, you will notice that you never use all these symbols and diagrams. You start with class and object diagrams (see Figures 4-4 and 4-5) in the analysis phase and refine these diagrams in various steps. Only when you are ready to generate code, do you add design symbols—and this is where the Booch method shines, you can document your object-oriented code. The Booch method consists of the following diagrams:

- Class diagrams
- Object diagrams
- State transition diagrams
- Module diagrams
- Process diagrams
- Interaction diagrams
The Booch methodology prescribes a macro development process and a micro development process.

4.4.1 The Macro Development Process

The macro process serves as a controlling framework for the micro process and can take weeks or even months. The primary concern of the macro process is technical management of the system. Such management is interested less in the actual object-oriented design than in how well the project corresponds to the requirements set for it and whether it is produced on time. In the macro process, the traditional phases of analysis and design to a large extent are preserved [4].

The macro development process consists of the following steps:

1. Conceptualization. During conceptualization, you establish the core requirements of the system. You establish a set of goals and develop a prototype to prove the concept.

2. Analysis and development of the model. In this step, you use the class diagram to describe the roles and responsibilities objects are to carry out in performing
the desired behavior of the system. Then, you use the object diagram to describe the desired behavior of the system in terms of scenarios or, alternatively, use the interaction diagram to describe behavior of the system in terms of scenarios.

3. **Design or create the system architecture.** In the design phase, you use the class diagram to decide what classes exist and how they relate to each other. Next, you use the object diagram to decide what mechanisms are used to regulate how objects collaborate. Then, you use the module diagram to map out where each class and object should be declared. Finally, you use the process diagram to determine to which processor to allocate a process. Also, determine the schedules for multiple processes on each relevant processor.

4. **Evolution or implementation.** Successively refine the system through many iterations. Produce a stream of software implementations (or executable releases), each of which is a refinement of the prior one.

5. **Maintenance.** Make localized changes to the system to add new requirements and eliminate bugs.

### 4.4.2 The Micro Development Process

Each macro development process has its own micro development processes. The micro process is a description of the day-to-day activities by a single or small group of software developers, which could look blurry to an outside viewer, since the analysis and design phases are not clearly defined.
FIGURE 4.5
An alarm class state transition diagram with Booch notation. This diagram can capture the state of a class based on a stimulus. For example, a stimulus causes the class to perform some processing, followed by a transition to another state. In this case, the alarm silenced state can be changed to alarm sounding state and vice versa.

The micro development process consists of the following steps:

1. Identify classes and objects.
2. Identify class and object semantics.
3. Identify class and object relationships.
4. Identify class and object interfaces and implementation.

4.5 THE JACOBSON ET AL. METHODOLOGIES

The Jacobson et al. methodologies (e.g., object-oriented Business Engineering (OOBE), object-oriented Software Engineering (OOSE), and Objectory) cover the entire life cycle and stress traceability between the different phases, both forward and backward. This traceability enables reuse of analysis and design work, possibly much bigger factors in the reduction of development time than reuse of code. At the heart of their methodologies is the use-case concept, which evolved with Objectory (Object Factory for Software Development).

4.5.1 Use Cases

Use cases are scenarios for understanding system requirements. A use case is an interaction between users and a system. The use-case model captures the goal of
the user and the responsibility of the system to its users (see Figure 4–6). In the requirements analysis, the use cases are described as one of the following [4]:

- Nonformal text with no clear flow of events.
- Text, easy to read but with a clear flow of events to follow (this is a recommended style).
- Formal style using pseudo code.

The use case description must contain

- How and when the use case begins and ends.
- The interaction between the use case and its actors, including when the interaction occurs and what is exchanged.
- How and when the use case will need data stored in the system or will store data in the system.
- Exceptions to the flow of events.
- How and when concepts of the problem domain are handled.

Every single use case should describe one main flow of events. An exceptional or additional flow of events could be added. The exceptional use case extends another use case to include the additional one. The use-case model employs extends and uses relationships. The extends relationship is used when you have one use case that is similar to another use case but does a bit more. In essence, it extends the functionality of the original use case (like a subclass). The uses relationship reuses common behavior in different use cases.

Use cases could be viewed as concrete or abstract. An abstract use case is not complete and has no actors that initiate it but is used by another use case. This inheritance could be used in several levels. Abstract use cases also are the ones that have uses or extends relationships.

**FIGURE 4–6**

Some uses of a library. As you can see, these are external views of the library system from an actor such as a member. The simpler the use case, the more effective it will be. It is unwise to capture all of the details right at the start; you can do that later.
4.5.2 Object-Oriented Software Engineering: Objectory

Object-oriented software engineering (OOSE), also called Objectory, is a method of object-oriented development with the specific aim to fit the development of large, real-time systems. The development process, called use-case driven development, stresses that use cases are involved in several phases of the development (see Figure 4–7), including analysis, design, validation, and testing. The use-case scenario begins with a user of the system initiating a sequence of interrelated events.

The system development method based on OOSE, Objectory, is a disciplined process for the industrialized development of software, based on a use-case driven design. It is an approach to object-oriented analysis and design that centers on understanding the ways in which a system actually is used. By organizing the analysis and design models around sequences of user interaction and actual usage scenarios, the method produces systems that are both more usable and more robust, adapting more easily to changing usage. Jacobson et al.’s Objectory has been developed and applied to numerous application areas and embodied in the CASE tool systems.

Objectory is built around several different models:

- **Use case-model.** The use-case model defines the outside (actors) and inside (use case) of the system’s behavior.
- **Domain object model.** The objects of the “real” world are mapped into the domain object model.
- **Analysis object model.** The analysis object model presents how the source code (implementation) should be carried out and written.
- **Implementation model.** The implementation model represents the implementation of the system.
- **Test model.** The test model constitutes the test plans, specifications, and reports.

![Diagram of the Objectory process](image)

**FIGURE 4–7**
The use-case model is considered in every model and phase.
The maintenance of each model is specified in its associated process. A process is created when the first development project starts and is terminated when the developed system is taken out of service.

4.5.3 Object-Oriented Business Engineering

Object-oriented business engineering (OOBE) is object modeling at the enterprise level. Use cases again are the central vehicle for modeling, providing traceability throughout the software engineering processes.

- Analysis phase. The analysis phase defines the system to be built in terms of the problem-domain object model, the requirements model, and the analysis model. The analysis process should not take into account the actual implementation environment. This reduces complexity and promotes maintainability over the life of the system, since the description of the system will be independent of hardware and software requirements. Jacobson [16] does not dwell on the development of the problem-domain object model, but rather he refers the developer to Coad and Yourdon's [11] or Booch's [6] discussion of the topic, who suggest that the customer draw a picture of his view of the system to promote discussions. In their view, a full development of the domain model will not localize changes and therefore will not result in the most "robust and extensible structure." This model should be developed just enough to form a base of understanding for the requirements model. The analysis process is iterative but the requirements and analysis models should be stable before moving on to subsequent models. Jacobson et al. suggest that prototyping with a tool might be useful during this phase to help specify user interfaces.

- Design and implementation phases. The implementation environment must be identified for the design model. This includes factors such as Database Management System (DBMS), distribution of process, constraints due to the programming language, available component libraries, and incorporation of graphical user interface tools. It may be possible to identify the implementation environment concurrently with analysis. The analysis objects are translated into design objects that fit the current implementation environment.

- Testing phase. Finally, Jacobson describes several testing levels and techniques. The levels include unit testing, integration testing, and system testing.

4.6 PATTERNS

An emerging idea in systems development is that the process can be improved significantly if a system can be analyzed, designed, and built from prefabricated and predefined system components. One of the first things that any science or engineering discipline must have is a vocabulary for expressing its concepts and a language for relating them to each other. Therefore, we need a body of literature to help software developers resolve commonly encountered, difficult problems and a vocabulary for communicating insight and experience about these problems and their solutions. The primary focus here is not so much on technology as on creating a culture to document and support sound engineering architecture and design [5].
In this section, we look at the concept of patterns; and in the next section, we look at another emerging method, frameworks.

The use of design patterns originates in the work done by a building architect named Christopher Alexander during the late 1970s. Alexander wrote two books, *A Pattern Language* [1] and *A Timeless Way of Building* [2], that, in addition to giving examples, described his rationale for documenting patterns. Alexander’s articulation on pattern work was soon employed by object-oriented thinkers looking for ways to describe commonly occurring design solutions and programming paradigms. As described in their seminal work in cataloging program design concepts, Gamma, Helm, Johnson, and Vlissides [15] say that the design pattern identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design. Furthermore, it identifies the participating classes and instances, their roles and collaborations, and the distribution of responsibilities. It describes when it applies, whether it can be applied in view of other design constraints, and the consequences and trade-offs of its use.

Another book that helped popularize the use of patterns is *Pattern-Oriented Software Architecture—A System* by Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal [10]. Currently, patterns are being used largely for software architecture and design and, more recently, for organizations, specification models, and many other aspects of software development processes.

The main idea behind using patterns is to provide documentation to help categorize and communicate about solutions to recurring problems. The pattern has a name to facilitate discussion and the information it represents. A definition that more closely reflects its use within the patterns community is by Riehle and Züllighoven [20]:

A pattern is [an] instructive information that captures the essential structure and insight of a successful family of proven solutions to a recurring problem that arises within a certain context and system of forces.

The documentation of a pattern, in essence, provides the contexts under which it is suitable and the constraints and forces that may affect a solution or its consequences. Communication about patterns is enabled by a vocabulary that describes the pattern and its related components such as name, context, motivation, and solution. By classifying these components and their nature (such as the structural or behavioral nature of the solution), we can categorize patterns.

