[PART]

OBJECT-ORIENTED Software Development

Chapter 1 introduced the software development process and discussed the importance of the requirements and specifications phases. This section (Part I) concentrates on the design and implementation phases of the software life cycle. Chapter 2, the first chapter in Part I, discusses the design phase and introduces fundamental concepts associated with object-oriented programming and design. Chapter 3 illustrates how these basic object-oriented techniques can be implemented in Java. Chapter 4, the final chapter in Part I, presents a significant case study that uses the concepts introduced in Chapters 2 and 3.

After working through the material in Part I, you will have a better understanding of object-oriented design and programming in Java. You will also be familiar with the Unified Modeling Language (UML), which is widely used in industry to document the design of object-oriented software systems. Finally, you will have had an opportunity to design and implement an object-oriented system.

[CHAPTER] 2 OBJECT-ORIENTED Design and Programming

2.1 Introduction

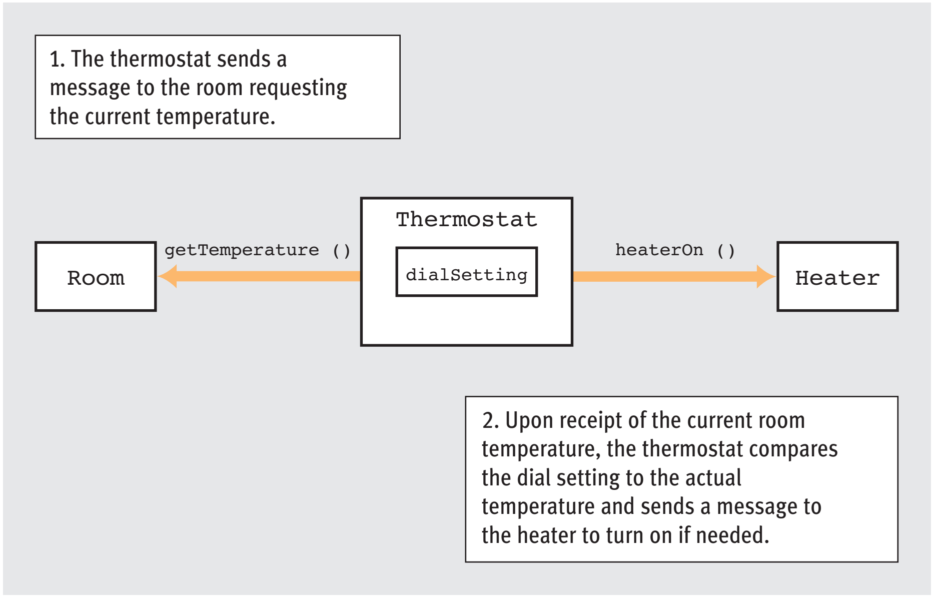
As human beings, we make sense of our environment by trying to identify and understand the things, or **objects**, that surround us. For example, I am writing this chapter in my dining room. I am sitting on a chair in front of a table that holds my computer. I can see windows and doors on every wall. In the center of the ceiling is a large fan controlled by two switches on the far wall. A thermostat controls the heaters beneath the windows on the opposite wall. The place where you are reading this chapter probably contains many similar objects, such as windows, lights, and controls for lighting and temperature.

Although the objects we see in our environment may differ, we perceive and describe these objects in similar ways. In fact, we do much more than perceive objects around us. We also interact with objects, and objects interact with each other, to get things done. For example, I am interacting with my computer to write these words. You are interacting with a printed version of this book by reading it. We describe and understand the world around us by (a) identifying the objects in our world, (b) understanding what these objects can do, and (c) interacting with these objects to accomplish a specific task. For example, to increase the temperature in my dining room, I first need to know that an object called a thermostat accomplishes this task. Then I need to find the thermostat and learn how to use it. Finally, I need to adjust it properly. The thermostat then interacts with the room to determine the current temperature and interacts with the heaters to change the temperature as necessary.

Objects by their nature hide, or **encapsulate**, their inner workings. Encapsulation means "to enclose in as if in a capsule." Placing this definition in an object-oriented context, encapsulation is a method by which information can be confined or hidden inside objects so it is not visible to the outside world. Using objects to encapsulate information makes it possible to view a complex system only in terms of its external interface, without regard for its internal structure. In other words, I neither know nor care about the inner workings of a thermostat. All I need to know is that if a room is too cold, I can warm it by raising the temperature setting, and if the room is too warm, I can cool it by lowering the setting. I have no need to know how the thermostat accomplishes these feats. In addition, if its internal structure is changed, it will not affect my ability to use the thermostat because I will still operate it in the same way. Encapsulation enables us to build and understand complex systems that consist of large collections of objects. Encapsulation also makes it possible to replace an old object with a new one without having to rebuild the entire system.

To help us understand what objects can do and how we interact with them, objects with similar properties and common attributes are grouped together into sets called **classes**. For example, even though you have never used the thermostat in my dining room, you could probably use it because you understand, in general, what a thermostat is and what it does. You expect to find a button or dial on the front, along with some temperature markings, and you use the dial to set the desired room temperature. This level of understanding allows you to walk into almost any room and adjust the temperature to your liking. Even though there are thousands of types of thermostats, and you do not know which one will be in a specific room, you can almost certainly use it correctly because all thermostat objects belong to the same class, and they share a set of common functions and attributes.

Computers, unlike human beings, do not describe their world in terms of objects, but instead focus on the sequence of steps required to complete a given task. A **procedure** is "a series of steps followed in a regular definite order," and at the machine level, a program is a collection of procedures and statements that are executed in a predetermined order. This fetching, decoding, and executing of instructions in a well-defined sequence is the very essence of the von Neumann architecture. To make this distinction between human and computer thought processes a little clearer, think for a moment how you would explain the behavior of a thermostat. You would describe it as a device that consists of a dial, a temperature sensor, and a connection to a heating unit. The dial records the desired room temperature. The temperature sensor monitors both the setting on the dial and the current room temperature and turns on the heater when the room temperature falls below the setting on the dial. Your description has been given in terms of entities, their functions, and their communications with each other. This type of description is called an **object-oriented view** of the problem, and it is diagrammed in Figure 2-1a.


1. The thermostat sends a message to the room requesting the current temperature.
Thermostat
dialSetting
Room
getTemperature ()
heaterOn ()
Heater
2. Upon receipt of the current room temperature, the thermostat compares the dial setting to the actual temperature and sends a message to the heater to turn on if needed.

[FIGURE 2-1a] Object-oriented description of a thermostat

A computer, on the other hand, would take a much more procedural, step-by-step view of the problem. It might describe a thermostat as a device that carries out the five-step algorithm in Figure 2-1b.

```
1   Obtain the current temperature of the room.
2   Obtain the current setting on the dial.
3   If the room temperature is less than the setting on the dial, turn on the heater.
4   If the room temperature is greater than or equal to the setting on the dial, turn
    off the heater.
5   Go back to Step 1.
```

[FIGURE 2-1b] Procedure-oriented description of a thermostat

This is called a **procedure-oriented view** of the problem. Although both descriptions explain what a thermostat does, most people would be more comfortable with the object-oriented view of Figure 2-1a, which describes a collection of interacting objects, whereas the computer would prefer the procedural, step-by-step description in Figure 2-1b.

Because human beings are generally most comfortable dealing with objects, you might think that all high-level programming languages would allow programmers to use objects to describe their programs. In fact, this is not the case. The earliest high-level programming languages (FORTRAN, COBOL, ALGOL, and Pascal) were strictly procedural languages. They gave programmers the tools required to produce a sequence of steps to solve a given task, much like the algorithm of Figure 2-1b. Essentially, these programming languages made programming easier by providing powerful, high-level procedural constructs to replace cumbersome, low-level machine language primitives. For example, instead of thinking about the dozen or so machine language instructions required to construct a loop, a programmer could use the high-level `for`, `while`, or `do` constructs instead. The compiler would then translate these high-level constructs into the actual machine code executed by the computer. Although procedural languages increased productivity and improved our ability to construct programs, programmers were still required to *think* like a computer. Instead of describing the world in terms of objects and their interactions, programmers were forced to construct a step-by-step, procedural description of the problem they wanted to solve. The designers of these early languages did not change the underlying *mode* of problem solving. They only changed the ease with which problems could be solved.

In the early days of language design, computer scientists did not explore object-oriented programming models for two reasons. First, when procedural programming languages were developed in the 1950s and 1960s, researchers were only beginning to understand the syntactic and semantic theories that were the basis of programming language design and compiler construction. Object-oriented languages require sophisticated compiler support that early compiler designers could not yet provide. The technology of the 1950s and 1960s was not ready to support object-oriented software development. Second, computing was very new, and language designers were still learning the basic principles of programming. It is natural to build upon what you know and what you have seen; thus, most early language design work focused on programming models that were closely related to the underlying von Neumann architecture.

In 1962, Ole-Johan Dahl and Kristen Nygaard at the Norwegian Computing Center in Oslo began designing the first object-based programming language, Simula. Simula was a special-purpose language designed for discrete event simulation, and it focused on modeling rather than general-purpose computing. This change in focus forced the use of several innovative computing techniques; for example, Simula was the first language that allowed computational constructs using objects. Simula never was widely used, but it had a significant influence on the future design and development of modern object-oriented languages.

Two important design projects were based on the early work with Simula. In 1972, Alan Kay of the Xerox Palo Alto Research Center (PARC) created a language called Smalltalk in an attempt to improve the novel ideas he saw in Simula. Kay was intrigued by the "promise of an entirely new way to structure computations." It took Kay several years to develop his insights and devise efficient mechanisms that worked on the primitive computer systems of the early 1970s. In 1980, Smalltalk became one of the first commercial object-oriented programming languages, and it set the standard for the languages that followed. The class libraries included with the Smalltalk environment were the basis for libraries provided with many modern object-oriented languages.

Bjarne Stroustrup used Simula while working on his PhD thesis at Cambridge University in England. He was studying distributed systems and felt that the object-oriented features of Simula would be ideal for his work. Stroustrup was impressed by the way "the concepts of the language helped me to think about the problems in my application." He was also intrigued that the mechanisms provided by Simula became even more helpful as the size of his program increased. In other words, he felt that the ideas embodied in Simula might be useful to manage the complexity of huge software projects.

After completing his PhD, Stroustrup was hired by Bell Laboratories and worked on a project to analyze how the UNIX operating system could be distributed over a computer network. He decided that the best way to attack this problem was to devise a new language that added Simula's class concept to C. At the time, C was the most widely used system implementation language. In 1979, Stroustrup began work on a new language he initially called "C with classes," but which later evolved into what we now know as C++, the first widely used object-oriented programming language. By the mid-1980s, C++ was becoming popular in the United States, clearly demonstrating the benefits of the object-oriented approach.

In 1991, Sun Microsystems embarked on a variety of projects to branch out into new areas of computing technology. One of these projects explored the use of inexpensive microprocessors in various consumer electronics devices, including personal digital assistants (PDAs) and televisions. The software to drive these devices would have to run on different platforms, fit into a small amount of memory, and be robust and secure. The team at Sun had planned to use C++, but they encountered a number of problems that made them consider designing a new language.

James Gosling began developing a language to address the problems faced by the Sun team. Because most programmers at that time were using either C or C++, the team decided to make the structure of this new language similar to C++. This made the structure easier for programmers to understand and eliminated the need for extreme retraining. Because the language would be used to develop software that could run on many different types of simple devices, the goals of the new language were to keep it small, robust, and highly portable. The language was initially named Oak, after the tree outside Gosling's office window. It was eventually renamed Java because of trademark conflicts with an existing Oak product. Sun's consumer electronics initiative failed, but by 1994, the World Wide Web was becoming increasingly popular. Java, although designed for a totally different purpose, turned out to be well suited for Web-based applications because it was platform independent, secure, and robust. The Web put distributed, secure programming in high demand, and Java found a new lease on life. By the beginning of the 21st century, Java was well on its way to displacing C++.

Regardless of whether you use Smalltalk, C++, or Java, the object-oriented paradigm is undoubtedly the most widely used model for the design and implementation of large software systems. The next section of this chapter introduces the topics of objects, classes, and inheritance, which are the fundamental concepts of object-oriented programming. The last section discusses techniques and tools you can use to design object-oriented programs. After reading this chapter, you will understand object-oriented programming and how to design simple object-oriented programs.

OLD DOG, NEW TRICKS

In the 1960s, the most widely taught programming languages in colleges and universities were FORTRAN and COBOL. In the 1970s, the most popular languages were Pascal for introductory courses and C/UNIX for advanced work. With the growing interest in object-oriented (OO) programming, the most popular language of the 1980s was C++, an object-oriented extension of C. In the late 1990s, Java took over the top spot in the computer science curriculum, a position it still holds today. The only certainty in using programming languages is that change is inevitable. During your career, many new languages will appear and then disappear.

For example, even though Java is enormously popular, there is a good deal of research into designing newer and better OO languages. Ruby is an OO language developed in 1996

continued

by Yukihiro Matsumoto, a Japanese computer scientist. Its goal is to simplify the work a programmer must do to design and build correct programs. Matsumoto based his work on POLS, the Principle of Least Surprise, in which all aspects of the language are intuitive and easily understood. Boo is an open source OO language developed in 2003 in the Netherlands. It is based on the Common Language Infrastructure developed by Microsoft. Boo was influenced by Python, a multiparadigm language developed in the 1990s. Like Ruby, Python is based on the philosophy that it is more important to design for the human than for the machine. Other OO languages being investigated include Unicon, Squeak, Dylan, Self, Io, and Agora. Will one of them, or another unknown language, become the next Java and take over the top spot in our curricula?

The point of this discussion is to emphasize three key ideas about learning a language:

1 Don't get "tunnel vision" and see programming through the lens of a single language's syntax and semantics. Learn and be comfortable with multiple languages as well as multiple paradigms, such as functional, logic based, scripting, concurrent, and rule based. Be open and accepting of different ways of viewing problems and their solutions.

2 Don't be unwilling to discard your current language to make use of a new, improved one. A programming language is simply a tool; like any tool, we typically throw it away when a better one is available.

3 The truly important part of programming language instruction is not the long litany of rules associated with the syntax of one particular language. Instead, it is the idea of "learning how to learn"—specifically, recognizing what is unique about a new language and understanding the new concepts that the language has made available. Once you have mastered this idea, the rest will follow.

2.2 Object-Oriented Programming

2.2.1 Objects

Although we have an intuitive understanding of objects in the real world, exactly what is an object in an object-oriented programming language? You might think this question would be easy to answer. Surprisingly, it is not, and there are several definitions in current usage.

Grady Booch, a well-known author and computer scientist, proposed the following widely used definition:

An object is an entity that has state, behavior, and identity.

For many, this definition may not be very helpful, and it may even be confusing, so let's take it apart and look at it piece by piece.

In Section 2.1, we saw that a thermostat needs certain values to remember to function properly. For example, it has to know the desired room temperature and whether it has turned on the heaters in the room. The information that an object needs to remember makes up its **state**. More formally, the state of an object is composed of the object's properties and the current values of those properties. A **property** is a characteristic, quality, or feature that contributes to making an object unique. For example, the desired temperature setting is a property of a thermostat but not of an elevator. An elevator would have properties such as the state of its doors, its direction of travel, and its current location. The **value** of a property is the specific quantity stored in the property at any given time. For example, an elevator's current values might be "doors closed, going up, on the 10th floor." The properties of an object are fixed, or static, whereas the values of these properties are dynamic and vary over time. A thermostat always needs to know the desired room temperature, but this setting may be 63 degrees at night and 68 degrees during the day.

The second characteristic in Booch's definition is behavior. The **behavior** of an object describes how it reacts in terms of state changes and interactions with other objects. Using the thermostat example again, one of its behaviors is that the heater turns on when the current room temperature falls below the desired temperature. The thermostat changes state in reaction to a change in temperature. As another example, an elevator will close its doors and move upward when it is on the ground floor and someone presses the button labeled 23. All objects exhibit different behaviors, which other objects may use to achieve a desired result.

Object-oriented programming languages model behaviors in one of two ways. A **message** is a unit of information that one object can send to another to invoke a certain behavior. Languages that model behavior using messages, such as Smalltalk, allow programmers to define the different messages an object is willing to receive and the specific behaviors that will be associated with the arrival of each message type. An object then sends the appropriate message to invoke a particular behavior. For example, a furnace will accept messages from a thermostat that instruct it to turn on or off, as shown in Figure 2-1a.

Languages such as Java use a different approach. They model behavior by providing procedures, called **methods**, which are associated with a specific type of object. These methods can be called, or **invoked**, by other objects to achieve certain behavior. Although they represent different viewpoints, "sending a message" and "invoking a method" do essentially the same thing. They both cause an object to perform a specific action or behavior.

Conventional parameter-passing mechanisms provide a way to pass information to the procedure that implements the behavior. Using methods, for example, a furnace would have an `activateFurnace()` method that takes a Boolean parameter specifying whether the furnace should be turned on or off. A thermostat object could then indicate that the furnace should be turned on by invoking the `activateFurnace()` method with the Boolean value true as the argument. When this method is invoked, the furnace object will have all the information it requires to determine whether to activate the heating unit. Of course, the methods associated with an object depend on the object type. A thermostat object will include `activateHeaters()` and `temperatureChange()` methods, whereas an elevator object would contain the methods `openDoor()`, `closeDoor()`, `goDown()`, and `goUp()`.

The methods associated with an object can be grouped into four categories based on the object's response, as summarized in Table 2-1. The first two categories, accessors and mutators, allow you to obtain or change the state of an object. An **accessor** is used to query but not change the state of an object; a **mutator** provides a way to change the state of an object. It is easy to see real-world examples of both method types. For example, a thermostat can display its current temperature setting (an accessor) and provide a mechanism for changing that setting (a mutator).

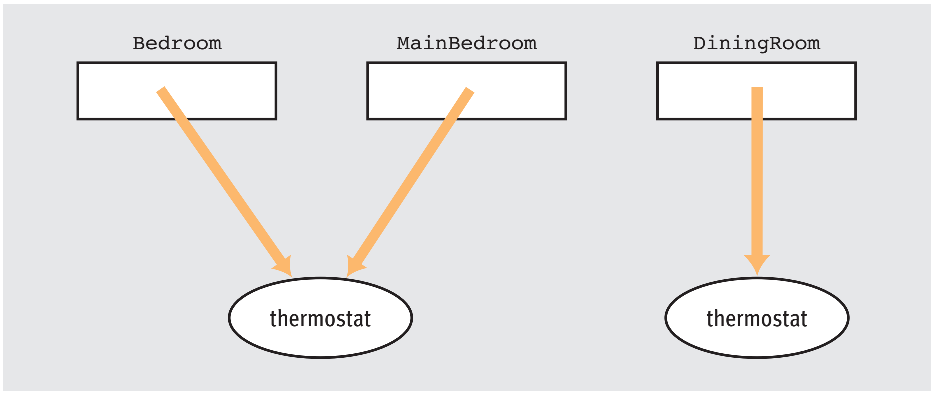
METHOD TYPE	ACTION PERFORMED
Accessor	Return some information about the object's state but do not change the object's state
Mutator	Change the object's state
Constructor	Create and initialize a new object
Destructor	Remove an object currently in the system; free up any resources associated with the object

[TABLE 2-1] Types of methods

In programming, as in the real world, objects must be created and destroyed. **Constructors** provide a mechanism for creating and initializing operations to carry out on a new object. A **destructor**, on the other hand, allows an existing object to be removed from the system when it is no longer needed.

Identity is the third and last part of Booch's definition of an object. An object-oriented language must provide a mechanism to distinguish one object from another. If I write a program that creates two thermostats, I must have a way of specifying which thermostat I want to use. I must be able to say something like, "I want to adjust the setting of the thermostat in the dining room, not the one in the bedroom."

Most object-oriented programming languages use **reference** variables to identify objects. A reference variable points to a specific object. For example, I may define the reference variables `DiningRoom` and `Bedroom` in my program and have them refer to the objects that model the thermostats in the dining room and bedroom, respectively. To change the setting of the bedroom thermostat, I would specify that the object referred to by the variable `Bedroom` should change its desired temperature. Note that a reference variable can refer to only one object at a time, but two or more reference variables may refer to the same object, as shown in Figure 2-2.



[FIGURE 2-2] Reference variables

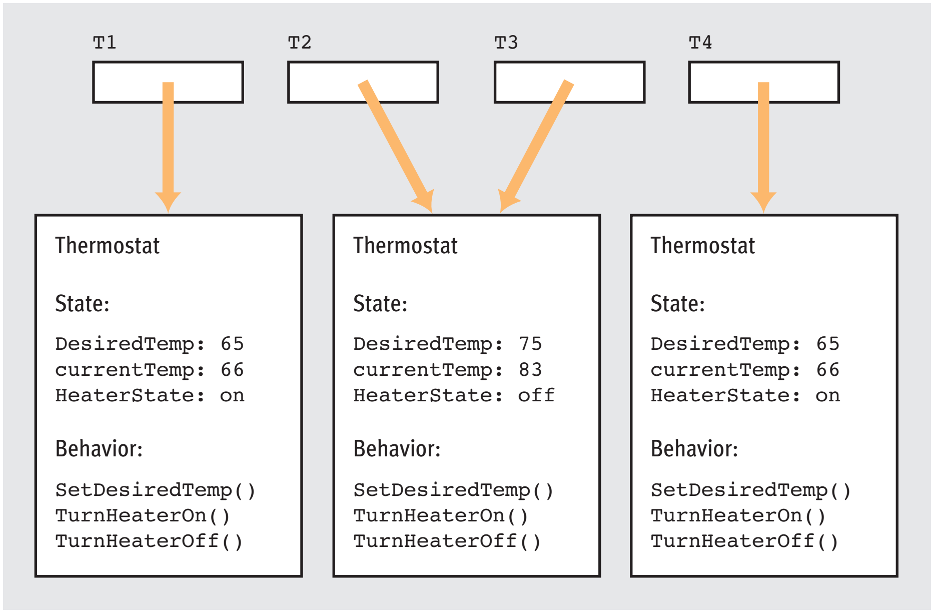
The concept of identity is closely related to that of **equality**. Equality attempts to answer the question: Are two objects equal? At first glance, this seems like a simple concept, but it is complicated by the fact that objects can be identified and distinguished in two different ways.

Imagine yourself standing at the same location on a river bank on two different days. Did you see the *same* river on both days, even though the water that flowed through it was different? In other words, do we identify a river by its location or by its contents? If we identify the river by content, then you saw two different rivers on those two days, even though you were in the same location.

Object-oriented programming languages typically provide the ability to compare objects by name (location) and by content. Two variables are said to be **name equivalent** if they both refer to the same object. The variables `Bedroom` and `MainBedroom` in Figure 2-2 are name equivalent because they both refer to the same thermostat object. When comparing objects by name, we are only interested in determining if they are the same object. Name equivalence is often implemented by comparing the memory addresses of the two objects. In this "shallow" way of identifying objects, two objects are considered the same if they occupy the same memory locations. That is, they are the exact same object.

Two objects are said to be **content equivalent** if both are from the same class, and the state information stored in each object is the same. To determine content equivalence, the programmer must understand the structure of the objects being compared and determine if the state of both objects is exactly the same.

Figure 2-3 shows three thermostat objects identified using the four reference variables `T1`, `T2`, `T3`, and `T4`. The variables `T2` and `T3` refer to the same object and are therefore considered name equivalent. The objects referenced by `T1` and `T4` are content equivalent because they have the same state, but they are not name equivalent because the objects occupy different areas in memory. Finally, `T1` and `T2` are neither name equivalent nor content equivalent, as they occupy different areas of memory and have different state values, respectively.



[FIGURE 2-3] Examples of name and content equivalence

2.2.2 Classes

Human beings group similar objects together to understand what the objects are capable of doing. For example, when I say "thermostat," you think about a device that controls the temperature setting. Not only do you have a general idea of what the object does, you have a good idea of how to use it.

A **class** is a group of objects that share common state and behavior. A class is an abstraction or description of an object. An object, on the other hand, is a concrete entity that exists in space and time. Object-oriented programming languages use classes to define the state and behavior associated with objects and to provide a means to create the objects that make up a program. You can think of a class as a blueprint or template from which objects can be created, or, to use object-oriented terminology, **instantiated**. While `Car` is a class, my 1993 Toyota Corolla is a specific instance, or object, of that class.

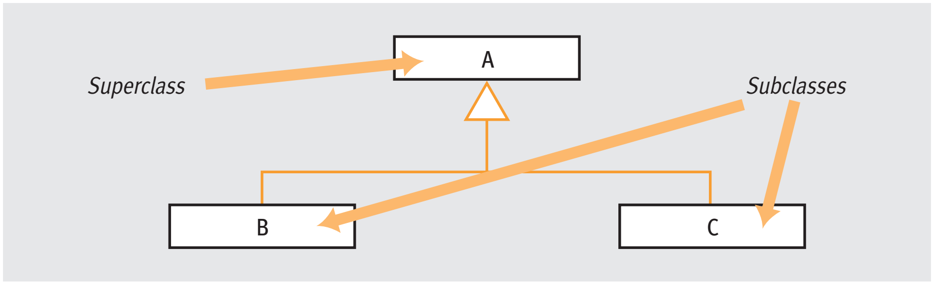
For example, to write a program that models the heating system in my dining room, I might start by defining three classes: `Room`, `Thermostat`, and `Heater`. I could then use these classes to instantiate as many distinct room, thermostat, and heater objects as required. Even though my dining room has two heaters, I only need to define a single heater class, from which I can instantiate two (or more) distinct heater objects. We saw this situation in Figure 2-3, in which a single thermostat class was used to instantiate three thermostat objects.

Although they are not typically referred to as classes, you can think of a data type in a conventional programming language as a class. An **integer** is a group of data values that share common state and behavior. The state of an integer consists of its numerical value, and the shared behavior includes operations such as addition, subtraction, multiplication, and division. The term *integer* does not represent a specific integer; instead, it refers to *all* integers. When we need a specific integer variable in a program, we use the integer data type to declare an integer variable (instantiation) and initialize its value. We can then use accessor methods to print its value and mutator methods to assign a new value.

2.2.3 Inheritance

Classes, like objects, do not exist in isolation. You can infer useful information about classes and the objects that can be instantiated from them based on the relationships that exist between classes. For example, consider the following three classes: `ProgrammableThermostat`, `AnalogThermostat`, and `Thermostat`. Given our understanding of these names, we can infer that both a `ProgrammableThermostat` and an `AnalogThermostat` are `Thermostats`. This means that anything a `Thermostat` can do, a `ProgrammableThermostat` and `AnalogThermostat` can do as well. You should be able to replace a `Thermostat` with either a `ProgrammableThermostat` or an `AnalogThermostat` and still expect the heating system to function correctly. The heating system may have additional functionality (e.g., it may now be programmable), and the techniques used to monitor and adjust the temperature may have changed, but the state and behavior in the original thermostat are still present in the new system.