A pattern involves a general description of a solution to a recurring problem bundle with various goals and constraints. But a pattern does more than just identify a solution; it also explains why the solution is needed. For better or for worse, however, the meteoric rise in popularity of software patterns frequently has caused them to be overhyped. Patterns have achieved buzzword status: It is immensely popular to use the word pattern to garner an audience. However, not every solution, algorithm, best practice, maxim, or heuristic constitutes a pattern (one or more key pattern ingredients may be absent). Even if something appears to have
all the requisite pattern components, it should not be considered a pattern until it has been verified to be a recurring phenomenon (preferably found in at least three existing systems; this often is called the *rule of three*). A “pattern in waiting,” which is not yet known to recur, sometimes is called a *proto-pattern*. Many also feel it is inappropriate to decisively call something a *pattern* until it has undergone some degree of peer scrutiny or review [5]. Coplien [12] explains that a good pattern will do the following:

* It *solves a problem*. Patterns capture solutions, not just abstract principles or strategies.
* It *is a proven concept*. Patterns capture solutions with a track record, not theories or speculation.
* The *solution is not obvious*. The best patterns generate a solution to a problem indirectly—a necessary approach for the most difficult problems of design.
* It *describes a relationship*. Patterns do not just describe modules, but describe deeper system structures and mechanisms.
* The *pattern has a significant human component*. All software serves human comfort or quality of life; the best patterns explicitly appeal to aesthetics and utility.

The majority of the initial patterns developed focus on design problems and still design patterns represent most solutions. However, more recent patterns encompass all aspects of software engineering, including development organization, the software development process, project planning, requirements engineering, and software configuration management.

### 4.6.1 Generative and Nongenerative Patterns

Generative patterns are patterns that not only describe a recurring problem, they can tell us how to generate something and can be observed in the resulting system architectures they helped shape. Nongenerative patterns are static and passive: They describe recurring phenomena without necessarily saying how to reproduce them. We should strive to document generative patterns because they not only show us the characteristics of good systems, they teach us how to build them. Alexander explains that the most useful patterns are generative:

> These patterns in our minds are, more or less, mental images of the patterns in the world: they are abstract representations of the very morphological rules which define the patterns in the world. However, in one respect they are very different. The patterns in the world merely exist. But the same patterns in our minds are dynamic. They have force. They are generative. They tell us what to do; they tell us how we shall, or may, generate them; and they tell us too, that under certain circumstances, we must create them. Each pattern is a rule which describes what you have to do to generate the entity which it defines. [2, pp. 181–82]

Alexander wants patterns, and especially pattern languages, to be capable of generating whole, living structures. Part of the desire to create architectures that
emulate life lies in the unique ability of living things to evolve and adapt to their ever-changing environments (not only for the sake of individual survival but also for survival of the species). Alexander wants to impart these same qualities into his architecture. Similarly, in software, good software architecture is all about being adaptable and resilient to change. So another aspect of generativity is about striving to create “living” architecture capable of dynamically adapting to fulfill changing needs and demands.

The successive application of several patterns, each encapsulating its own problem and forces, unfolds a larger solution, which emerges indirectly as a result of the smaller solutions. It is the generation of such emergent behavior that appears to be what is meant by generativity. In this fashion, a pattern language should guide its users to generate whole architectures that possess the quality. This particular aspect of Alexander’s paradigm seems a bit too mystical for some people’s tastes [5].

4.6.2 Patterns Template

Every pattern must be expressed “in the form of a rule [template] which establishes a relationship between a context, a system of forces which arises in that context, and a configuration, which allows these forces to resolve themselves in that context” [2].

Currently, several different pattern templates have been defined that eventually will represent a pattern. Despite this, it is generally agreed that a pattern should contain certain essential components. The following essential components should be clearly recognizable on reading a pattern [5]:

- **Name.** A meaningful name. This allows us to use a single word or short phrase to refer to the pattern and the knowledge and structure it describes. Good pattern names form a vocabulary for discussing conceptual abstractions. Sometimes, a pattern may have more than one commonly used or recognizable name in the literature. In this case, it is common practice to document these nicknames or synonyms under the heading of aliases or also known as. Some pattern forms also provide a classification of the pattern in addition to its name.

- **Problem.** A statement of the problem that describes its intent: the goals and objectives it wants to reach within the given context and forces. Often the forces oppose these objectives as well as each other.

- **Context.** The *preconditions* under which the problem and its solution seem to recur and for which the solution is desirable. This tells us the pattern’s applicability. It can be thought of as the initial configuration of the system before the pattern is applied to it.

- **Forces.** A description of the relevant *forces* and constraints and how they interact or conflict with one another and with the goals we wish to achieve (perhaps with some indication of their priorities). A concrete scenario that serves as the *motivation* for the pattern frequently is employed (see also Examples). Forces reveal the intricacies of a problem and define the kinds of *trade-offs* that must be considered in the presence of the tension or dissonance they create. A good pattern description should fully encapsulate all the forces that have an impact on it.
• Solution. Static relationships and dynamic rules describing how to realize the desired outcome. This often is equivalent to giving instructions that describe how to construct the necessary products. The description may encompass pictures, diagrams, and prose that identify the pattern’s structure, its participants, and their collaborations, to show how the problem is solved. The solution should describe not only the static structure but also dynamic behavior. The static structure tells us the form and organization of the pattern, but often the behavioral dynamics is what makes the pattern “come alive.” The description of the pattern’s solution may indicate guidelines to keep in mind (as well as pitfalls to avoid) when attempting a concrete implementation of the solution. Sometimes, possible variants or specializations of the solution are described as well.

• Examples. One or more sample applications of the pattern that illustrate a specific initial context; how the pattern is applied to and transforms that context; and the resulting context left in its wake. Examples help the reader understand the pattern’s use and applicability. Visual examples and analogies often can be very useful. An example may be supplemented by a sample implementation to show one way the solution might be realized. Easy-to-comprehend examples from known systems usually are preferred.

• Resulting context. The state or configuration of the system after the pattern has been applied, including the consequences (both good and bad) of applying the pattern, and other problems and patterns that may arise from the new context. It describes the postconditions and side effects of the pattern. This is sometimes called a resolution of forces because it describes which forces have been resolved, which ones remain unresolved, and which patterns may now be applicable. Documenting the resulting context produced by one pattern helps you correlate it with the initial context of other patterns (a single pattern often is just one step toward accomplishing some larger task or project).

• Rationale. A justifying explanation of steps or rules in the pattern and also of the pattern as a whole in terms of how and why it resolves its forces in a particular way to be in alignment with desired goals, principles, and philosophies. It explains how the forces and constraints are orchestrated in concert to achieve a resonant harmony. This tells us how the pattern actually works, why it works, and why it is “good.” The solution component of a pattern may describe the outwardly visible structure and behavior of the pattern, but the rationale is what provides insight into the deep structures and key mechanisms going on beneath the surface of the system.

• Related patterns. The static and dynamic relationships between this pattern and others within the same pattern language or system. Related patterns often share common forces. They also frequently have an initial or resulting context that is compatible with the resulting or initial context of another pattern. Such patterns might be predecessor patterns whose application leads to this pattern, successor patterns whose application follows from this pattern, alternative patterns that describe a different solution to the same problem but under different forces and constraints, and codependent patterns that may (or must) be applied simultaneously with this pattern.

• Known uses. The known occurrences of the pattern and its application within existing systems. This helps validate a pattern by verifying that it indeed is a
proven solution to a recurring problem. Known uses of the pattern often can
serve as instructional examples (see also Examples).

Although it is not strictly required, good patterns often begin with an abstract
that provides a short summary or overview. This gives readers a clear picture of
the pattern and quickly informs them of its relevance to any problems they may
wish to solve (sometimes such a description is called a thumbnail sketch of the pat-
tern, or a pattern thumbnail). A pattern should identify its target audience and
make clear what it assumes of the reader.

4.6.3 Antipatterns
A pattern represents a “best practice,” whereas an antipattern represents “worst
practice” or a “lesson learned.” Antipatterns come in two varieties:

• Those describing a bad solution to a problem that resulted in a bad situation.
• Those describing how to get out of a bad situation and how to proceed from there
to a good solution.

Antipatterns are valuable because often it is just as important to see and under-
stand bad solutions as to see and understand good ones. Coplien explains that

The study of anti-patterns is an important research activity. The presence of “good” pat-
terns in a successful system is not enough; you also must show that those patterns are
absent in unsuccessful systems. Likewise, it is useful to show the presence of certain
patterns (anti-patterns) in unsuccessful systems, and their absence in successful sys-
tems. [12]

4.6.4 Capturing Patterns
Writing good patterns is very difficult, explains Appleton [5]. Patterns should pro-
vide not only facts (like a reference manual or users' guide) but also tell a story
that captures the experience they are trying to convey. A pattern should help its
users comprehend existing systems, customize systems to fit user needs, and con-
struct new systems. The process of looking for patterns to document is called pattern mining (or sometimes reverse architec
ting). An interesting initiative started within the software community is to share experience with patterns and develop
an ever-growing repository of patterns. People can contribute new solutions,
lessons learned (or antipatterns), and more examples within a variety of contexts.

How do you know a pattern when you come across one? The answer is you do
not always know. You may jot down the beginning of some things you think are
patterns, but it may turn out that these are not patterns at all, or they are only pieces
of patterns, simply good principles, or general rules that may form part of the rationa
e for a particular pattern. It is important to remember that a solution in which
no forces are present is not a pattern [5].

These guidelines are summarized from Buschmann et al. [10]:

• Focus on practicability. Patterns should describe proven solutions to recurring
problems rather than the latest scientific results.
• Aggressive disregard of originality. Pattern writers do not need to be the original inventor or discoverer of the solutions that they document.
• Nonanonymous review. Pattern submissions are shepherded rather than reviewed. The shepherd contacts the pattern author(s) and discusses with him or her how the patterns might be clarified or improved on.
• Writers’ workshops instead of presentations. Rather than being presented by the individual authors, the patterns are discussed in writers’ workshops, open forums where all attending seek to improve the patterns presented by discussing what they like about them and the areas in which they are lacking.
• Careful editing. The pattern authors should have the opportunity to incorporate all the comments and insights during the shepherding and writers’ workshops before presenting the patterns in their finished form.

4.7 FRAMEWORKS

Frameworks are a way of delivering application development patterns to support best practice sharing during application development—not just within one company, but across many companies—through an emerging framework market. This is not an entirely new idea. Consider the following [22]:

• An experienced programmer almost never codes a new program from scratch—she’ll use macros, copy libraries, and template-like code fragments from earlier programs to make a start on a new one. Work on the new program begins by filling in new domain-specific code inside the older structures.
• A seasoned business consultant who has worked on many consulting projects performing data modeling almost never builds a new data model from scratch—he’ll have a selection of model fragments that have been developed over time to help new modeling projects hit the ground running. New domain-specific terms will be substituted for those in his library models.

A framework is a way of presenting a generic solution to a problem that can be applied to all levels in a development [22]. However, design and software frameworks are the most popular. A definition of an object-oriented software framework is given by Gamma et al. [15]:

A framework is a set of cooperating classes that make up a reusable design for a specific class of software. A framework provides architectural guidance by partitioning the design into abstract classes and defining their responsibilities and collaborations. A developer customizes a framework to a particular application by subclassing and composing instances of framework classes. The framework captures the design decisions that are common to its application domain. Frameworks thus emphasize design reuse over code reuse, though a framework will usually include concrete subclasses you can put to work immediately.

A single framework typically encompasses several design patterns. In fact, a framework can be viewed as the implementation of a system of design patterns.
Even though they are related in this manner, it is important to recognize that frameworks and design patterns are two distinctly separate beasts: A framework is executable software, whereas design patterns represent knowledge and experience about software. In this respect, frameworks are of a physical nature, while patterns are of a logical nature: Frameworks are the physical realization of one or more software pattern solutions; patterns are the instructions for how to implement those solutions [5].

Gamma et al. describe the major differences between design patterns and frameworks as follows [15]:

- **Design patterns are more abstract than frameworks.** Frameworks can be embodied in code, but only examples of patterns can be embodied in code. A strength of frameworks is that they can be written down in programming languages and not only studied but executed and reused directly. In contrast, design patterns have to be implemented each time they are used. Design patterns also explain the intent, trade-offs, and consequences of a design.
- **Design patterns are smaller architectural elements than frameworks.** A typical framework contains several design patterns but the reverse is never true.
- **Design patterns are less specialized than frameworks.** Frameworks always have a particular application domain. In contrast, design patterns can be used in nearly any kind of application. While more specialized design patterns are certainly possible, even these would not dictate an application architecture.

### 4.8 THE UNIFIED APPROACH

The approach promoted in this book is based on the best practices that have proven successful in system development and, more specifically, the work done by Booch, Rumbaugh, and Jacobson in their attempt to unify their modeling efforts. The unified approach (UA) (see Figure 1-1) establishes a unifying and unitary framework around their works by utilizing the unified modeling language (UML) to describe, model, and document the software development process. The idea behind the UA is not to introduce yet another methodology. The main motivation here is to combine the best practices, processes, methodologies, and guidelines along with UML notations and diagrams for better understanding object-oriented concepts and system development.

The unified approach to software development revolves around (but is not limited to) the following processes and concepts (see Figure 4–8). The processes are:

- Use-case driven development
- Object-oriented analysis
- Object-oriented design
- Incremental development and prototyping
- Continuous testing
The methods and technology employed include

- Unified modeling language used for modeling.
- Layered approach.
- Repository for object-oriented system development patterns and frameworks.
- Component-based development (Although, UA promote component-based development, the treatment of the subject is beyond the scope of the book.)

The UA allows iterative development by allowing you to go back and forth between the design and the modeling or analysis phases. It makes backtracking very easy and departs from the linear waterfall process, which allows no form of backtracking.

4.8.1 Object-Oriented Analysis

Analysis is the process of extracting the needs of a system and what the system must do to satisfy the users' requirements. The goal of object-oriented analysis is to first understand the domain of the problem and the system's responsibilities by understanding how the users use or will use the system. This is accomplished by constructing several models of the system. These models concentrate on describing what the system does rather than how it does it. Separating the behavior of a system from the way it is implemented requires viewing the system from the user's
perspective rather than that of the machine. OOA Process consists of the following steps:

1. Identify the Actors.
2. Develop a simple business process model using UML Activity diagram.
3. Develop the Use Case.
4. Develop interaction diagrams.
5. Identify classes.

4.8.2 Object-Oriented Design

Booch [9] provides the most comprehensive object-oriented design method. Ironically, since it is so comprehensive, the method can be somewhat imposing to learn and especially tricky to figure out where to start. Rumbaugh et al.’s and Jacobson et al.’s high-level models provide good avenues for getting started. UA combines these by utilizing Jacobson et al.’s analysis and interaction diagrams, Booch’s object diagrams, and Rumbaugh et al.’s domain models. Furthermore, by following Jacobson et al.’s life cycle model, we can produce designs that are traceable across requirements, analysis, design, coding, and testing. OOD Process consists of:

- Designing classes, their attributes, methods, associations, structures and protocols, apply design axioms
- Design the Access Layer
- Design and prototype User interface
- User Satisfaction and Usability Tests based on the Usage/Use Cases
- Iterate and refine the design

4.8.3 Iterative Development and Continuous Testing

You must iterate and reiterate until, eventually, you are satisfied with the system. Since testing often uncovers design weaknesses or at least provides additional information you will want to use, repeat the entire process, taking what you have learned and reworking your design or moving on to reprototyping and retesting. Continue this refining cycle through the development process until you are satisfied with the results. During this iterative process, your prototypes will be incrementally transformed into the actual application. The UA encourages the integration of testing plans from day 1 of the project. Usage scenarios can become test scenarios; therefore, use cases will drive the usability testing. Usability testing is the process in which the functionality of software is measured. Chapter 13 will cover usability testing.

4.8.4 Modeling Based on the Unified Modeling Language

The unified modeling language was developed by the joint efforts of the leading object technologists Grady Booch, Ivar Jacobson, and James Rumbaugh with contributions from many others. The UML merges the best of the notations used by the three most popular analysis and design methodologies: Booch’s methodology, Jacobson et al.’s use case, and Rumbaugh et al.’s object modeling technique. The
UML is becoming the universal language for modeling systems; it is intended to be used to express models of many different kinds and purposes, just as a programming language or a natural language can be used in many different ways. The UML has become the standard notation for object-oriented modeling systems. It is an evolving notation that still is under development. The UA uses the UML to describe and model the analysis and design phases of system development (UML notations will be covered in Chapter 5).

4.8.5 The UA Proposed Repository

In modern businesses, best practice sharing is a way to ensure that solutions to process and organization problems in one part of the business are communicated to other parts where similar problems occur. Best practice sharing eliminates duplication of problem solving. For many companies, best practice sharing is institutionalized as part of their constant goal of quality improvement. Best practice sharing must be applied to application development if quality and productivity are to be added to component reuse benefits. Such sharing extends the idea of software reusability to include all phases of software development such as analysis, design, and testing [22].

The idea promoted here is to create a repository that allows the maximum reuse of previous experience and previously defined objects, patterns, frameworks, and user interfaces in an easily accessible manner with a completely available and easily utilized format. As we saw previously, central to the discussion on developing this best practice sharing is the concept of a pattern. Everything from the original user request to maintenance of the project as it goes to production should be kept in the repository. The advantage of repositories is that, if your organization has done projects in the past, objects in the repositories from those projects might be useful. You can select any piece from a repository—from the definition of one data element, to a diagram, all its symbols, and all their dependent definitions, to entries—for reuse.

The UA's underlying assumption is that, if we design and develop applications based on previous experience, creating additional applications will require no more than assembling components from the library. Additionally, applying lessons learned from past developmental mistakes to future projects will increase the quality of the product and reduce the cost and development time. Some basic capability is available in most object-oriented environments, such as Microsoft repository, VisualAge, PowerBuilder, Visual C++, and Delphi. These repositories contain all objects that have been previously defined and can be reused for putting together a new software system for a new application. If a new requirement surfaces, new objects will be designed and stored in the main repository for future use.

The same arguments can be made about patterns and frameworks. Specifications of the software components, describing the behavior of the component and how it should be used, are registered in the repository for future reuse by teams of developers.

The repository should be accessible to many people. Furthermore, it should be relatively easy to search the repository for classes based on their attributes, methods,
or other characteristics. For example, application developers could select prebuilt components from the central component repository that match their business needs and assemble these components into a single application, customizing where needed.

Tools to fully support a comprehensive repository are not accessible yet, but this will change quickly and, in the near future, we will see more readily available tools to capture all phases of software development into a repository for use and reuse.

4.8.6 The Layered Approach to Software Development

Most systems developed with today's CASE tools or client-server application development environments tend to lean toward what is known as two-layered architecture: interface and data (see Figure 4–9).

In a two-layered system, user interface screens are tied to the data through routines that sit directly behind the screens; for example, a routine that executes when you click on a button. With every interface you create, you must re-create the business logic needed to run the screen. The routines required to access the data must exist within every screen. Any change to the business logic must be accomplished in every screen that deals with that portion of the business. This approach results in objects that are very specialized and cannot be reused easily in other projects.

A better approach to systems architecture is one that isolates the functions of the interface from the functions of the business. This approach also isolates the business from the details of the data access (see Figure 4–10). Using the three-
FIGURE 4-11
Business objects represent tangible elements of the application. They should be completely independent of how they are represented to the user or how they are physically stored.

A layered approach, you are able to create objects that represent tangible elements of your business yet are completely independent of how they are represented to the user (through an interface) or how they are physically stored (in a database). The three-layered approach consists of a view or user interface layer, a business layer, and an access layer (see Figure 4–11).

4.8.6.1 The Business Layer  The business layer contains all the objects that represent the business (both data and behavior). This is where the real objects such as Order, Customer, Line item, Inventory, and Invoice exist. Most modern object-oriented analysis and design methodologies are generated toward identifying these kinds of objects.