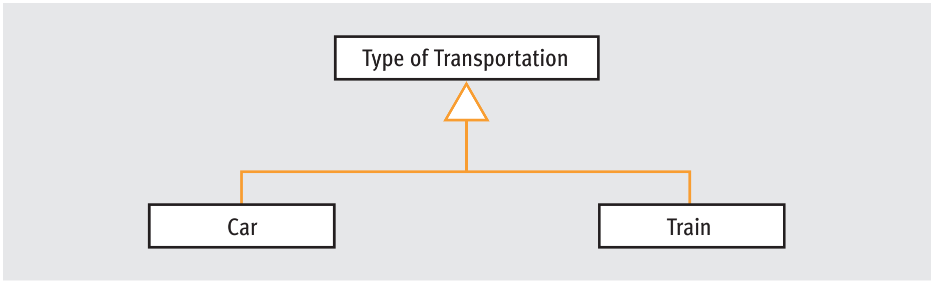
When the state and behavior of one class are a subset of the state and behavior of another more general class, the classes are related by **inheritance**. The more general class (for example, `Thermostat`) is referred to as a **superclass**, or **parent**, of a second, more specialized class (such as `ProgrammableThermostat`). The second class is called a **subclass** of the superclass and is said to inherit the state and behavior of the superclass. The inheritance relationship can be diagrammed as shown in Figure 2-4.



[FIGURE 2-4] Inheritance

When a subclass such as B or C in Figure 2-4 inherits from a superclass, the state and behavior of the subclass are an extension of the state and behavior associated with the parent class. The subclass is thus a more specialized form of the superclass.

When two classes are related by inheritance, the **is-a relationship** will apply to the classes. The is-a relationship holds between two classes when one class is a specialized instance of the second. For example, a `ProgrammableThermostat` *is a* `Thermostat`, and thus is an appropriate subclass. Similarly, a sports car *is a* car and a minivan *is a* car, so both would be appropriate subclasses of the superclass `Car`. A train is certainly not a car, and a car is not a train, so it would be inappropriate to place them in a subclass/superclass relationship. However, both cars and trains are forms of transportation, so the inheritance hierarchy in Figure 2-5 might be entirely appropriate.



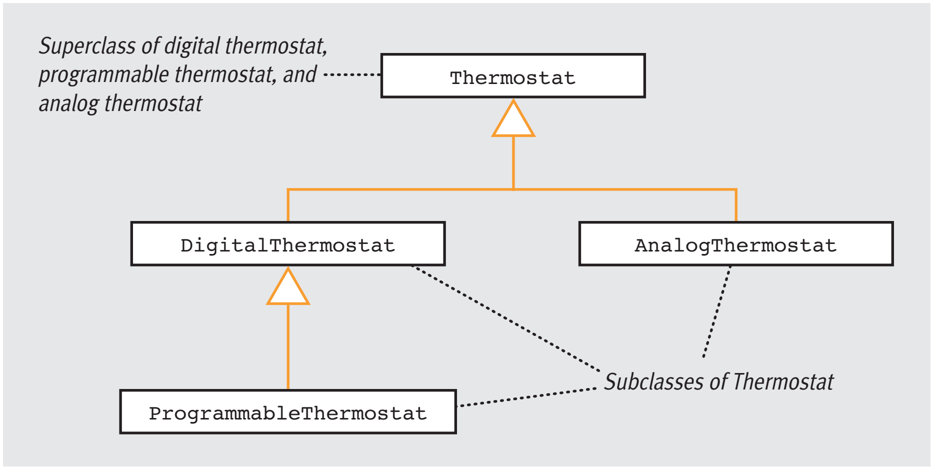
[FIGURE 2-5] Is-a relationship

The **has-a relationship**, on the other hand, holds when one class is a component of the second. For example, a car has a steering wheel, but a car is not a steering wheel. Therefore, it would not be appropriate to make a steering wheel a subclass of a car.

Because instances of a subclass contain all of the state and behavior associated with their superclass, an instance of a subclass can mimic the behavior of the superclass. The subclass should be indistinguishable from an instance of the superclass, and you can substitute instances of the subclass for instances of the superclass in any situation with no observable effect. In other words, a subclass automatically inherits the state and behavior of its parent. In Figure 2-4, any behavior exhibited by an instance of A is automatically available to instances of the classes B and C. Similarly, in Figure 2-5, anything a type of transportation can do (such as move people and carry things) can be done by both cars and trains, although not necessarily in the same way.

When used properly, inheritance allows a programmer to create a new class that exhibits slightly different behavior than its superclass. A side benefit of using inheritance is code reuse. For example, if I have developed some complex behavior to describe the operation of a car, it would be foolish and time-consuming to rewrite the same code for sports cars and minivans. Instead, by making these subclasses of a car, all sports car objects and all minivan objects will inherit this code and exhibit exactly the same behavior with no additional work. Inheritance is a way to increase programmer productivity via code sharing.

Inheritance allows us to organize classes into hierarchies based on their inheritance relationships. The class hierarchy of the thermostat classes discussed earlier is shown in Figure 2-6.



[FIGURE 2-6] Thermostat class hierarchy

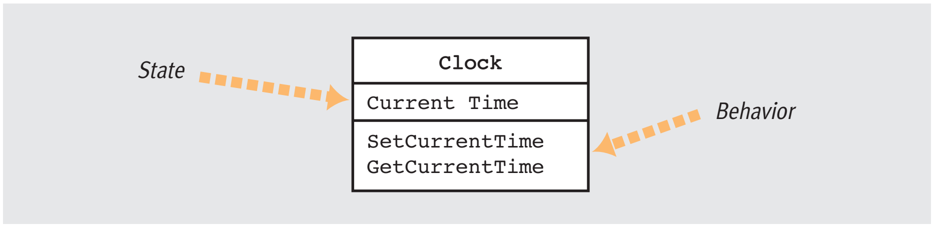
Inheritance is transitive, which means a subclass can inherit state and behavior from superclasses many levels away. In Figure 2-6, `ProgrammableThermostat` is a subclass of `DigitalThermostat`, and `DigitalThermostat` is a subclass of `Thermostat`; thus, `ProgrammableThermostat` is also a subclass of `Thermostat`. This means that a `ProgrammableThermostat` will inherit state and behavior from both the `DigitalThermostat` and `Thermostat` classes. Inheritance is perhaps the single most powerful tool provided by object-oriented programming languages.

Not surprisingly, inheritance can take different forms in an object-oriented program and can be used in different ways. Five of the most common forms of inheritance are summarized in Table 2-2.

FORM OF INHERITANCE	DESCRIPTION
Specification	The superclass defines behavior that is implemented in the subclass but not in the superclass; this provides a way to guarantee that the subclass implements the same behavior
Specialization	The subclass is a specialized form of the superclass but satisfies the specifications of the parent class in all relevant aspects
Extension	The subclass adds new functionality to the parent class but does not change any inherited behavior
Limitation	The subclass restricts the use of some behavior inherited from the superclass
Combination	The subclass inherits features from more than one superclass; this form is commonly called multiple inheritance

[TABLE 2-2] Five forms of inheritance

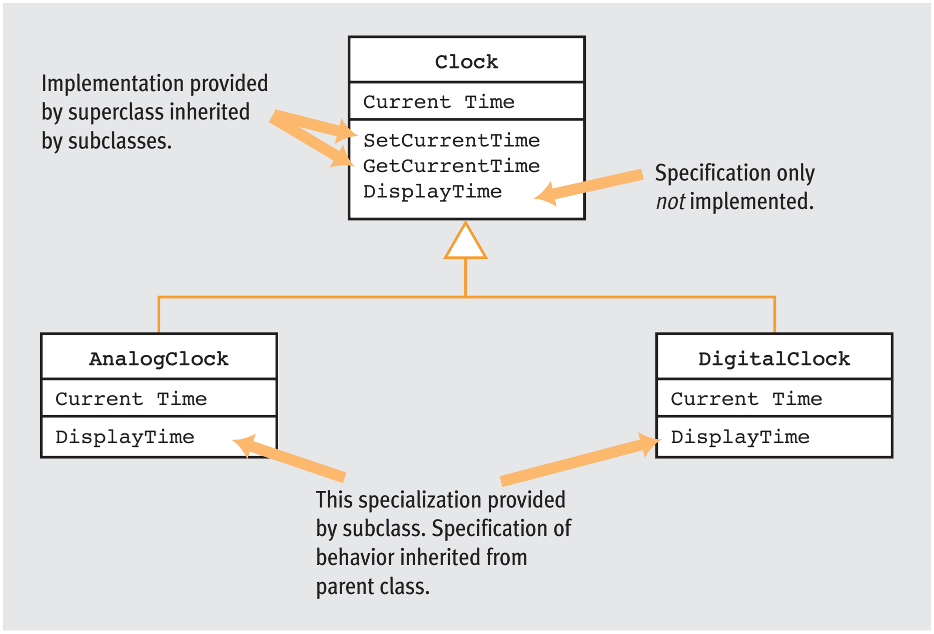
To make the differences between these five forms of inheritance more concrete, consider a class called `Clock` that defines the functionality of a simple clock. The state defined by `Clock` includes the current time and two behaviors that allow you to set the time and ask for the current time (see Figure 2-7).



[FIGURE 2-7] `Clock` class

The **specification** form of inheritance described in Table 2-2 specifies the required set of behaviors for a superclass that any of its subclasses must provide. When using this form of inheritance, the subclass inherits the specification of its superclass's behavior, and the subclass must provide the implementation for all of these behaviors. In a sense, the specification form of inheritance specifies what the subclass must do but not how to do it. This form of inheritance is useful when you have a highly generalized type of object, such as the class `Timepiece`. What is important to the superclass are the specific behaviors that every object must exhibit, not the details of the implementation. In essence, the subclass is only inheriting the behavioral specification. The specification form of inheritance holds that you cannot be considered a type of `Timepiece` or `Clock` unless you provide the behaviors that both determine and set the time—the two essential behaviors of all clocks. However, the details of how a particular type of clock either sets or determines the time are not as important and can be specified by the subclass.

The **specialization** form of inheritance is most easily defined in terms of the is-a relationship. The subclass is a more specialized form of the superclass. The specialization form of inheritance differs from the specification form in that the subclass inherits both the specification of a behavior and the implementation of some or all behaviors provided by the superclass. The parent states not only what you must do, but, for at least some of the behaviors, how you do it. This form of inheritance is useful when you implement a group of objects that are related in some way. For example, consider the classes `DigitalClock` and `AnalogClock`, both of which define clocks. They differ in how they display the time but not in how they update and store the time. Therefore, it makes sense to place the code required to implement common behavior in the superclass and let it be inherited by the subclass. Any subclass of `Clock` is then automatically able to keep track of the time; that is, it inherits the two behaviors "set current time" and "get current time" from the parent. How a clock displays the time is only specified in the superclass; it is implemented differently in the two subclasses, as shown in Figure 2-8.


Clock
Current Time
SetCurrentTime
GetCurrentTime
DisplayTime
Implementation provided by superclass inherited by subclasses.
Specification only *not* implemented.
AnalogClock
Current Time
DisplayTime
DigitalClock
Current Time
DisplayTime
This specialization provided by subclass. Specification of behavior inherited from parent class.

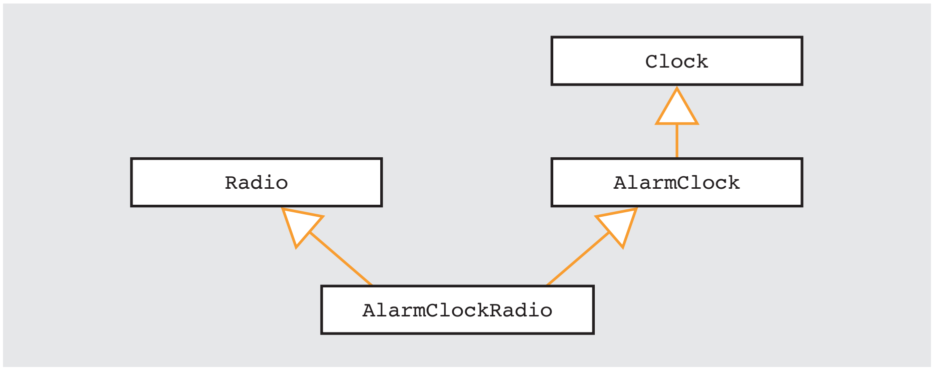
[FIGURE 2-8] Specialization form of inheritance

Inheritance can sometimes expand the existing function of a superclass. The **extension** form of inheritance is used to add totally new capabilities to the subclass. In the specialization form of inheritance, subclasses have the same behavioral properties as the superclass, but in each subclass, these behaviors may be implemented quite differently. In the extension form of inheritance, the subclass has behaviors that are not present in the superclass. For example, it is possible to purchase an atomic clock that automatically sets itself based on radio signals broadcast from Fort Collins, Colorado. These clocks are typically accurate to within a few milliseconds, and you do not have to adjust them when daylight saving time takes effect. An atomic clock's added ability to automatically set itself is an extension to the set of behaviors provided by a standard clock. You would not require every clock to provide this functionality.

When you think about the atomic clock, you realize that you might never need to provide a way to set its time manually. In fact, because an atomic clock can probably determine the time more accurately than any human, you may not want to permit humans to set the time by hand on an atomic clock. This is an example of the **limitation** form of inheritance; its strict definition requires that you remove a behavior that is inherited from a superclass to the subclass. Notice that this requirement violates the is-a relationship. If we remove the ability to set the time on an atomic clock, it is no longer a clock because it does not exhibit all the behaviors of the superclass. Therefore, when you use this form of inheritance, a behavior is not actually removed but is simply implemented as a "no operation." An atomic clock would still provide a set time method, but invoking that method would have no effect on the state of the clock.