The responsibilities of the business layer are very straightforward: Model the objects of the business and how they interact to accomplish the business processes. When creating the business layer, however, it is important to keep in mind a couple of things. These objects should not be responsible for the following:

- **Displaying details.** Business objects should have no special knowledge of how they are being displayed and by whom. They are designed to be independent of any particular interface, so the details of how to display an object should exist in the interface (view) layer of the object displaying it.
- **Data access details.** Business objects also should have no special knowledge of “where they come from.” It does not matter to the business model whether the data are stored and retrieved via SQL or file I/O. The business objects need to know only to whom to talk about being stored or retrieved. The business objects are modeled during the object-oriented analysis.

A business model captures the static and dynamic relationships among a collection of business objects. Static relationships include object associations and aggregations. For example, a customer could have more than one account or an order could be aggregated from one or more line items. Dynamic relationships show how the business objects interact to perform tasks. For example, an order interacts with inventory to determine product availability. An individual business object can appear in different business models. Business models also incorporate control objects that
direct their processes. The business objects are identified during the object-oriented analysis. Use cases can provide a wonderful tool to capture business objects.

4.8.6.2 The User Interface (View) Layer  The user interface layer consists of objects with which the user interacts as well as the objects needed to manage or control the interface. The user interface layer also is called the view layer.

This layer typically is responsible for two major aspects of the applications:

- **Responding to user interaction.** The user interface layer objects must be designed to translate actions by the user, such as clicking on a button or selecting from a menu, into an appropriate response. That response may be to open or close another interface or to send a message down into the business layer to start some business process; remember, the business logic does not exist here, just the knowledge of which message to send to which business object.

- **Displaying business objects.** This layer must paint the best possible picture of the business objects for the user. In one interface, this may mean entry fields and list boxes to display an order and its items. In another, it may be a graph of the total price of a customer's orders.

The user interface layer's objects are identified during the object-oriented design phase. However, the requirement for a user interface or how a user will use the system is the responsibility of object-oriented analysis. Use cases can provide a very useful tool for understanding user interface requirements.

4.8.6.3 The Access Layer  The access layer contains objects that know how to communicate with the place where the data actually reside, whether it be a relational database, mainframe, Internet, or file. Regardless of where the data actually reside, the access layer has two major responsibilities:

- **Translate request.** The access layer must be able to translate any data-related requests from the business layer into the appropriate protocol for data access. (For example, if Customer number 55552 needs to be retrieved, the access layer must be able to create the correct SQL statement and execute it.)

- **Translate results.** The access layer also must be able to translate the data retrieved back into the appropriate business objects and pass those objects back up into the business layer.

Access objects are identified during object-oriented design.

4.9 SUMMARY

In this chapter, we looked at current trends in object-oriented methodologies, sometimes known as second-generation object-oriented methods, which have been toward combining the best aspects of today's most popular methods.

Each method has its strengths. Rumbaugh et al. have a strong method for producing object models (sometimes known as domain object models). Jacobson et al. have a strong method for producing user-driven requirement and object-oriented
analysis models. Booch has a strong method for producing detailed object-oriented design models.

Each method has a weakness, too. While Rumbaugh et al.'s OMT has strong methods for modeling the problem domain, OMT models cannot fully express the requirements. Jacobson et al. de-emphasize object modeling and, although they cover a fairly wide range of the life cycle, they do not treat object-oriented design to the same level as Booch, who focuses almost entirely on design, not analysis.

Booch and Rumbaugh et al. are object centered in their approaches and focus more on figuring out what are the objects of a system, how are they related, and how do they collaborate with each other. Jacobson et al. are more user centered, in that everything in their approach derives from use cases or usage scenarios.

The main idea behind a pattern is the documentation to help categorize, communicate about, and locate solutions to recurring problems. Frameworks are a way of delivering application development patterns to support best practice sharing during application development. A single framework typically encompasses several design patterns. In fact, a framework can be viewed as the implementation of a system of design patterns. Writing good patterns is very difficult, since it should not only provide facts but also tell a story that captures the experience the pattern is trying to convey.

The UA is an attempt to combine the best practices, processes, and guidelines along with UML notations and diagrams for better understanding object-oriented concepts and object-oriented system development. The UA consists of the following processes:

- Use-case driven development
- Object-oriented analysis
- Object-oriented design
- Incremental development and prototyping
- Continuous testing

Furthermore, it utilizes the methods and technologies such as, unified modeling language, layered approach and promotes repository for all phases of software development.

**KEY TERMS**

- Abstract use case (p. 69)
- Framework (p. 77)
- Pattern (p. 72)
- Pattern mining (p. 76)
- Pattern thumbnail (p. 76)
- Proto-pattern (p. 73)

**REVIEW QUESTIONS**

1. What is a method?
2. What is a methodology?
3. What is process?
4. Describe the difference between a method and a process?
5. What are the phases of OMT? Briefly describe each phase.
6. What is an object model? What are the other OMT models?
7. What is the main advantage of DFD?
8. What is the strength of OMT?
9. Name five Booch diagrams.
10. Briefly describe the Booch system development processes.
11. What is the strength of Booch methodology?
12. What is Objectory?
13. Name the models in Objectory.
14. What is a use case?
15. What is the reason for having abstract use cases?
16. What are some of the ways that use cases can be described?
17. What must a use case contain?
18. What is the strength of the Jacobson et al. methodology?
19. Describe the difference between patterns and frameworks.

PROBLEMS
1. Consult the World Wide Web or a library to obtain an article on a real-world application that has incorporated a use-case model. Write a summary report of your finding.
2. Consult the Web or the library to obtain an article on future trends in object-oriented software development. Write a summary of your findings.
3. The best way to learn how to recognize and document useful patterns is by learning from others who have done it well. Consult the Web or the library to obtain articles that describe patterns (do not choose just one) and try to see if you can recognize all the necessary pattern components and desirable qualities mentioned in this chapter. When you see one that appeals to you, ask yourself why it is good. If you see one you dislike, try to figure out exactly what about the pattern leaves you unsatisfied. Read as much as you can, and try to learn from the masters. For an excellent source on patterns, obtain the paper written by Appleton [5]; it provides numerous resources for learning more about patterns. Examine how it is meaningful to you and how it will help you accomplish future goals. Write a summary of your findings.
4. Imagine that you are a methodologist and would like to develop your own object-oriented methodology by combining many different object-oriented methodologies. Use the materials in this chapter, Chapter 3, and the Web to create your own object-oriented methodology system development life cycle.
5. Consult the Web or the library to obtain an article that compares different methodologies.
6. This chapter did not cover all the methodologies listed earlier. Consult the Web or your friendly school library and write a short paper on one of the following methodologies:
   a. Shlaer and Mellor’s concept of the recursive design approach.
   b. Beck and Cunningham’s Classes, Responsibilities, and Collaborators (CRC) cards.
   c. Wirfs-Brock et al.’s responsibility-driven design.
   d. Coad and Yourdon’s lightweight and prototype-oriented approach to methods.

REFERENCES


10. Buschmann, Frank; Meunier, Regine; Rohnert, Hans; Sommerlad, Peter; and Stal, Michael. Pattern-Oriented Software Architecture—A System of Patterns. Chichester, UK: Wiley and Sons Ltd., 1996.


Chapter Objectives

You should be able to define and understand

- Modeling and its benefit.
- Different types of models.
- Basics of Unified Modeling Language (UML) and its modeling diagrams.
  - UML Class diagram.
  - UML Use case diagram.
  - UML Sequence diagram.
  - UML Collaboration diagram.
  - UML Statechart diagram.
  - UML Activity diagram.
  - UML Component diagram.
  - UML Deployment diagram.

5.1 INTRODUCTION

A model is an abstract representation of a system, constructed to understand the system prior to building or modifying it. The term system is used here in a broad sense to include any process or structure. For example, the organizational structure of a corporation, health services, computer software, instruction of any sort (including computers), the national economy, and so forth all would be termed systems.
Efraim Turban [9] describes a model as a simplified representation of reality. A model is simplified because reality is too complex or large and much of the complexity actually is irrelevant to the problem we are trying to describe or solve. A model provides a means for conceptualization and communication of ideas in a precise and unambiguous form. The characteristics of simplification and representation are difficult to achieve in the real world, since they frequently contradict each other. Thus, modeling enables us to cope with the complexity of a system.

Most modeling techniques used for analysis and design involve graphic languages. These graphic languages are sets of symbols. The symbols are used according to certain rules of the methodology for communicating the complex relationships of information more clearly than descriptive text. The main goal of most CASE tools is to aid us in using these graphic languages, along with their associated methodologies.

Modeling frequently is used during many of the phases of the software life cycle, such as analysis, design, and implementation. For example, Objectory is built around several different models:

* **Use-case model.** The use-case model defines the outside (actors) and inside (use case) of the system’s behavior.
* **Domain object model.** Objects of the “real” world are mapped into the domain object model.
* **Analysis object model.** The analysis object model presents how the source code (i.e., the implementation) should be carried out and written.
* **Implementation model.** The implementation model represents the implementation of the system.
* **Test model.** The test model constitutes the test plans, specifications, and reports.

Modeling, like any other object-oriented development, is an iterative process. As the model progresses from analysis to implementation, more detail is added, but it remains essentially the same.

In this chapter, we look at unified modeling language (UML) notations and diagrams. The main idea here is to gain exposure to the UML syntax, semantics, and modeling constructs. Many new concepts will be introduced here from a modeling standpoint. We apply these concepts in system analysis and design contexts in later chapters.

### 5.2 Static and Dynamic Models

Models can represent static or dynamic situations. Each representation has different implications for how the knowledge about the model might be organized and represented [7].

#### 5.2.1 Static Model

A **static model** can be viewed as a snapshot of a system’s parameters at rest or at a specific point in time. Static models are needed to represent the structural or
static aspect of a system. For example, a customer could have more than one account or an order could be aggregated from one or more line items. Static models assume stability and an absence of change in data over time. The unified modeling language class diagram is an example of a static model.

5.2.2 Dynamic Model

A dynamic model, in contrast to a static model, can be viewed as a collection of procedures or behaviors that, taken together, reflect the behavior of a system over time. Dynamic relationships show how the business objects interact to perform tasks. For example, an order interacts with inventory to determine product availability.

A system can be described by first developing its static model, which is the structure of its objects and their relationships to each other frozen in time, a baseline. Then, we can examine changes to the objects and their relationships over time. Dynamic modeling is most useful during the design and implementation phases of the system development. The UML interaction diagrams and activity models are examples of UML dynamic models.

5.3 WHY MODELING?

Building a model for a software system prior to its construction is as essential as having a blueprint for building a large building. Good models are essential for communication among project teams. As the complexity of systems increases, so does the importance of good modeling techniques. Many other factors add to a project’s success, but having a rigorous modeling language is essential. A modeling language must include [2]

- Model elements—fundamental modeling concepts and semantics.
- Notation—visual rendering of model elements.
- Guidelines—expression of usage within the trade.

In the face of increasingly complex systems, visualization and modeling become essential, since we cannot comprehend any such system in its entirety. The use of visual notation to represent or model a problem can provide us several benefits relating to clarity, familiarity, maintenance, and simplification.

- Clarity. We are much better at picking out errors and omissions from a graphical or visual representation than from listings of code or tables of numbers. We very easily can understand the system being modeled because visual examination of the whole is possible.
- Familiarity. The representation form for the model may turn out to be similar to the way in which the information actually is represented and used by the employees currently working in the problem domain. We, too, may find it more comfortable to work with this type of representation.
- Maintenance. Visual notation can improve the maintainability of a system. The visual identification of locations to be changed and the visual confirmation of
those changes will reduce errors. Thus, you can make changes faster, and fewer errors are likely to be introduced in the process of making those changes.

- Simplification. Use of a higher level representation generally results in the use of fewer but more general constructs, contributing to simplicity and conceptual understanding.

Turban cites the following advantages of modeling [9]:

1. Models make it easier to express complex ideas. For example, an architect builds a model to communicate ideas more easily to clients.
2. The main reason for modeling is the reduction of complexity. Models reduce complexity by separating those aspects that are unimportant from those that are important. Therefore, it makes complex situations easier to understand.
3. Models enhance and reinforce learning and training.
4. The cost of the modeling analysis is much lower than the cost of similar experimentation conducted with a real system.
5. Manipulation of the model (changing variables) is much easier than manipulating a real system.

To summarize, here are a few key ideas regarding modeling:

- A model is rarely correct on the first try.
- Always seek the advice and criticism of others. You can improve a model by reconciling different perspectives.
- Avoid excess model revisions, as they can distort the essence of your model. Let simplicity and elegance guide you through the process.

## 5.4 Introduction to the Unified Modeling Language

The unified modeling language is a language for specifying, constructing, visualizing, and documenting the software system and its components. The UML is a graphical language with sets of rules and semantics. The rules and semantics of a model are expressed in English, in a form known as object constraint language (OCL). OCL is a specification language that uses simple logic for specifying the properties of a system. The UML is not intended to be a visual programming language in the sense of having all the necessary visual and semantic support to replace programming languages. However, the UML does have a tight mapping to a family of object-oriented languages, so that you can get the best of both worlds.

The goals of the unification efforts were to keep it simple; to cast away elements of existing Booch, OMT, and OOSE methods that did not work in practice; to add elements from other methods that were more effective; and to invent new methods only when an existing solution was unavailable. Because the UML authors, in effect, were designing a language (albeit a graphical one), they had to strike a proper balance between minimalism (everything is text and boxes) and overengineering (having a symbol or figure for every conceivable modeling element). To that end, they were very careful about adding new things: They did not want to make the UML unnecessarily complex. A similar situation exists with the
problem of UML not supporting other diagrams. Booch et al. explain that other diagrams, such as the data flow diagram (DFD), were not included in the UML because they do not fit as cleanly into a consistent object-oriented paradigm. For example, activity diagrams accomplish much of what people want from DFDs and then some; activity diagrams also are useful for modeling work flow. The authors of the UML clearly are promoting the UML diagrams over all others for object-oriented projects but do not condemn all other diagrams. Along the way, however, some things were found that were advantageous to add because they had proven useful in other modeling practice.

The primary goals in the design of the UML were as follows [2, p. 3]:

1. Provide users a ready-to-use, expressive visual modeling language so they can develop and exchange meaningful models.
2. Provide extensibility and specialization mechanisms to extend the core concepts.
3. Be independent of particular programming languages and development processes.
4. Provide a formal basis for understanding the modeling language.
5. Encourage the growth of the OO tools market.
6. Support higher-level development concepts.
7. Integrate best practices and methodologies.

This section of the chapter is based on the *The Unified Modeling Language, Notation Guide Version 1.1* written by Grady Booch, Ivar Jacobson, and James Rumbaugh [2].

### 5.5 UML DIAGRAMS

Every complex system is best approached through a small set of nearly independent views of a model; no single view is sufficient. Every model may be expressed at different levels of fidelity. The best models are connected to reality. The UML defines nine graphical diagrams:

1. Class diagram (static)
2. Use-case diagram
3. Behavior diagram (dynamic):
   3.1. Interaction diagram:
      3.1.1. Sequence diagram
      3.1.2. Collaboration diagram
   3.2. Statechart diagram
   3.3. Activity diagram
4. Implementation diagram:
   4.1. Component diagram
   4.2. Deployment diagram

The choice of what models and diagrams one creates has a great influence on how a problem is encountered and how a corresponding solution is shaped. We will study applications of different diagrams throughout the book. However, in this chapter we concentrate on the UML notations and its semantics.
5.6 UML CLASS DIAGRAM

The UML class diagram, also referred to as object modeling, is the main static analysis diagram. These diagrams show the static structure of the model. A class diagram is a collection of static modeling elements, such as classes and their relationships, connected as a graph to each other and to their contents; for example, the things that exist (such as classes), their internal structures, and their relationships to other classes. Class diagrams do not show temporal information, which is required in dynamic modeling.

Object modeling is the process by which the logical objects in the real world (problem space) are represented (mapped) by the actual objects in the program (logical or a mini world). This visual representation of the objects, their relationships, and their structures is for ease of understanding. To effectively develop a model of the real world and to determine the objects required in the system, you first must ask what objects are needed to model the system. Answering the following questions will help you to stay focused on the problem at hand and determine what is inside the problem domain and what is outside it:

- What are the goals of the system?
- What must the system accomplish?

You need to know what objects will form the system because, in the object-oriented viewpoint, objects are the primary abstraction. The main task of object modeling is to graphically show what each object will do in the problem domain, describe the structure (such as class hierarchy or part-whole) and the relationships among objects (such as associations) by visual notation, and determine what behaviors fall within and outside the problem domain.

5.6.1 Class Notation: Static Structure

A class is drawn as a rectangle with three components separated by horizontal lines. The top name compartment holds the class name, other general properties of the class, such as attributes, are in the middle compartment, and the bottom compartment holds a list of operations (see Figure 5–1).

Either or both the attribute and operation compartments may be suppressed. A separator line is not drawn for a missing compartment if a compartment is suppressed; no inference can be drawn about the presence or absence of elements in it. The class name and other properties should be displayed in up to three sections. A stylistic convention of UML is to use an italic font for abstract classes and a normal (roman) font for concrete classes.

5.6.2 Object Diagram

A static object diagram is an instance of a class diagram. It shows a snapshot of the detailed state of the system at a point in time. Notation is the same for an object diagram and a class diagram. Class diagrams can contain objects, so a class diagram with objects and no classes is an object diagram.
5.6.3 Class Interface Notation

Class interface notation is used to describe the externally visible behavior of a class; for example, an operation with public visibility. Identifying class interfaces is a design activity of object-oriented system development. The UML notation for an interface is a small circle with the name of the interface connected to the class. A class that requires the operations in the interface may be attached to the circle by a dashed arrow. The dependent class is not required to actually use all of the operations. For example, a Person object may need to interact with the BankAccount object to get the Balance; this relationship is depicted in Figure 5–2 with UML class interface notation.

5.6.4 Binary Association Notation

A binary association is drawn as a solid path connecting two classes, or both ends may be connected to the same class. An association may have an association name. Furthermore, the association name may have an optional black triangle in it, the point of the triangle indicating the direction in which to read the name. The end of an association, where it connects to a class, is called the association role (see Figure 5–3).

5.6.5 Association Role

A simple association—the technical term for it is binary association—is drawn as a solid line connecting two class symbols. The end of an association, where it connects to a class, shows the association role. The role is part of the association, not
part of the class. Each association has two or more roles to which it is connected. In Figure 5-3, the association worksFor connects two roles, employee and employer. A Person is an employee of a Company and a Company is an employer of a Person.

The UML uses the term association navigation or navigability to specify a role affiliated with each end of an association relationship. An arrow may be attached to the end of the path to indicate that navigation is supported in the direction of the class pointed to. An arrow may be attached to neither, one, or both ends of the path. In particular, arrows could be shown whenever navigation is supported in a given direction. In the UML, association is represented by an open arrow, as represented in Figure 5-4. Navigability is visually distinguished from inheritance, which is denoted by an unfilled arrowhead symbol near the superclass.

In Figure 5-4, the association is navigable in only one direction, from the BankAccount to Person, but not the reverse. This might indicate a design decision, but it also might indicate an analysis decision, that the Person class is frozen and cannot be extended to know about the BankAccount class, but the BankAccount class can know about the Person class.