In **single inheritance**, a subclass has at most one direct superclass from which it inherits behavior. **Combination** is a form of inheritance in which a subclass inherits directly from two or more superclasses.

Consider the inheritance relationship among the classes `Clock`, `AlarmClock`, `Radio`, and `AlarmClockRadio` in Figure 2-9. The `AlarmClockRadio` class has both `Radio` and `AlarmClock` as direct superclasses. This means that an `AlarmClockRadio` can act as both a `Radio` and an `AlarmClock` at the same time. The term **multiple inheritance** is commonly used to describe the combination form of inheritance in programming languages. Not all object-oriented programming languages allow multiple inheritance.



[FIGURE 2-9] Multiple inheritance

When used correctly, inheritance is a **technique** that can help solve some of the problems associated with developing extremely large software systems, such as those described in Chapter 1. For example, let's say you develop software for a company that analyzes the efficiency of residential home heating systems. Given that hundreds of types of thermostats could be used in a home, how can you design this software to analyze all of the thermostats on the market as well as new ones developed in the future?

One way to solve this problem is to create a generic thermostat class that defines the state and behavior associated with any thermostat the system might be expected to simulate. The software could then be written in terms of a generic thermostat class. When the time comes to add a specific thermostat model to the system, we would design a subclass for the model that inherits from the generic thermostat superclass. Because the program cannot distinguish between an object of the subclass and an object of the superclass, the software will continue to work properly.

This is an example of the specification form of inheritance described in Table 2-2, which provides a consistent interface for a group of classes within the system. It enables you to develop programs using standard software components, just as computer engineers build computers using standard hardware components. Once a thermostat class has been written, it is easy to reuse the class in a completely different program. You can also provide functionality in the generic thermostat class that all of its subclasses can use—the specialization form of inheritance. Finally, if this type of thermostat has unique features, we can use the extension and limitation forms of inheritance to customize its behavior.

Chapter 3 demonstrates how these inheritance concepts are realized in Java, and Chapter 4 implements a home heating system as a case study to illustrate the power and capability of object-oriented design.

This section discussed the basic concepts of object-oriented programs. These programs consist of a number of objects that have state, behavior, and identity and interact to accomplish some task. Objects are instantiated from classes, which are like cookie cutters in the sense that they describe the general structure of an object. Classes that define similar state and behavior can be organized into a hierarchical structure using subclasses and inheritance. A subclass automatically inherits the state and behavior of its superclass. Inheritance is an extremely powerful tool that can increase programmer productivity and help solve some of the problems associated with developing large software systems.

The next section examines the steps that precede the actual coding effort: the specification and design phases of the software life cycle, summarized in Figure 1-2. The section looks at how software is designed based on information in the user requirements and problem specifications described in Section 1.2.1. The chapter concludes by introducing the basics of the Unified Modeling Language (UML), a tool commonly used to document the design of object-oriented systems.

2.3 Object-Oriented Design

Your junior and senior years in high school were part of an important process that led you where you are today: the beginning of your college career. The process started when you and your guidance counselor discussed career options and worked to identify career paths that you might follow after high school. Your next goal was to determine exactly what you wanted to do and then identify colleges that would prepare you for the career path you selected. You had to determine what type of college you wanted to attend and whether you wanted to live on or off campus. At this stage, you were trying to determine what your educational experience would be like. You asked questions like: "Given my SAT scores, will I be accepted to this school?" and "How long will it take to earn my degree?" In a way, you were treating your future college experience as a computer program and were attempting to define the output of the process based on the input. In the end, you found a college that met your educational needs and prepared you for the career path you selected.

Software development, like selecting a college, is not a single step but a process consisting of many steps. A common misunderstanding among students is that software development begins and ends with programming. Nothing could be further from the truth. Writing code is a significant part of the software life cycle, but other important steps—the requirements, specification, and design phases—must take place *before* the coding effort can begin. These important steps were diagrammed in Figure 1-2.

During the **requirements phase** of the software life cycle, the user's needs are identified and documented. The **requirements document**, which is the end result of this effort, describes the behavior of the proposed software from the user's perspective. The focus of the requirements document is to describe what the program will *do* rather than explaining *how* the program will be implemented. The information in the requirements document is similar to the information you collected during the initial stages of your college search process. At that time, you were more concerned about determining your needs than how you would achieve them. For example, you may have known that you wanted to pursue a career in computer science, but you were not sure how you would attain this goal.

The **specification** phase of the software life cycle involves defining the explicit behavior of a program. The **specification document** produced during this phase focuses on identifying the specific outputs of the program for all possible inputs, as shown in Figure 1-3b. If a specific type of hardware environment, programming language, or algorithm must be used, that information is documented during the specification phase. The requirements document describes the desired effects the software should produce, and the specification document defines the behavior of the software that produces those effects. For example, during the requirements phase, we may learn that the user requires a program that can solve polynomial equations. This is the result that we want to achieve. During the specification phase, we describe the specific behavior to achieve that result. That is, the program will input the $n + 1$

coefficients of an nth degree polynomial as floating-point numbers and will output the n real roots of the equation to two decimal places. If some or all of the roots are imaginary, the specification document must describe how to handle them.

Taken together, the requirement and specification documents provide the information that the system analyst needs to design a program and solve the problem. Once the requirements and specification phases are completed, the **design phase** can begin. The design phase is guided by the information in the requirements and specification documents. The object-oriented design process focuses on identifying and defining the classes, the behaviors of each class, and the interactions that take place between the classes. The focus is on the functionality provided by the classes, not the details of how the classes are implemented. The code that implements the classes is written during the **implementation phase**. The design document serves as a guide to the programmers responsible for implementing the software.

Design is one of the most difficult, but most interesting, phases in the software development process. Learning and mastering a new programming language is challenging, but a programming language is only a tool used to realize a design. Consider, for example, the role of a hammer in building a new home. The hammer plays a crucial role in the construction process, but it is a very small part of the process. There are different types of hammers, and you need to pick the correct one for the task at hand, but the construction process is guided by the architect's blueprints, not by the hammer. The role of coding in software development is very similar to the hammer's role in the construction process. Coding is an important part of the process, and knowing how to code is an important skill, but it is not the central focus or driving force behind the development effort.

Several different designs for a software system can evolve from a single set of requirement and specification documents. The trick is to produce a design that allows programmers to develop software that is efficient, easy to build, and easy to maintain. This task is not as easy as you might think. You may find several good designs for a particular piece of software, each with advantages and disadvantages. You need to consider each design and objectively determine which one is right for the task at hand. Producing a design is a highly creative task, and there is no single correct answer, any more than an architect can create a single "correct home design" from a user's statement of likes and dislikes. The process of selecting the right design for a software system cannot always be clearly stated, and you often must rely on intuition or experience to determine which design is the best.

Even though this text focuses on implementation issues, it also provides some of the experience needed to master the art of software design. The case study in Chapter 4 illustrates how to derive a design from the requirements and specifications for a simple software system. When you read this case study, keep in mind that several different designs could have been used. Study the specific design presented and the decisions that were made. Then consider some of the alternative designs that were possible and discuss these alternatives with your instructor and other members of the class. This will give you experience with some of the basic steps in the software design process.

The next section of this chapter introduces a technique to identify the classes needed in a software system, based on the information in the requirements and specifications documents. The chapter concludes with a discussion of the Unified Modeling Language (UML), a graphical modeling language used to express and represent designs.

EXTREME PROGRAMMING (XP)

Most of us are familiar with the latest rage in outdoor activities, *extreme sports,* in which daredevils parachute off tall buildings, scuba dive into dark, water-filled caverns, or perform surfing-style acrobatics while falling from an airplane. One of the latest rages in computer science is *extreme programming*, abbreviated XP. Extreme programming is a new approach to software engineering and program design, developed by Kent Beck, Ward Cunningham, and Ron Jeffries in the late 1990s. It claims to take traditional software engineering practices to the "extreme" level.

XP is based on the concept of *agile programming*, also called *adaptive programming*, in which software is rapidly developed over many short time frames, called iterations, each of which may last only a week or two. Each iteration is virtually its own miniature software project. This allows programmers to quickly adapt to design and specification changes, compared with strategies in which a complete design is done first and then fixed for the life of the project. XP users claim this approach allows them to be more responsive to the rapidly changing needs of their customers. Many developers have become dedicated adherents of the XP philosophy, and it is becoming more popular as a software design and implementation strategy. XP is frequently studied in advanced courses in software engineering.

One of the most interesting things about this approach is that it is not simply a set of "best technical practices," but also a set of "best programmer welfare practices" that address how to reconcile the competing needs of productivity and humanity. For example, a basic principle of XP is "sustainable pace," which maintains that developers should not work more than 40 hours a week. If overtime is required in one week, the following week must not include overtime. XP adherents say that a tired programmer is a sloppy programmer who has less creativity and makes more errors.

XP is unique in how it treats software development not simply as a set of technical computing skills, but as a set of interpersonal skills drawn from the fields of management, psychology, and organizational behavior.

2.3.1 Finding Classes

The primary goals of the design phase are to identify the classes you will use in the system and to determine what relationships exist between the classes in the system. Making the correct decisions in this phase requires talent and experience. No algorithms can guarantee you will extract the best set of classes to use in your design based on the contents of the requirements document. The purpose of this section is to describe a simple technique that can help you identify which classes will be part of the final design.

The vast majority of classes needed in a system are actually identified in the requirements and specification documents. The trick is learning how to read these documents so you can identify the classes and their interrelationships. Objects, whether in a program or in the real world, are what we interact with to do work. In English, a noun is a part of speech that refers to an entity, quality, state, action, or concept. In an object-oriented view, nouns give names to objects. Therefore, the nouns that appear in the requirements and specification documents can help identify candidates for classes in our object-oriented solution. In the same sense, the verbs that appear in a requirements document often provide clues for which behaviors the classes should exhibit.

Recall the dining room description at the beginning of this chapter. The nouns that appeared in this description include *room*, *thermostat*, and *heaters*—the exact objects that make up the heating system discussed at the beginning of this chapter.

You probably noticed far more than three nouns in the dining room description. The words *wall*, *windows*, *ceiling*, and *door* also appeared, yet there are no wall, window, ceiling, or door classes in our home heating system software design. There is not a one-to-one relationship between the nouns in the requirements document and the classes that end up in the final system design. Using nouns to identify classes is a way to *start* the design process. The nouns in a requirements document provide a list of potential classes, but you must refine the list until you arrive at the set of classes that makes up your design.

Example 2.1: An Inventory Program

Consider the requirements and specifications for the inventory system shown in Figure 2-10. The nouns in the requirements section are printed in boldface.

Requirements:

Every **department** is required to maintain an **inventory** of the **items** for which it is responsible. For any **item** in the **inventory**, the **program** must be able to report the **name**, **serial number**, **cost**, and **location** of the **item**. **Users** must have the ability to

continued

add, delete, and locate any **item** contained in the **inventory**. The **program** provides a reporting feature that produces a **list** of all **items** in the **inventory** or a **list** of **items** for a single **location**.

Specifications:

- The information must be stored in an ASCII text file on a magnetic disk.
- The interface to the program must be menu driven.
- The name and serial number of an item may each contain up to 40 characters.
- The cost of an item ranges from $0 to $1000.
- Up to 15 users will be able to use the system simultaneously.

[FIGURE 2-10] Inventory system requirements and specifications

This description contains a total of 10 unique nouns: *department*, *inventory*, *item*, *program*, *name*, *serial number*, *cost*, *location*, *users*, and *list*. For each noun, we need to ask: Does this noun represent a class that is relevant to the design of the system and should be included in the software? Remember that our goal is to develop an object-oriented model for one program; we are not trying to create a model of the entire world. Clearly, the words *inventory* and *item* are central to the program because we are designing an inventory system to manage items. The term *department* also seems central to our software because departments maintain the inventory. What about the word *users*? Certainly, a user is necessary to run the program, but a user does not need to be modeled by our software, so we can remove it from our list.

What about the word *program*? Users will run the program, and the system will be packaged as a program, but like the word *users,* we do not need to include it in our design. Finally, we can eliminate the word *list* because it refers to the physical output that users will obtain if they use the reporting feature of the system. The words *name*, *serial number, cost,* and *location* refer not to unique classes but to the state of an item, so we can also remove them from our list of candidates.

That leaves us with three potential classes: department, inventory, and item. Based on the description of the problem, this system must contain at least two classes: `Inventory` and `Item`. The `Inventory` class will act as a repository of items, and information about each physical item will be stored as state information in the `Item` class. What about `Department`, however? Is it a separate class or part of the state of an item? If `Department` is a class, then an inventory object (along with the items in that inventory) would be associated with the department. For example, the item "bowling shirt" would be associated with the inventory of the Sporting Goods Department. This might make it difficult for users to determine if a particular item is contained in the inventory of another department, assuming that users only have access to their department object. For example, someone in the Men's Clothing Department would be able to look up the price of a shirt, but not the price of a bowling shirt, because that item is in the inventory of a different department. Making department information part of the state of an item would mean that one inventory object could be used to store all the items owned by the corporation.

Which is the correct approach? Based on the information in Figure 2-10, we cannot decide, and if we were creating this design in real life, we would need to consult with the store owners to determine how they want the program to function, modify the requirements, and then select the appropriate design. For purposes of this discussion, assume that department information is part of the state information of an item, so our final design will consist of two classes: `Inventory` and `Item`.