5.6.6 Qualifier

A qualifier is an association attribute. For example, a person object may be associated to a Bank object. An attribute of this association is the account#. The account# is the qualifier of this association (see Figure 5-5).
A qualifier is shown as a small rectangle attached to the end of an association path, between the final path segment and the symbol of the class to which it connects. The qualifier rectangle is part of the association path, not part of the class. The qualifier rectangle usually is smaller than the attached class rectangle (see Figure 5–5).

**5.6.7 Multiplicity**

*Multiplicity* specifies the range of allowable associated classes. It is given for roles within associations, parts within compositions, repetitions, and other purposes. A multiplicity specification is shown as a text string comprising a period-separated sequence of integer intervals, where an interval represents a range of integers in this format (see Figure 5–5):

lower bound .. upper bound.

The terms *lower bound* and *upper bound* are integer values, specifying the range of integers including the lower bound to the upper bound. The star character (*) may be used for the upper bound, denoting an unlimited upper bound. If a single integer value is specified, then the integer range contains the single values. For example,

0..1
0..*
1..3, 7..10, 15, 19..*

**5.6.8 OR Association**

An *OR association* indicates a situation in which only one of several potential associations may be instantiated at one time for any single object. This is shown as a dashed line connecting two or more associations, all of which must have a class in common, with the constraint string {or} labeling the dashed line (see Figure 5–6). In other words, any instance of the class may participate in, at most, one of the associations at one time.

**5.6.9 Association Class**

An *association class* is an association that also has class properties. An association class is shown as a class symbol attached by a dashed line to an association
path. The name in the class symbol and the name string attached to the association path are the same (see Figure 5–7). The name can be shown on the path or the class symbol or both. If an association class has attributes but no operations or other associations, then the name may be displayed on the association path and omitted from the association class to emphasize its "association nature." If it has operations and attributes, then the name may be omitted from the path and placed in the class rectangle to emphasize its "class nature."

5.6.10 N-Ary Association
An n-ary association is an association among more than two classes. Since n-ary association is more difficult to understand, it is better to convert an n-ary association to binary association. However, here, for the sake of completeness, we cover the notation of n-ary association. An n-ary association is shown as a large diamond with a path from the diamond to each participant class. The name of the association (if any) is shown near the diamond. The role attachment may appear on each path as with a binary association. Multiplicity may be indicated; however, qualifiers and aggregation are not permitted. An association class symbol may be at-

---

**FIGURE 5–6**
An OR association notation. A car may associate with a person or a company.

**FIGURE 5–7**
Association class.
FIGURE 5–8
An n-ary (ternary) association that shows association among class, year, and student classes. The association class GradeBook which contains the attributes of the associations such as grade, exam, and lab.

tached to the diamond by a dashed line, indicating an n-ary association that has attributes, operation, or associations. The example depicted in Figure 5–8 shows the grade book of a class in each semester.

5.6.11 Aggregation and Composition (a-part-of)
Aggregation is a form of association. A hollow diamond is attached to the end of the path to indicate aggregation. However, the diamond may not be attached to both ends of a line, and it need not be presented at all (see Figure 5–9).

Composition, also known as the a-part-of, is a form of aggregation with strong ownership to represent the component of a complex object. Composition also is referred to as a part-whole relationship. The UML notation for composition is a solid diamond at the end of a path. Alternatively, the UML provides a graphically nested form that, in many cases, is more convenient for showing composition (see Figure 5–10).

Parts with multiplicity greater than one may be created after the aggregate itself but, once created, they live and die with it. Such parts can also be explicitly removed before the death of the aggregate.

5.6.12 Generalization
Generalization is the relationship between a more general class and a more specific class. Generalization is displayed as a directed line with a closed, hollow arrowhead.

FIGURE 5–9
Association path.
at the superclass end (see Figure 5-11). The UML allows a *discriminator* label to be attached to a generalization of the superclass. For example, the class BoeingAirplane has instances of the classes Boeing 737, Boeing 747, Boeing 757, and Boeing 767, which are subclasses of the class BoeingAirplane. Ellipses (...) indicate that the generalization is incomplete and more subclasses exist that are not shown (see Figure 5-12). The constructor complete indicates that the generalization is complete and no more subclasses are needed.

If a text label is placed on the hollow triangle shared by several generalization paths to subclasses, the label applies to all of the paths. In other words, all subclasses share the given properties.
 figure 5-12
Ellipses (. . .) indicate that additional classes exist and are not shown.

5.7 USE-CASE DIAGRAM

The use-case concept was introduced by Ivar Jacobson in the object-oriented software engineering (OOSE) method [5]. The functionality of a system is described in a number of different use cases, each of which represents a specific flow of events in the system.

A use case corresponds to a sequence of transactions, in which each transaction is invoked from outside the system (actors) and engages internal objects to interact with one another and with the system's surroundings.

The description of a use case defines what happens in the system when the use case is performed. In essence, the use-case model defines the outside (actors) and inside (use case) of the system's behavior. Use cases represent specific flows of events in the system. The use cases are initiated by actors and describe the flow of events that these actors set off. An actor is anything that interacts with a use case: It could be a human user, external hardware, or another system. An actor represents a category of user rather than a physical user. Several physical users can play the same role. For example, in terms of a Member actor, many people can be members of a library, which can be represented by one actor called Member.

A use-case diagram is a graph of actors, a set of use cases enclosed by a system boundary, communication (participation) associations between the actors and the use cases, and generalization among the use cases.
Figure 5-13 diagrams use cases for a Help Desk. A use-case diagram shows the relationship among the actors and use cases within a system. A client makes a call that is taken by an operator, who determines the nature of the problem. Some calls can be answered immediately; other calls require research and a return call.

A use case is shown as an ellipse containing the name of the use case. The name of the use case can be placed below or inside the ellipse. Actors' names and use case names should follow the capitalization and punctuation guidelines of the model.

An actor is shown as a class rectangle with the label <<actor>>, or the label and a stick figure, or just the stick figure with the name of the actor below the figure (see Figure 5-14).
These relationships are shown in a use-case diagram:

1. **Communication.** The communication relationship of an actor in a use case is shown by connecting the actor symbol to the use-case symbol with a solid path. The actor is said to "communicate" with the use case.

2. **Uses.** A uses relationship between use cases is shown by a generalization arrow from the use case.

3. **Extends.** The extends relationship is used when you have one use case that is similar to another use case but does a bit more. In essence, it is like a subclass.

### 5.8 UML: Dynamic Modeling (Behavior Diagrams)

It is impossible to capture all details of a complex system in just one model or view. Kleyan and Gingrich explain:

One must understand both the structure and the function of the objects involved. One must understand the taxonomic structure of class objects, the inheritance and mechanisms used, the individual behaviors of objects, and the dynamic behavior of the system as a whole. The problem is somewhat analogous to that of viewing a sports event such as tennis or a football game. Many different camera angles are required to provide an understanding of the action taking place. Each camera reveals particular aspects of the action that could not be conveyed by one camera alone. [6]

The diagrams we have looked at so far largely are static. However, events happen dynamically in all systems: Objects are created and destroyed, objects send messages to one another in an orderly fashion, and in some systems, external events trigger operations on certain objects. Furthermore, objects have states. The state of an object would be difficult to capture in a static model.

The state of an object is the result of its behavior. Booch provides us an excellent example: "When a telephone is first installed, it is in idle state, meaning that no previous behavior is of great interest and that the phone is ready to initiate and receive calls. When someone picks up the handset, we say that the phone is now off-hook and in the dialing state; in this state, we do not expect the phone to ring: we expect to be able to initiate a conversation with a party or parties on another telephone. When the phone is on-hook, if it rings and then we pick up the handset, the phone is now in the receiving state, and we expect to be able to converse with the party that initiated the conversation [1]."

Booch explains that describing a systematic event in a static medium such as on a sheet of paper is difficult, but the problem confronts almost every discipline. In object-oriented development, you can express the dynamic semantics of a problem with the following diagrams:

**Behavior diagrams (dynamic):**
- Interaction diagrams
- Sequence diagrams
- Collaboration diagrams
- Statechart diagrams
- Activity diagrams
Each class may have an associated activity diagram that indicates the behavior of the class's instance (its object). In conjunction with the use-case model, we may provide a scripts or an interaction diagram to show the time or event ordering of messages as they are evaluated [1].

5.8.1 UML Interaction Diagrams

Interaction diagrams are diagrams that describe how groups of objects collaborate to get the job done. Interaction diagrams capture the behavior of a single use case, showing the pattern of interaction among objects. The diagram shows a number of example objects and the messages passed between those objects within the use case [3]. There are two kinds of interaction models: sequence diagrams and collaboration diagrams.

5.8.1.1 UML Sequence Diagram

Sequence diagrams are an easy and intuitive way of describing the behavior of a system by viewing the interaction between the system and its environment. A sequence diagram shows an interaction arranged in a time sequence. It shows the objects participating in the interaction by their lifelines and the messages they exchange, arranged in a time sequence.

A sequence diagram has two dimensions: the vertical dimension represents time, the horizontal dimension represents different objects. The vertical line is called the object's lifeline. The lifeline represents the object's existence during the interaction. This form was first popularized by Jacobson. An object is shown as a box at the top of a dashed vertical line (see Figure 5-15). A role is a slot for an object within a collaboration that describes the type of object that may play the role and its relationships to other roles. However, a sequence diagram does not show the relationships among the roles or the association among the objects. An object role is shown as a vertical dashed line, the lifeline.

**FIGURE 5-15**

An example of a sequence diagram.

Telephone Call

---

![Sequence Diagram](image)

Caller

OffHook

DialTone

DialNumber

Exchange

RingTone

Receiver

OffHook

Talk

OnHook
Each message is represented by an arrow between the lifelines of two objects. The order in which these messages occur is shown top to bottom on the page. Each message is labeled with the message name. The label also can include the argument and some control information and show self-delegation, a message that an object sends to itself, by sending the message arrow back to the same lifeline. The horizontal ordering of the lifelines is arbitrary. Often, call arrows are arranged to proceed in one direction across the page, but this is not always possible and the order conveys no information.

The sequence diagram is very simple and has immediate visual appeal—this is its great strength. A sequence diagram is an alternative way to understand the overall flow of the control of a program. Instead of looking at the code and trying to find out the overall sequence of behavior, you can use the sequence diagram to quickly understand that sequence [3].

5.8.1.2 UML Collaboration Diagram Another type of interaction diagram is the collaboration diagram. A collaboration diagram represents a collaboration, which is a set of objects related in a particular context, and interaction, which is a set of messages exchanged among the objects within the collaboration to achieve a desired outcome. In a collaboration diagram, objects are shown as figures. As in a sequence diagram, arrows indicate the message sent within the given use case. In a collaboration diagram, the sequence is indicated by numbering the messages. Some people argue that numbering the messages makes it more difficult to see the sequence than drawing the lines on the page. However, since the collaboration diagram is more compressed, other things can be shown more easily—for example, how the objects are linked together—and the layout can be overlaid with packages or other information.

A collaboration diagram provides several numbering schemes. The simplest is illustrated in Figure 5–16. You can also use a decimal numbering scheme (see Fig-

**FIGURE 5–16**
A collaboration diagram with simple numbering.

Telephone Call

Object

- Caller
  - 1: OffHook
- Exchange
  - 4: RingTone

Message

- 2: DialTone
- 3: DialNumber
- 5: OffHook
- 6: OnHook

Talk
Figure 5-17: A collaboration diagram with decimal numbering.

Figure 5-17, where 1.2: DialNumber means that the Caller (1) is calling the Exchange (2); hence, the number 1.2. The UML uses the decimal scheme because it makes it clear which operation is calling which other operation, although it can be hard to see the overall sequence [3].

Different people have different preferences when it comes to deciding whether to use sequence or collaboration diagrams. Fowler and Scott suggest that a sequence diagram is easier to read. Others prefer a collaboration diagram, because they can use the layout to indicate how objects are statically connected [3]. Fowler and Scott argue that the main advantage of interaction diagrams (both collaboration and sequence) is simplicity. You easily can see the message by looking at the diagram. The disadvantage of interaction diagrams is that they are great only for representing a single sequential process; they begin to break down when you want to represent conditional looping behavior. However, conditional behavior can be represented in sequence or collaboration diagrams through two methods. The preferred method is to use separate diagrams for each scenario. Another way is to use conditions on messages to indicate the behavior. The main guideline in developing interaction diagrams is simplicity. The interaction diagram loses its clarity with more complex conditional behavior. If you want to capture complex behavior in a single diagram, use an activity diagram, which will be described in a later section.

An interaction diagram basically is used to examine the behavior of objects within a single use case. It is good at showing collaboration among the objects but not so good at precise definition of the behavior [3].

5.8.2 UML Statechart Diagram

A statechart diagram (also called a state diagram) shows the sequence of states that an object goes through during its life in response to outside stimuli and messages. The state is the set of values that describes an object at a specific point in time and is represented by state symbols and the transitions are represented by arrows connecting the state symbols. A statechart diagram may contain subdiagrams.
A state diagram represents the state of the method execution (that is, the state of the object executing the method), and the activities in the diagram represent the activities of the object that performs the method. The purpose of the state diagram is to understand the algorithm involved in performing a method. To complete an object-oriented design, the activities within the diagram must be assigned to objects and the control flows assigned to links in the object diagram.

A statechart diagram is similar to a Petri net diagram, where a token (shown by a solid black dot) represents an activity symbol. When an activity symbol appears within a state symbol, it indicates the execution of an operation. Executing a particular step within the diagram represents a state within the execution of the overall method. The same operation name may appear more than once in a state diagram, indicating the invocation of the same operation in a different phase. An outgoing solid arrow attached to a statechart symbol indicates a transition triggered by the completion of the activity. The name of this implicit event need not be written, but conditions that depend on the result of the activity or other values may be included. An event occurs at the instant in time when the value is changed. A message is data passed from one object to another. At a minimum, a message is a name that will trigger an operation associated with the target object; for example, an Employee object that contains the name of an employee. If the Employee object received a message (getEmployeeName) asking for the name of the employee, an operation contained in the Employee class (e.g., returnEmployeeName) would be invoked. That operation would check the attribute Employee and then assign the value associated with that attribute back to the object that sent the message in the first place. In this case, the state of the Employee object would not have been changed. Now, consider a situation where the same Employee object received a message (updateEmployeeAddress) that contained a parameter (2000 21st Street, Seattle, WA):

updateEmployeeAddress (2000 21st Street, Seattle, WA)

In this case the object would invoke an operation from its class that would modify the value associated with the attribute Employee, changing it from the old address to the new address; therefore, the state of the employee object has been changed.

A state is represented as a rounded box, which may contain one or more compartments. The compartments are all optional. The name compartment and the internal transition compartment are two such compartments:

- The name compartment holds the optional name of the state. States without names are "anonymous" and all are distinct. Do not show the same named state twice in the same diagram, since it will be very confusing.
- The internal transition compartment holds a list of internal actions or activities performed in response to events received while the object is in the state, without changing states.

The syntax used is this: event-name argument-list / action-expression; for example, help / display help.
Two special events are entry and exit, which are reserved words and cannot be used for event names. These terms are used in the following ways: entry / action-expression (the action is to be performed on entry to the state) and exit / action-expression (the action is to be performed on exit from the state).

The statechart supports nested state machines; to activate a substate machine use the keyword do: do / machine-name (argument-list). If this state is entered, after the entry action is completed, the nested (sub)state machine will be executed with its initial state. When the nested state machine reaches its final state, it will exit the action of the current state, and the current state will be considered completed. An initial state is shown as a small dot, and the transition from the initial state may be labeled with the event that creates the objects; otherwise, it is unlabeled. If unlabeled, it represents any transition to the enclosing state. A final state is shown as a circle surrounding a small dot, a bull’s-eye. This represents the completion of activity in the enclosing state and triggers a transition on the enclosing state labeled by the implicit activity completion event, usually displayed as an unlabeled transition (see Figure 5-18).

The transition can be simple or complex. A simple transition is a relationship between two states indicating that an object in the first state will enter the second state and perform certain actions when a specific event occurs; if the specified con-
ditions are satisfied, the transition is said to "fire." Events are processed one at a time. An event that triggers no transition is simply ignored.

A complex transition may have multiple source and target states. It represents a synchronization or a splitting of control into concurrent threads. A complex transition is enabled when all the source states are changed, after a complex transition "fires" all its destination states. A complex transition is shown as a short heavy bar.¹ The bar may have one or more solid arrows from states to the bar (these are source states); the bar also may have one or more solid arrows from the bar to states (these are the destination states). A transition string may be shown near the bar. Individual arrows do not have their own transition strings (see Figure 5–19).

There certainly is no reason to prepare a state diagram for each class in your system. Indeed, many developers create rather large systems without bothering to create any state diagrams. However, state diagrams are useful when you have a class that is very dynamic. In that situation, it often is helpful to prepare a state diagram to be sure you understand each of the possible states an object of the class could take and what event (message) would trigger each transition from one state to another. In effect, state diagrams emphasize the use of events and states to determine the overall activity of the system.

### 5.8.3 UML Activity Diagram

An activity diagram is a variation or special case of a state machine, in which the states are activities representing the performance of operations and the transitions are triggered by the completion of the operations. Unlike state diagrams that focus on the events occurring to a single object as it responds to messages, an activity diagram can be used to model an entire business process. The purpose of an activity diagram is to provide a view of flows and what is going on inside a use case or among several classes. However, activity diagram can also be used to represent a class’s method implementation as we will see throughout the book.

¹A synchronization bar, which can represent synchronization, forking, or both.
An activity model is similar to a statechart diagram, where a token (shown by a black dot) represents an operation. An activity is shown as a round box, containing the name of the operation. When an operation symbol appears within an activity diagram or other state diagram, it indicates the execution of the operation. Executing a particular step within the diagram represents a state within the execution of the overall method. The same operation name may appear more than once in a state diagram, indicating the invocation of the same operation in different phases. An outgoing solid arrow attached to an activity symbol indicates a transition triggered by the completion of the activity. The name of this implicit event need not be written, but the conditions that depend on the result of the activity or other values may be included (see Figure 5–20). Several transitions with different conditions imply a branching off of control. If conditions are not disjoint, then the branch is nondeterministic. The concurrent control is represented by multiple arrows leaving a synchronization bar, which is represented by a short thick bar with incoming and outgoing arrows. Joining concurrent control is expressed by multiple arrows entering the synchronization bar. The activity diagram depicted in Figure 5–20, “Process Mortgage Request,” is a multistep operation, all of which are completed before the single operation Draw up insurance policy.
An activity diagram is used mostly to show the internal state of an object, but external events may appear in them. An external event appears when the object is in a "wait state," a state during which there is no internal activity by the object and the object is waiting for some external event to occur as the result of an activity by another object (such as a user input or some other signal). The two states are wait state and activity state. More than one possible event might take the object out of the wait state; the first one that occurs triggers the transition. A wait state is the "normal" state.

Activity and state diagrams express a decision when conditions (the UML calls them guard conditions) are used to indicate different possible transitions that depend on Boolean conditions of container object. The figure provided for a decision is the traditional diamond shape, with one or more incoming arrows and two or more outgoing arrows, each labeled by a distinct guard condition. All possible outcomes should appear on one of the outgoing transitions (see Figure 5–21).

Actions may be organized into swimlanes, each separated from neighboring swimlanes by vertical solid lines on both sides. Each swimlane represents responsibility for part of the overall activity and may be implemented by one or more objects. The relative ordering of the swimlanes has no semantic significance but might indicate some affinity. Each action is assigned to one swimlane. A transition may cross lanes; there is no significance to the routing of the transition path (see Figure 5–22).