Now that we have identified the classes, how do we define the state and the behavior for each of them? As you have seen, the state of an object may be derived from some of the nouns that appear in the requirements. We know that an `Item` object has the following state: department, name, serial number, cost, and location. We have also decided that an `Inventory` object will have a collection of items as part of its state. What about the behavior of these classes? The `Item` class is fairly straightforward because its role in this system is to maintain the information associated with each item. The only requirements are probably accessor and mutator methods that allow users to obtain and change these values. For example, `getDepartment()` and `setDepartment()` methods would allow users to obtain or modify the department where the item is located.

The `Inventory` class is a different story, and the clues for its behavior come from the verbs in the requirements document that describe which operations users perform on the inventory. The document states, "Users must have the ability to add, delete, and locate any item contained in the inventory." This implies that an `Inventory` object would have the methods listed in Table 2-3 to make these operations possible. Additional behaviors would be required to list the contents of the inventory.

METHOD	DESCRIPTION
`void add(SerialNumber sn, Item it)`	Add the item `it` identified by the serial number `sn` to the inventory
`boolean delete (SerialNumber sn)`	Delete the item identified by the serial number `sn` from the inventory; return true if the item was deleted and false if the item is not found
`Item locate(SerialNumber sn)`	Locate the item identified by the serial number `sn`; return a reference to the item if it is in the inventory or null if the item is not found

[TABLE 2-3] `Inventory` methods

We have presented the criteria for selecting nouns to identify classes that will appear in the final design. Because this process is not exact, it might be helpful to discuss criteria used to eliminate nouns from consideration.

The principle of object-oriented programming is to build software based on objects rather than functionality. A common mistake is calling something a class when it is really a function or behavior of a class. Classes whose names end with *er*, such as `Lister` or `Finder`, are a good example of this type of mistake. A class is not supposed to take one particular action; it provides a set of behaviors that act on or modify the state of the object. If a class only does one thing, it is probably not a class, but a behavior that should be associated with some class in your design.

Example 2.2: A Banking System

The requirements and specifications for a simple banking system are shown in Figure 2-11. As before, the nouns in the requirements section are printed in boldface.

Requirements:

A **bank** offers **customers** three types of **accounts**: **checking**, **savings**, and **money market**. **Checking accounts** pay 3 percent **interest**, **savings accounts** have a minimum **balance** of $150 and pay 5 percent **interest**, and **money market accounts** have a minimum **balance** of $1000 and pay 10 percent **interest**. The **system** is to provide a **report** feature that lists all the **accounts** and the **interest** they have accrued during the current **period** (principal * interest / 12). The **system** will also provide an **auditor** feature that, when invoked, prints all **accounts** whose **balance** falls below the minimum required for that type of **account**.

For each **account**, the **system** will record the **account number**, **name**, **current balance**, and **interest rate**.

Specifications:

- Account numbers contain only numeric characters and are exactly 10 characters in length.
- Names may contain up to 30 characters.
- Balances may range from $0 to $1,000,000.
- There will never be more than 2500 accounts in the bank.
- The user will interact with the system using a graphical user interface.

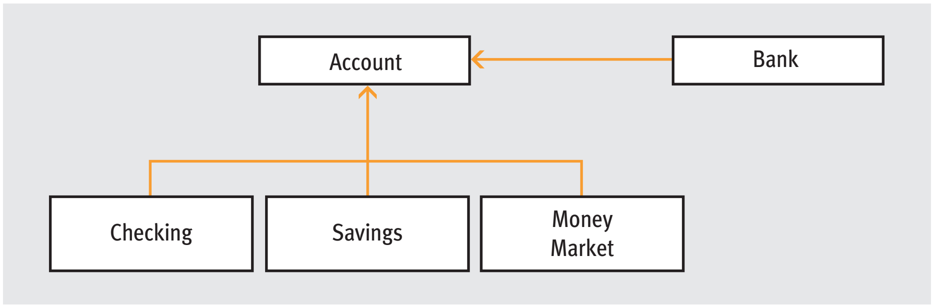
[FIGURE 2-11] Banking system requirements and specifications

Clearly, the purpose of this system is similar to that of the inventory system of Figure 2-10, but instead of storing items, the bank stores accounts. We can reuse some of the ideas developed in the design of the inventory system. As before, one class, `Bank`, will be responsible for storing all of the account information. Like the `Inventory` class from the previous example, the `Bank` class will allow you to add, delete, and locate accounts, as shown in Table 2-4.

METHOD	DESCRIPTION
`void add( String an, Account a)`	Add the account `a`, identified by the account number `an`, to the bank
`boolean delete (String an)`	Delete the account identified by the account number `an` from the bank; return true if the account was deleted and false if the account was not found
`Account locate (String an)`	Locate the account identified by the account number `an`; return a reference to the account if it is in the bank or null if the account is not found

[TABLE 2-4] Bank methods

So far, everything seems reasonable except that the bank has three different types of accounts: checking, savings, and money market. In other words, a checking account is-an account, a savings account is-an account, and a money market account is-an account. The only differences are their interest rates and the minimum balances required. Because the specific account types are only specialized forms of an account, it seems reasonable to use the concept of inheritance first presented in Section 2.2.3. The superclass `Account` will provide the attributes (account number, name, current balance, and interest rate) and behaviors (accessors and mutators) common to all accounts. The specialized rules regarding minimum balances and the computation of interest will be delegated to the subclasses that define an account type. The `Bank` class will then store `Accounts` and not worry about the specific types of accounts it stores (see Figure 2-12).



[FIGURE 2-12] Banking system design

Consider how the design of the banking system would change if you were asked to add a no-interest checking account to the system. Using inheritance, this would be an easy task; you would only have to create a new subclass that captures the details of the new account type. All of the other standard behaviors will be inherited directly from the superclass `Account`. Because the new account type would be an `Account`, the `Bank` class would already know how to work with the new class. Thus, the only modification to the existing code would be the addition of the new class. In a procedural language, the changes would be much more substantial.

The last design issue you need to deal with is the auditor feature. You might be tempted to create an `Auditor` class that would go through all the accounts stored in the bank and check to see if they satisfy the minimum requirements. The problem with this decision is that you would be creating a class that consists of a single method and whose primary purpose is to provide functionality. This is exactly the problem discussed earlier in this section. A better way to deal with the auditing feature is to provide an `audit()` method in the `Bank`, `Account`, `Checking`, `Savings`, and `MoneyMarket` classes. The `audit()` method in `Bank` would step through the accounts it manages, invoking the specialized audit methods for each type of account, one by one. The `audit()` method in the `Checking`, `Savings`, and `MoneyMarket` classes would verify that the minimum requirements were met for each account type and, if not, print an appropriate message. Note that the end result is the same for both auditor designs. The difference is that the second design is much easier to extend. Once again, ask yourself what is needed to add a new account type. For the second design, you only need to add a new subclass to the system that describes the new account and make sure that it includes its own specialized audit method. In the first design, you must not only add a new subclass, you must also remember to modify the `Auditor` class to handle the new account type.

One mark of a good design is that when a change is made to the specification of the problem, the design change is localized to a single class rather than propagating through many classes. When you find yourself saying things like, "We must add a new subclass A and modify classes B, C, and D to deal with this new subclass," you greatly increase the likelihood for error and complicate the maintenance process.

By working through these two designs, you should have an idea of how to start designing an object-oriented system. Clearly, a few pages cannot give you all the information and experience you need to build large, robust systems. Another problem that these two examples highlighted is that a natural language—English in our case—is entirely too ambiguous to precisely describe the design of software. In the next section, we introduce the Unified Modeling Language (UML), a tool that accurately describes the design of object-oriented systems.

2.3.2 Unified Modeling Language (UML)

A software design is useless if it cannot be clearly, concisely, and correctly described to the programmers writing the code. In the construction industry, architects use blueprints to describe their design to the contractors who construct the building. Software architects need a similar form of “software blueprints”—that is, a standard way of clearly describing the system to be built. This section introduces the **Unified Modeling Language (UML)**, a graphical language used to visually express the design of a software system. UML provides a way for a software architect to represent a design in a standard format so that a team of programmers can correctly implement the system.

Prior to the development of UML, software architects used many different and incompatible techniques to express their designs. Because no method was universally accepted, it was difficult for software architects to share their designs with each other and, more importantly, with the programming teams responsible for implementing the software. In 1994, Grady Booch, James Rumbaugh, and Ivar Jacobson started work on UML. One of their goals was to provide a unified way of modeling any large system, not just software, using object-oriented design techniques. For UML to be widely used, it was important that the language be available to everyone. Therefore, the resulting language is a nonproprietary industrial standard and open to all.

Several aspects of a system must be described in a design. For example, the functional aspects of a system describe the static structure and the dynamic interactions between system components, whereas nonfunctional aspects include items such as timing requirements, reliability, or deployment strategies. To describe all of the relevant aspects of a software system, UML provides five different views. A **view** consists of a number of diagrams that highlight a particular aspect of the system. Four of the views provided by UML are summarized in Table 2-5.

VIEW	DESCRIPTION
Use-case	Describes the functionality that the system should deliver as perceived by external actors (users); documents system requirements and specifications
Logical	Illustrates how the system’s functionality will be implemented in terms of its static structure and dynamic behavior
Component	Shows the organization of the code components
Deployment	Illustrates the deployment of the system into physical architecture with computers and devices called nodes

[TABLE 2-5] UML views

In addition to the four views of Table 2-5, UML defines nine different **diagram types** that describe specific aspects of the system. Because a single diagram cannot possibly capture all the information required to describe a system, each separate UML view includes several diagrams. Table 2-6 lists the nine types of UML diagrams, a brief description of each, and the view in which each diagram is typically used.

DIAGRAM TYPE	DESCRIPTION	VIEWS
Use-case	Captures a typical interaction between a user and a computing system; useful when defining the user's view of the system	Use-case
Class	Describes the classes that make up a system and the various kinds of static relationships that exist among them	Logical
Object	A variant of the class diagram, except that an object diagram shows a number of instances of classes instead of the actual classes	Logical
State	Describes all the possible states of an object and how the object's state changes as a result of messages sent to it	Logical, concurrency
Sequence	Describes how a group of objects collaborates in some behavior, concentrating on the messages sent to elicit that behavior	Logical, concurrency
Collaboration	Describes how a group of objects collaborates in some behavior, concentrating on the static connections between the objects	Logical, concurrency
Activity	Shows a sequential flow of activities performed in an operation	Logical, concurrency
Component	Describes the physical structure of the code in terms of code components	Concurrency, component
Deployment	Shows the physical architecture of the software and hardware components that make up a system	Concurrency, deployment

[TABLE 2-6] UML diagrams

UML is a powerful tool that has many features, and it can express very complicated designs. In this text, we are only interested in the logical view of a system (the static structure and dynamic behavior). Therefore, we will make extensive use of class, object, state, and sequence diagrams when expressing our designs.

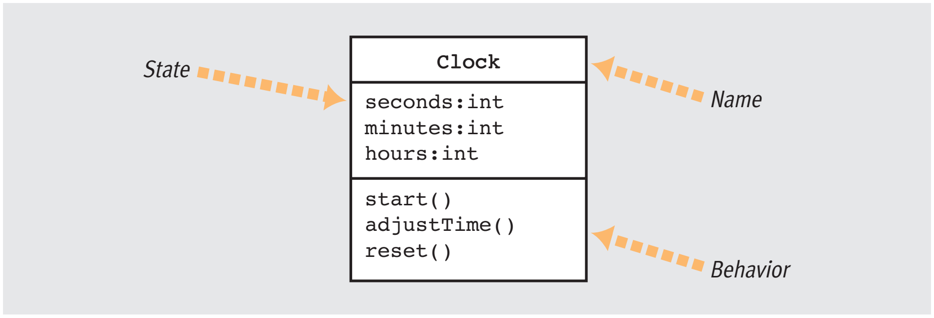
Unlike a programming language, UML does not have rigid rules for what must be included in a diagram, and it allows the software architect to determine how much detail to include in the final diagram. When developing your designs, keep in mind what you are trying to illustrate and who will use your diagrams. For example, a programmer requires detailed information about an object's state and behavior, but a system analyst may only require a general description of the classes that make up the system. Your diagram should provide enough information to illustrate your design but not so much detail that a reader gets lost. The structure of the UML diagrams should be based on the readers' needs.

2.3.2.1 Class Diagrams

A class diagram in UML (see Table 2-6) describes the static structure of a system in terms of its classes and their relationships. Class diagrams are the most common way to describe the design of an object-oriented system, and you will use them all the time. Class diagrams, like UML, are very expressive and provide ways to describe even the subtlest aspects of a class. To avoid becoming lost in the details, we will only describe the more commonly used features of class diagrams. As you gain more experience, you should read more about their advanced features, as they can be very useful. There are many excellent references on UML.

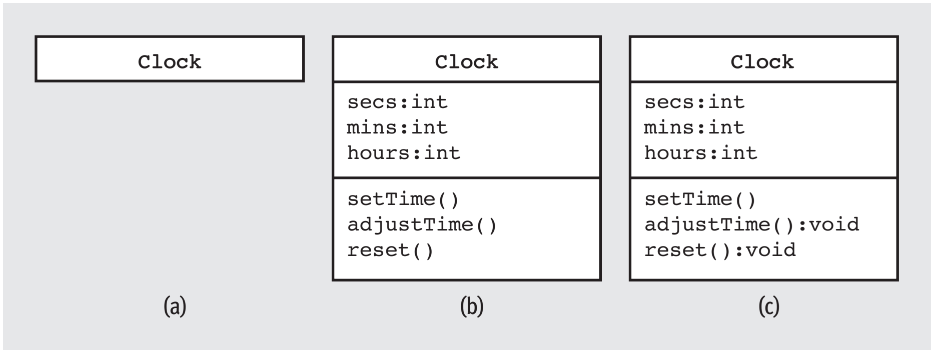
It may not be possible or even desirable to use a single class diagram to describe a complete system. It is better to concentrate on key areas of the design and then document these ideas with different class diagrams. Keeping the diagrams simple and using them to convey key concepts of a design can be much more effective than using the "shotgun" approach of describing everything in as compact a space as possible.

A class in a class diagram is drawn as a rectangle that can be divided into three compartments (see Figure 2-13). The name of the class appears in bold text, centered in the compartment at the top of the rectangle.



[FIGURE 2-13] `Clock` class diagram

The compartments that describe the state and behavior of the class are optional, as shown in Figure 2-14a. Type information for the methods that make up the behavior of the class is also optional; however, the names of routines that take parameters must be followed by opening and closing parentheses, even if you put nothing inside them. You can specify parameter and return types for behaviors using the colon notation shown in Figures 2-14b and 2-14c.



[FIGURE 2-14] UML model of a `Clock` class

When drawing a class diagram, you want to include only as much information as a reader needs to understand your design. Do not overwhelm a reader with trivial detail when it is not required. For example, most class diagrams do not contain obvious behaviors, such as accessors or mutators, so that the less obvious behaviors in the class are easier to recognize. Similarly, we may omit such relatively unimportant details as void return values, as we did in Figure 2-14b. Classes whose behaviors are well known, such as classes provided in a system library, will either be drawn as a single rectangle with no compartments, as in Figure 2-14a, or omitted from the diagram altogether.

Describing the classes in a software system does not provide enough information for a programmer to understand how these classes work together in a program. For example, if you were asked to describe a family, it would not be enough to say, "A family consists of parents and children." You would need to provide additional information to describe the nature of the relationship between parents and children. To completely specify the static structure of a system, you must define the classes and the nature of the relationships between them.

A relationship exists between two classes if one class "knows" about the other. In most object-oriented programming languages, a relationship exists between two classes if an instance of one class invokes a method or accesses the state of an instance of the other class. Knowing that a relationship exists between two classes does not provide any information about the *nature* of the relationship. For example, the relationship between the parents in a family is considerably different from the relationship between the parents and the children. UML defines several types of relationships that can describe how two or more classes are related in a class diagram. This text uses three different types of relationships: associations, dependencies, and generalizations.

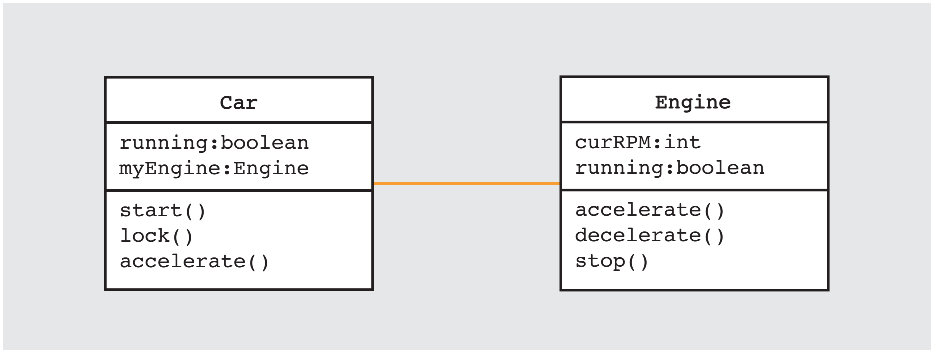
An **association** is a relationship that ties two or more classes together and represents a structural relationship between instances of a class. An association indicates that objects of one class are connected to objects of another. This connection is permanent and makes up part of the state of one of the associated classes. From a programming perspective, two classes are associated if the state of one class contains a reference to an instance of the other class.

In the home heating system, for example, relationships exist among the thermostat, heater, and users of the system. The relationship between the thermostat and heater makes up the structure of the system. As long as the heating system exists, the thermostat will know about the heater. This type of relationship, which represents a permanent structural relationship between two classes, is classified as an association. The relationship between the thermostat and heater is clearly different from the relationship between a user and the thermostat.

A user is an important part of the system and uses the thermostat to adjust the room temperature; however, a user is not part of the system structure nor is the system always associated with a user. The heating system in my dining room continues to function whether I am in the room or not.

For another example of association, consider the relationship between instances of the `Engine` and `Car` classes in an automotive simulation program. The relationship between instances of the `Engine` and `Car` classes forms part of the structure of a `Car`. As long as a car exists, the car will know about, or be in a relationship with, the engine. This is not true of the relationship between the `Car` and `Driver` classes. At night when I am sleeping and my van is in the garage, my van can still function as a car. On the other hand, if someone breaks into my garage and removes the engine from my van, it will no longer function as a car. Therefore, although relationships exist among these classes, there is an association between the `Engine` and `Car` classes but not between `Driver` and `Car`.

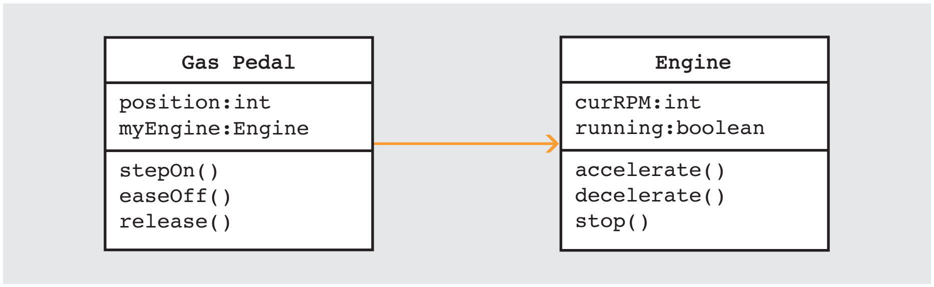
In UML, a solid line is drawn between two classes to represent an association. The UML class diagram in Figure 2-15 specifies that the relationship between the `Car` and `Engine` classes is an association.



[FIGURE 2-15] UML association

The association between the `Car` and `Engine` of Figure 2-15 goes in both directions; the car knows about the engine and the engine knows about the car. Associations, however, are not always bidirectional. Consider the relationship between the `Engine` and `GasPedal` classes. This relationship can be described as an association because it is permanent and is part of the car's structure. Unlike the association between the `Car` and `Engine` classes, the association between the gas pedal and the engine does not go in both directions. The gas pedal knows about the engine because it invokes the engine's `accelerate()` method; however, the engine does not know about the gas pedal because it never invokes a method on that class.

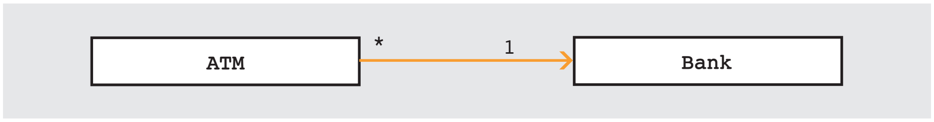
Navigability information can be included in a UML class diagram to clarify the nature of the relationship between two classes. As shown in Figure 2-16, an arrow is added to the solid line that represents an association to indicate the *direction* of a relationship. The arrow indicates that the gas pedal knows about the engine, but the engine does not know about the gas pedal.



[FIGURE 2-16] Adding navigability information to an association

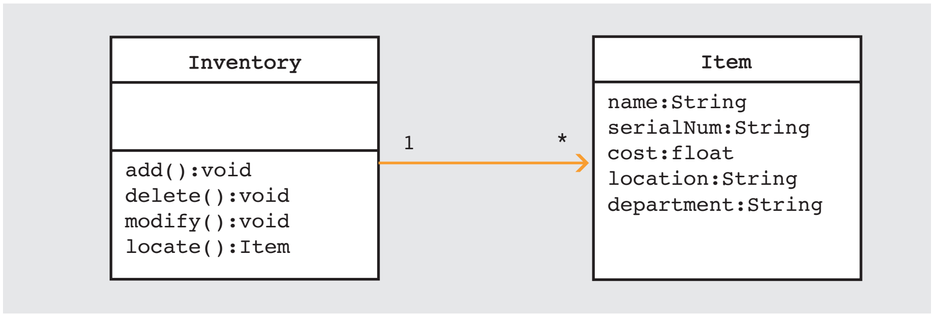
In addition to navigability, **multiplicity** can be used to describe the nature of a relationship between classes. Consider the relationship between the automated teller machine (`ATM`) and `Bank` classes in an electronic banking system. This relationship is an association because it is structural and permanent (in other words, the ATM always needs to know about the bank). Furthermore, the association is one way because the ATM knows about the bank (in other words, invokes methods on it), but the bank does not invoke methods on the ATM. However, the ATM class will probably have several instances, whereas the `Bank` class has only one instance. This last bit of information can be included in a UML class diagram by adding multiplicity information to the association.

Multiplicity information has been added to the UML diagram in Figure 2-17. The * indicates that the `ATM` class may have zero, one, or more instances. The 1 next to the `Bank` class indicates that the system has exactly one bank.



[FIGURE 2-17] Multiplicity information added to an association

Let's use association, navigability, and multiplicity to model the inventory system described in Figure 2-10 and designed in the previous section. Recall that the design consisted of two classes: `Inventory` and `Item`. The `Inventory` class was responsible for storing items and providing methods to add, delete, locate, and modify items in the collection. The `Item` class captured the state associated with the items owned by each department. The UML class diagram, shown in Figure 2-18, models this design.



[FIGURE 2-18] Inventory system

The class diagram in Figure 2-18 includes a description of the two classes, `Inventory` and `Item`, that make up the inventory system. Because the behaviors associated with the `Item` class consist exclusively of accessors and mutators (for example, `getName()` and `getCost()`—the types of methods you would expect to find in such a class)—they have been omitted from the diagram for clarity. Furthermore, the type of data structure used to implement the `Inventory` class will not affect the system's design, so it has been omitted to make the diagram easier to understand.

The relationship between the `Inventory` and `Item` classes is drawn as an association because it describes the structure of the system (i.e., the inventory contains items). In this case, the inventory will not invoke methods of the `Item` class, but the inventory clearly must know about the items. The navigation information specifies that the inventory knows about the items, but that the items do not know about the inventory. Finally, the multiplicity information specifies that a single inventory will hold zero, one, or more items and that the system consists of exactly one inventory object.

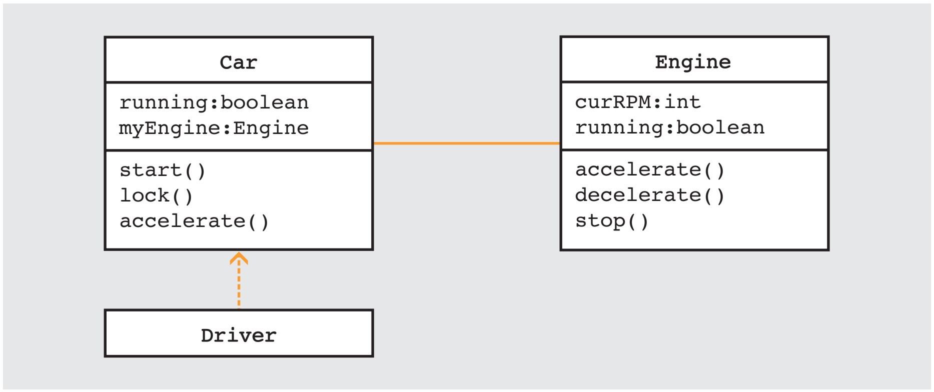
The second type of relationship that we use in this text is called a **dependency**. A dependency relationship means that a change in one class may affect another class that uses it. A dependency is inherently a one-way relationship in which one class is dependent on the other. Returning to the automotive simulation program, a relationship clearly exists between the `Car` and `Driver` classes, but this relationship should not be classified as an association because it does not constitute part of the system's structure. It would be more descriptive to indicate that the driver uses the car, because if the `Car` class changes (perhaps the car no longer has an automatic transmission), the driver may need to change in order to use the car.

To clarify the difference between an association and a dependency, look at the way these relationships are implemented in a program. Associations are usually implemented as part of the state of a class. For example, in the code that implements the state of the

`GasPedal` class, you would expect to find a variable that refers to the engine object the gas pedal controls. As long as the car exists, a gas pedal object will always be associated with a specific engine object. The state variable provides a mechanism for the gas pedal to access the engine.

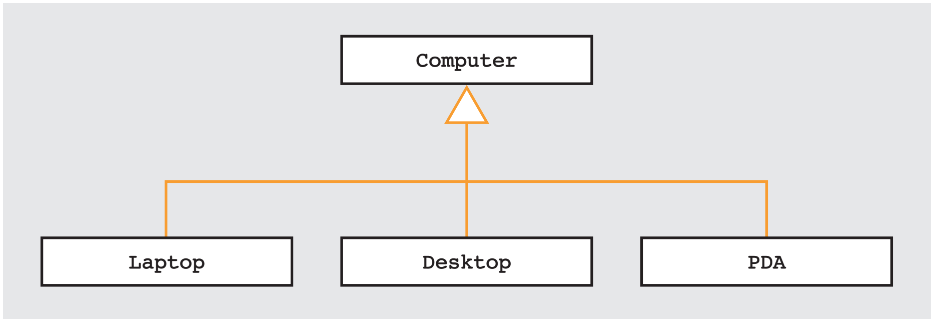
A dependency typically takes the form of a local variable, parameter, or return value in the methods that implement an object's behavior. These types of variables are often referred to as **automatic** because they are created and destroyed as needed. They do not represent a permanent structural relationship. If the variable is in scope (in other words, if it exists), one class has a way to access another class it depends on. So, at some point in the lifetime of the object, it may know about an instance of a class with which it related, and at other times it will not. This reflects the transient, or nonpermanent, nature of a dependency. In an automotive simulation program, you would expect the driver to have a `driveCar()` method that takes as a parameter a reference to the car. The car is clearly not part of the state of the driver. Furthermore, a driver only knows about a specific car when actually driving it. When the driver is finished with the car, the relationship ends.