5.8.4 Implementation Diagrams

Implementation diagrams show the implementation phase of systems development, such as the source code structure and the run-time implementation structure. There are two types of implementation diagrams: Component diagrams show the structure of the code itself, and deployment diagrams show the structure of the runtime system. These are relatively simple, high-level diagrams compared with the diagrams we have considered so far. Although we look at component-based development later in this book, a full discussion of implementation is beyond the scope of this book. This section is included to show the place of implementation in the UML.
5.8.4.1 Component Diagram Component diagrams model the physical components (such as source code, executable program, user interface) in a design. These high-level physical components may or may not be equivalent to the many smaller components you use in the creation of your application. For example, a user interface may contain many other off-the-shelf components purchased to put together a graphical user interface.

Another way of looking at components is the concept of packages. A package is used to show how you can group together classes, which in essence are smaller scale components. Packages will be covered in the next section, but a point worth mentioning here is that a package usually will be used to group logical components of the application, such as classes, and not necessarily physical components. However, the package could be a first approximation of what eventually will turn into physical grouping. In that case, the package will become a component [4].

A component diagram is a graph of the design’s components connected by dependency relationships. A component is represented by the boxed figure shown in Figure 5–23. Dependency is shown as a dashed arrow.

5.8.4.2 Deployment Diagram Deployment diagrams show the configuration of run-time processing elements and the software components, processes, and objects that live in them. Software component instances represent run-time manifestations
of code units. In most cases, component diagrams are used in conjunction with deployment diagrams to show how physical modules of code are distributed on various hardware platforms. In many cases, component and deployment diagrams can be combined [4].

A deployment diagram is a graph of nodes connected by communication association. Nodes may contain component instances, which means that the component lives or runs at that node. Components may contain objects; this indicates that the object is part of the component. Components are connected to other components by dashed-arrow dependencies, usually through interfaces, which indicate one component uses the services of another. Each node or processing element in the

FIGURE 5-24
The basic UML notation for a deployment diagram.
A package is a grouping of model elements. Packages themselves may contain other packages. A package may contain both subordinate packages and ordinary model elements. The entire system can be thought of as a single high-level package with everything else in it. All UML model elements and diagrams can be organized into packages.

A package is represented as a folder, shown as a large rectangle with a tab attached to its upper left corner. If contents of the package are not shown, then the name of the package is placed within the large rectangle. If contents of the package are shown, then the name of the package may be placed on the tab (see Figure 5–25). The contents of the package are shown within the large rectangle.

Figure 5–26 shows an example of several packages. This figure shows three packages (Clients, Bank, and Customer) and three classes (Account class, Savings class, and Checking class) inside the Business Model package. A real model would have many more classes in each package. The contents might be shown if they
FIGURE 5–26
A package and its dependencies.

are small, or they might be suppressed from higher levels. The entire system is a package.

Figure 5–26 also shows the hierarchical structure, with one package dependent on other packages. For example, the Customer depends on the package Business Model, meaning that one or more elements within Customer depend on one or more elements within the other packages. The package Business Model is shown partially expanded. In this case, we see that the package Business Model owns the classes Bank, Checking, and Savings as well as the packages Clients and Bank. Ownership may be shown by a graphic nesting of the figures or by the expansion of a package in a separate drawing.

Packages can be used to designate not only logical and physical groupings but also use-case groups. A use-case group, as the name suggests, is a package of use cases.

Model dependency represents a situation in which a change to the target element may require a change to the source element in the dependency, thus indicating the relationship between two or more model elements. It relates the model elements themselves and does not require a set of instances for its meaning. A dependency is shown as a dashed arrow from one model element to another on which the first element is dependent (see Figure 5–27).
5.10 UML EXTENSIBILITY

In this section, we look at general purpose mechanisms, which may be applied to any modeling element, and at the extensibility of the UML.

5.10.1 Model Constraints and Comments

Constraints are assumptions or relationships among model elements specifying conditions and propositions that must be maintained as true; otherwise the system described by the model would be invalid. Some constraints, such as association OR constraints, are predefined in the UML; others may be defined by users.

Constraints are shown as text in braces, {} (see Figure 5–27). The UML also provides language for writing constraints in the OCL. However, the constraint may be written in a natural language. A constraint may be a “comment,” in which case it is written in text. For an element whose notation is a text string such as an attribute, the constraint string may follow the element text string. For a list of elements whose notation is a list of text strings, such as the attributes within class, the constraint string may appear as an element in the list. The constraint applies to

FIGURE 5–27
An example of constraints. A person is a manager of people who work for the accounting department.

FIGURE 5–28
Note.
all succeeding elements of the list until reaching another constraint string list element or the end of the list. A constraint attached to an individual list element does not supersede the general constraints but may modify individual constraints within the constraint’s string. For a class or association path, the constraints string may be placed near the symbol name.

The example depicted in Figure 5-27 shows two classes and two associations. The constraint is shown as a dashed arrow from one element to the other, labeled by the constraints string in braces. The direction of the arrow is relevant information within the constraint.

5.10.2 Note

A note is a graphic symbol containing textual information; it also could contain embedded images. It is attached to the diagram rather than to a model element. A note is shown as a rectangle with a “bent corner” in the upper right corner. It can contains any length text (see Figure 5-28).

5.10.3 Stereotype

Stereotypes represent a built-in extensibility mechanism of the UML. User-defined extensions of the UML are enabled through the use of stereotypes and constraints. A stereotype, in effect, is a new class of modeling element introduced during modeling. It represents a subclass of an existing modeling element with the same form (attributes and relationships) but a different intent. UML stereotypes extend and tailor the UML for a specific domain or process.

The general presentation of a stereotype is to use a figure for the base element but place a keyword string above the name of the element (if used, the keyword string is the name of a stereotype within matched guillemets, "<<", ">>", such as "<<flow>>"). Note that a guillemet looks like a double angle-bracket, but it is a single character in most fonts. The stereotype allows extension of UML notation as well as a graphic figure, texture, and color. The figure can be used in one of two ways: (1) instead of or in addition to the stereotype keyword string as part of the symbol for the base model element or (2) as the entire base model element (see Figure 5-29). Other information contained by the base model element symbol is suppressed.

The main shortcoming of extensive use of stereotypes is that it makes the model less universal and not easily interchangeable with other tools or software systems.

5.11 UML META-MODEL

The UML defines notations as well as a meta-model. UML graphic notations can be used not only to describe the system’s components but also to describe a model itself. This is known as a meta-model. In other words, a meta-model is a model of modeling elements. The purpose of the UML meta-model is to provide a single,
common, and definitive statement of the syntax and semantics of the elements of the UML.

The meta-model provides us a means to connect different UML diagrams. The connection between the different diagrams is very important, and the UML attempts to make these couplings more explicit through defining the underlying model (meta-model) while imposing no methodology.

The presence of this meta-model has made it possible for its developers to agree on semantics and how those semantics would be best rendered. This is an important step forward, since it can assure consistency among diagrams. The meta-model also (in the future) can serve as a means to exchange data between different CASE tools. Additionally, the meta-model has made it possible for a team to explore ways to make the modeling language much simpler by, in a sense, unifying the elements of the unified modeling language. Figure 5–30 is an example of the UML meta-model that describes relationship with association and generalization; similarly association is depicted as a composition of association roles. Here we have used UML modeling elements (such as generalization and composition) to describe the model itself; hence, the term meta-model.

Most users of methods do not need such a deep understanding to get some value out of UML notation. However, it does help define what constitutes a well-formed model, that is, one that is syntactically correct.
5.12 SUMMARY

A model is a simplified representation of reality, simplified because reality is too complex or large and much of the complexity actually is irrelevant to the problem being described or solved.

The unified modeling language was developed by Booch, Jacobson, and Rumbaugh. The UML encompasses the unification of their modeling notations.

The UML class diagram is the main static structure analysis diagram for the system. It represents the class structure of a system with relationships between classes and inheritance structure. The class diagram is developed through use-case, sequence, and collaboration diagrams.

The use-case diagram captures information on how the system or business works or how you wish it to work. It is a scenario-building approach in which you model the processes of the system. It is an excellent way to lead into object-oriented analysis of the system.

In the UML sequence diagram is for dynamic modeling, where objects are represented by vertical lines and messages passed back and forth between the objects are modeled by horizontal vectors between the objects.

The UML collaboration diagram is an alternative view of the sequence diagram, showing in a scenario how objects interrelate with one another.

Statechart diagrams, another form of dynamic modeling, focus on the events occurring within a single object as it responds to messages; an activity diagram is used to model an entire business process. Thus, an activity model can represent several different classes.

Implementation diagrams show the implementation phase of systems development, such as the source code and run-time implementation structures. The two
types of implementation diagrams are component diagrams, which show the structure of the code itself, and deployment diagrams, which show the structure of the run-time system.

Stereotypes represent a built-in extensibility mechanism of the UML. User-defined extensions of the UML are enabled through the use of stereotypes and constraints.

UML graphical notations can be used not only to describe the system’s components but also to describe a model itself; this is known as a meta-model. It is a model of modeling elements. The purpose of the UML meta-model is to provide a single, common, and definitive statement of the syntax and semantics of the elements of the UML.

KEY TERMS

Activity diagram (p. 109)
Association class (p. 97)
Class diagram (p. 94)
Collaboration diagram (p. 105)
Component diagram (p. 112)
Deployment diagram (p. 112)
Dynamic model (p. 91)
Generalization (p. 99)
Implementation diagram (p. 111)
Interaction diagram (p. 104)
Lifeline (p. 104)
Meta-model (p. 117)
Model dependency (p. 115)
Model (p. 89)
Multiplicity (p. 97)
N-ary association (p. 98)
Note (p. 117)
OR association (p. 97)
Package (p. 114)
Qualifier (p. 96)
Sequence diagram (p. 104)
Statechart diagram (p. 106)
Static model (p. 90)
Stereotype (p. 117)
Swimlane (p. 111)
Use-case diagram (p. 101)

REVIEW QUESTIONS

1. What is a model?
2. Why do we need to model a problem?