In a UML class diagram, a dependency is drawn as a dashed line. The arrow points to the independent element. In Figure 2-19, the arrow captures the fact that if the car changes, the driver may need to change, but if the driver changes, the car is not affected.



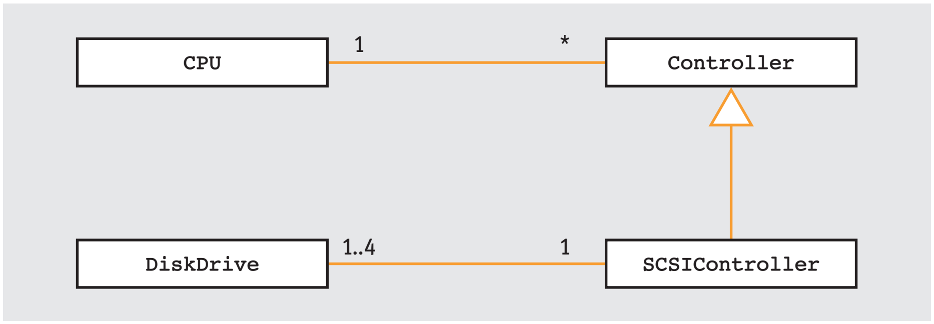
[FIGURE 2-19] UML association and dependency

Generalization, or inheritance, is the third type of relationship we will use in this text. In an inheritance relationship, a subclass is a specialized form of the superclass. If you look at the relationship from the superclass' perspective, you could say that the superclass is a generalization of its subclasses. This generalization relationship is denoted by a triangle that connects a subclass to its parent class. The triangle is connected to and points at the parent class, as shown in Figure 2-20.



[FIGURE 2-20] Generalization (inheritance)

Both associations and generalizations can be illustrated in a single diagram, as shown in Figure 2-21. This diagram describes the relationships between a processor, disk controller, and disk drive. The CPU is associated with the class `Controller` that describes all controllers. `SCSIController`, a subclass of `Controller`, is the class associated with `DiskDrive`. In a class diagram, associations should only be shown at the highest possible level. For example, in Figure 2-21, the association is between `CPU` and `Controller` (the superclass), not between `CPU` and `SCSIController` (the subclass). This indicates that `CPU` is designed to work with any `Controller`, not just a `SCSIController`.



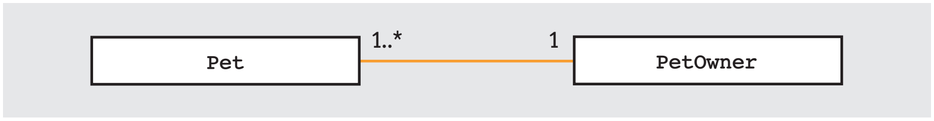
[FIGURE 2-21] Simple computer system

2.3.2.2 Example Class Diagrams

The best way to master the basics of class diagrams is to use them to model simple systems. This section presents four UML class diagrams, along with a description of the system that each diagram models. To improve your understanding of class diagrams, take a few minutes to review each one before reading the description of the diagram in the text. Write a description of the system being modeled. Then read the description of the system that follows and reconcile your description with that in the text.

Example 2.3: Veterinary System

The first class diagram models two classes in a system a veterinarian uses to track patients (see Figure 2-22).



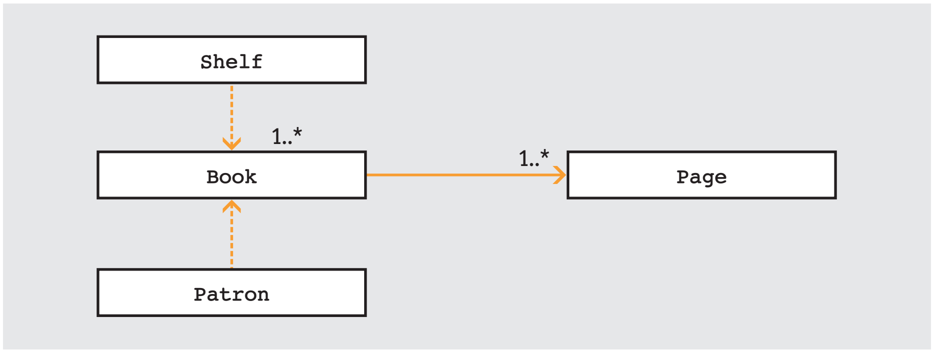
[FIGURE 2-22] Pets and pet owners

`Pet` and `PetOwner` are two of the classes in the veterinarian's animal tracking software system. The class `Pet` includes the state the system maintains for a single animal, and the class `PetOwner` contains the state associated with the owner of a pet. In the UML class diagram of Figure 2-22, the state and behavior for these classes have been omitted because the only item of interest is the relationship between the two classes. The state of every pet object contains a reference to its owner. Because this relationship provides structural information and instances of the `Pet` class will always contain a reference to a pet owner, the relationship is modeled as an association.

The absence of navigability information in the drawing indicates that the association is bidirectional. In other words, a pet knows about its owner, and a pet owner knows about its pet. Finally, the multiplicity information shows that every pet has exactly one owner, and every pet owner has one or more pets. If the multiplicity information was omitted, nothing could be said about the number of owners associated with a pet and the number of pets associated with a pet owner.

Example 2.4: Library System

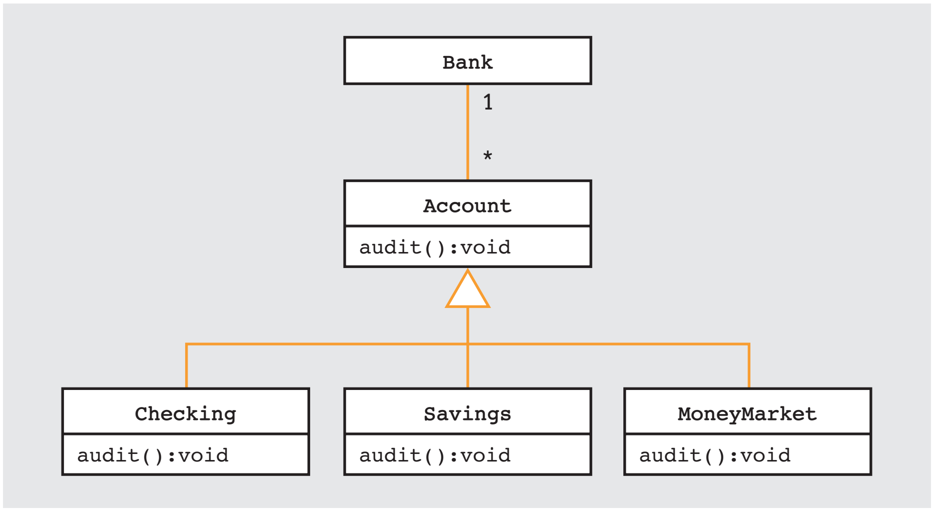
The class diagram in Figure 2-23 describes the relationships that exist between books, pages of a book, shelves, and the patrons of a typical library. This diagram indicates that a book contains one or more pages. This relationship has been modeled as an association because the pages are actually part of the book. If we rip the pages out of the book, it is no longer a book. Note, however, that whether the pages are in the book or not, they are still pages. This is why the solid line that specifies the association between the book and page classes has an arrow that points to the `Page` class. Compare this to the relationship between the `Shelf` and `Book` classes. Clearly, a book is not part of a shelf, and a shelf is not part of a book; however, should the properties of a book change, the shelf may have to change to accommodate the book. In this relationship, therefore, the `Shelf` class is dependent on the `Book` class. The arrow specifies the independent class (meaning that the class does not have to change).



[FIGURE 2-23] Library

Example 2.5: Banking System

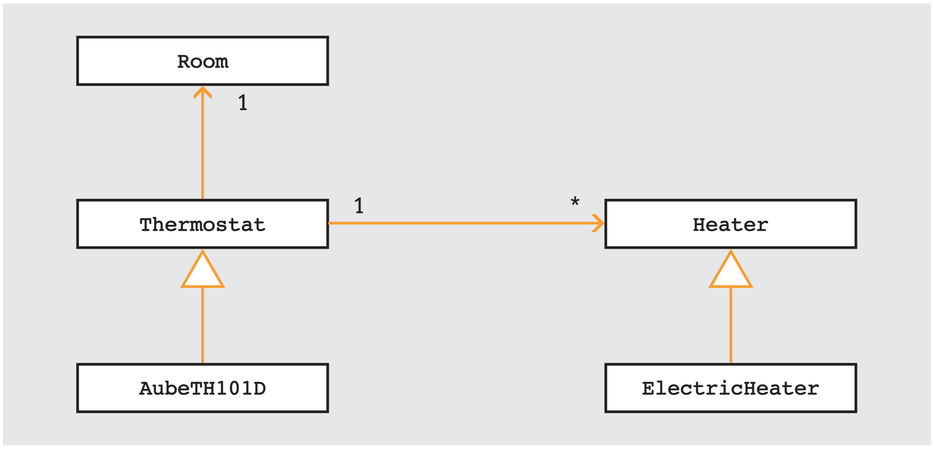
In this system (see Figure 2-24), a single bank is associated with, or has, one or more accounts. The lack of navigability information indicates that the association goes in both directions. Clearly, the bank knows about its accounts, but the accounts also know about the bank. This means that an account can use the services provided by its bank, such as obtaining the current interest rate set by the Federal Reserve Bank. An account is a generalization of three classes: `Checking`, `Savings`, and `MoneyMarket`. The `audit()` method is defined in the `Account` class and is a behavior that all of its subclasses will have.



[FIGURE 2-24] Banking system

Example 2.6: Home Heating System

Finally, this section would not be complete without Figure 2-25, which contains a UML class diagram that models a simple home heating system.



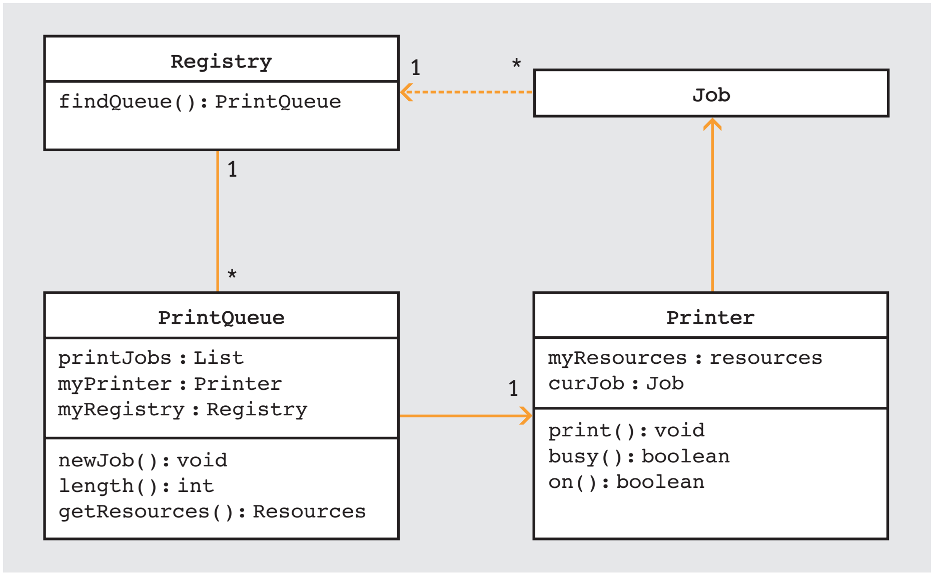
[FIGURE 2-25] Home heating system

UML class diagrams are an excellent tool for capturing the static aspects of a system—the classes in the system and the relationships between classes. What we have not discussed yet is how to capture the dynamic aspects of a system. Using a class diagram, it is not possible to document that in the home heating system of Figure 2-25, the thermostat queries the room for the current temperature and turns on the heaters if necessary. A class diagram can show that a thermostat knows about a room and knows about the heaters, but it cannot say anything about what specific interactions the thermostat has with these items. The next section introduces UML sequence diagrams, which describe the dynamic behavior of an object-oriented system.

2.3.2.3 Sequence Diagrams

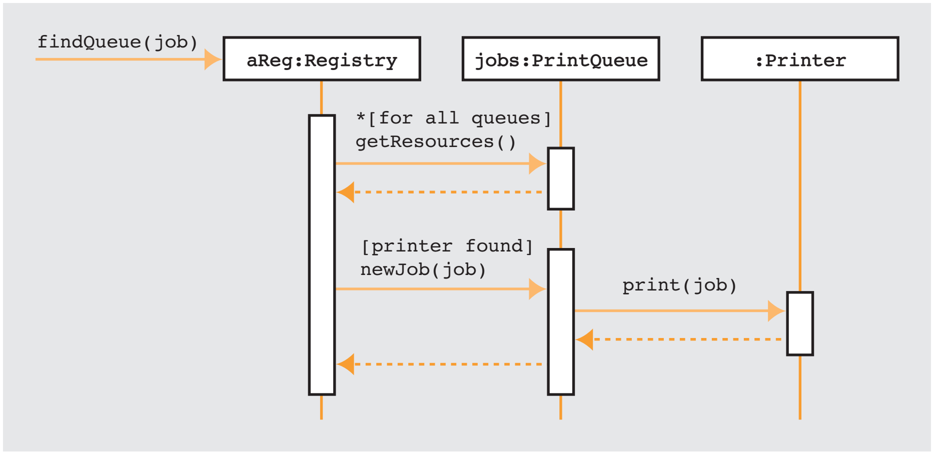
Sequence diagrams describe how groups of objects dynamically collaborate with each other in a system. Typically, a single sequence diagram describes a single behavior. It diagrams a number of objects and the messages passed between these objects.

To illustrate some basic features of sequence diagrams, this section models the behavior of a printing system. The system consists of a number of printers, each of which has different resources: the size and type of paper it is holding or the ability to print in color. Each printer is serviced by a queue that holds its print jobs. The printer registry maintains information about printer resources and the location of the queues that service the printers. The class diagram for the system is shown in Figure 2-26.



[FIGURE 2-26] Printing system

Although the class diagram in Figure 2-26 explains the static structure of the printing system, it does not explain how the objects collaborate to achieve a specific behavior. For example, a programmer needs to ask how a job finds a printer with exactly the set of resources it requires. The answer is that when a job needs to be sent to a printer, a message `findQueue()` is sent to the registry; the message contains the job to be printed and a list of required resources. The registry then searches each of its print queues for the first printer that has the resources required to print the job. If such a printer exists, the registry sends a `newJob()` message to the print queue, which in turn sends a `print()` message to the correct printer. This process is described in the sequence diagram in Figure 2-27.



[FIGURE 2-27] Sequence diagram for the printing system of Figure 2-26

Time flows from top to bottom in a sequence diagram, and the vertical lines that run down the diagram denote the lifetime of an object. Because a sequence diagram illustrates objects as opposed to classes, it uses a slightly different labeling convention. Each line is labeled with the name of the object, followed by the name of the class from which the object is instantiated. A colon separates the name of the object from the name of the class. It is not necessary to supply names for all of the objects; however, the class name must always be given. When labeling a line using only a class name, the class name must be preceded by a colon.

A vertical rectangle shows that an object is active—that is, handling a request. The object can send requests to other objects, which is indicated by a horizontal arrow. An object can also send itself requests, which is indicated with an arrow that points back to the object. The dashed horizontal arrow indicates a return from a message, not a new message. In Figure 2-27, the print queue sends the `print(job)` message to a printer, which is followed by a return. Note that after handling the message, the printer object is no longer active, and the vertical rectangle ends.

You can place two forms of control information in a sequence diagram: condition and iteration. The condition marker indicates when a message is sent. The condition appears in square brackets above the message—for example, `[printer found]`. The message is only sent if the condition is true. The iteration marker indicates that a message is sent many times to multiple receiver objects. The basis of the iteration appears within square brackets immediately preceded by an asterisk (*). In Figure 2-27, the control information `*[for all queues]` indicates that the registry will send a `getResources()` message to each of the print queue objects it is managing. The condition marker `[printer found]` specifies that the print job will be sent to a printer only if one with the correct resources is located.

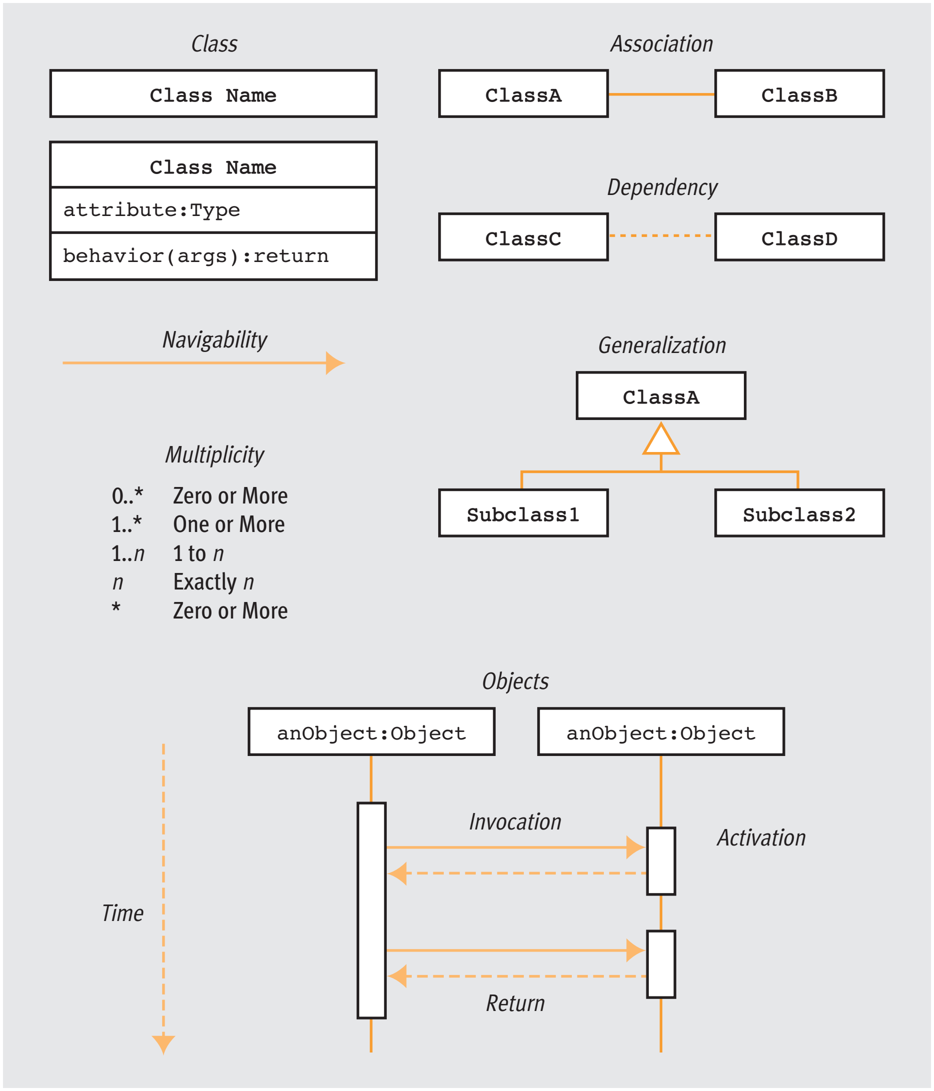
A sequence diagram illustrates the sequencing of events in an object-oriented system. It does not show the algorithms that are involved; it only shows the order in which messages are sent. Each of the classes that appears in a sequence diagram should be described in a separate class diagram. Note that if a sequence diagram indicates that an object sends a message to another object, then the corresponding class diagram must show a relationship, either an association or dependency, between those two classes. You should include a separate sequence diagram for each of the major behaviors in the system being modeled. The definition of a major behavior depends on your audience and what information you are trying to convey in your drawing, but as a guideline, you should include behaviors that are central to understanding the software being developed. For example, in our home heating system, it would be essential to understand the behaviors associated with obtaining the room temperature and activating the heaters. However, the behaviors associated with testing the thermostat or setting it for vacation would be less important. Be careful not to hide the central aspects of the design by including unnecessary information.

2.4 Summary

As we mentioned in Chapter 1, you must do a great deal of preparatory work before you start to write code. This preparatory work is called problem specification and program design.

This chapter looked more closely at the specification and design phases and introduced some fundamental principles of one development method called object-oriented design. This technique allows us to design and build software in a manner more closely related to how people organize and think about systems. Instead of a step-by-step procedural approach, the object-oriented model allows designers to think in terms of entities encapsulated with their external behaviors. Users do not need to know exactly how these behaviors are implemented, only that they are provided. This allows the internal implementation of a behavior to change without affecting a user's program, which greatly facilitates program maintenance. In addition, the object-oriented model provides a concept called inheritance, which allows classes to automatically acquire functionality from other classes, enhancing programmer productivity. Encapsulation and inheritance are two of the most important characteristics of object-oriented design.

The chapter also introduced a formalized notation for expressing and representing object-oriented designs: the Unified Modeling Language (UML). UML is an industrial standard that can express the design of a software system in the same way that blueprints can describe the design of a building. UML provides an expressive tool that a software architect can use to convey a design to a team of programmers. We will use UML throughout the text, including the case study in Chapter 4. The features of UML we introduced in this section are summarized in Figure 2-28.



[FIGURE 2-28] Summary of UML modeling elements

Our discussions in this chapter were rather abstract, using such lofty concepts as classes, behaviors, states, associations, and dependencies. However, once the design of our software is complete, we would like to reflect the design in our actual code by using an object-oriented programming language that linguistically supports the ideas presented in this chapter: classes, objects, methods, encapsulation, and inheritance.

Chapter 3 shows how these basic object-oriented concepts are realized and implemented in Java. Chapter 4 presents a significant case study that uses the concepts introduced in this chapter, along with the Java programming techniques from Chapter 3, to design and implement an object-oriented solution to our home heating problem.

ALAN KAY

Alan Kay is a pioneer of modern computing and was directly responsible for much of the information technology you use today. His amazingly accurate perception of the future led him to make seminal contributions in programming languages, graphical interfaces, networking, computer hardware, and the educational uses of technology. One of his most important contributions was the Smalltalk language. Many of the ideas embodied in modern object-oriented programming (a term that Kay himself coined) were based on Smalltalk, which was developed by Kay and his team at the Xerox Palo Alto Research Center (PARC).

Kay was once quoted as saying, "The best way to predict the future is to invent it," and he achieved that lofty goal. During the 1960s, 1970s, and 1980s, he and his colleagues produced an amazingly long list of new innovations, including bitmapped displays, window-based interfaces, pointing devices, laser printing, WYSIWYG editors, desktop publishing, electronic music, the Ethernet, and parts of the Internet. In fact, much of what we now consider "modern computing environments" are due to Kay and his colleagues at PARC, Apple, Disney, Hewlett-Packard, and the Viewpoints Research Institute. In Kay's own words, "The real romance is out ahead and yet to come. The computer revolution hasn't yet started."

In 1968, Kay helped to develop the Dynabook, which he called "a personal computer for children of all ages." The Dynabook was a portable computer with a flat screen, stylus, and mass storage, all connected to a wireless network. Kay had essentially envisioned what we now call a laptop computer, more than a dozen years before it became a commercial reality. Kay is still working to design and build low-cost, portable networked computers for children. In 2005, the MIT Media Laboratory unveiled a $100 laptop that Kay helped develop. The plan is to sell the laptops to schools around the world so students everywhere can access the vast storehouse of information available on the Internet and the Web.

For his visionary work in computing and education, Kay received the 2003 A. M. Turing Award from the ACM (Association for Computing Machinery), the highest award in the field of computer science.

EXERCISES

1 Review the design of some of the earliest object-oriented programming languages, such as Simula and Smalltalk. Discuss their influence on the C++ and Java languages.

2 Consider one process or task you do every day, such as making toast, buying groceries, or brushing your teeth. Identify at least two nontrivial objects involved in the process and detail their classes' data members and methods. Draw class diagrams for each of the classes you found.

3 Explain the distinction between the is-a and has-a relationships.

4 Determine whether the following groups would be related by inheritance. If they are, identify the superclasses and their subclasses: car/truck/vehicle, car/bicycle, rock/paper/scissors, and car/driver/tire.

5 List common methods of finding classes in a requirements document.

6 List and discuss differences between the design processes for an object-oriented program and a procedural program.

7 Define the term *object* in the context of object-oriented programming. Justify your definition by explaining how building "your" objects allows you to develop better code.

8 List the phases of the software life cycle and give a brief description of each. List common processes, the deliverables they produce, and the uses for those deliverables.

9 Write detailed and precise English instructions to perform a moderately complicated task. Write a procedural and an object-oriented solution to solve this task. Describe the differences in how you would use and reuse the code that would implement the procedural solution and how you would use the code that would implement the object-oriented solution.

10 List and explain differences between an `Integer` object and an integer variable.

11 Create a UML diagram for the most basic chair you can imagine. Call the class `ChairA`. Create two new types of chairs called `ChairB` and `ChairC`. Let `ChairB` and `ChairC` each extend `ChairA`, adding new functionality and features to the basic design of `ChairA`. Draw a UML diagram that describes the three chairs and illustrates their relationship. Derive `ChairD` from `ChairB`, adding at least one new feature and modifying the way in which one of `ChairB`'s methods works. Write pseudocode for the original `ChairB` method and its modified `ChairD` version. Add `ChairD` to the UML diagram for `ChairB`.

12 A requirements document for a medium-sized project could contain hundreds of nouns. Some of those nouns will become core classes of your software solution. The rest, while important, don't figure into the design quite so directly. Your solution's correctness, budget, and profit margin depend on finding those core nouns quickly and reliably. What characteristics will you look for to help distinguish the nouns that will become your classes? Write a short requirements document describing the desired operation of a program. Draw a class diagram for this program.

13 Two friends have decided to rent an apartment together. They work in different locations, with coordinates `X1`, `Y1` and `X2`, `Y2`. Given coordinates for local towns, design and diagram an object-oriented program that minimizes and equalizes the distance each friend must commute. Design the solution so that cities may be ranked by additional criteria. Your design should maximize (or minimize, as appropriate) these preferences as well.

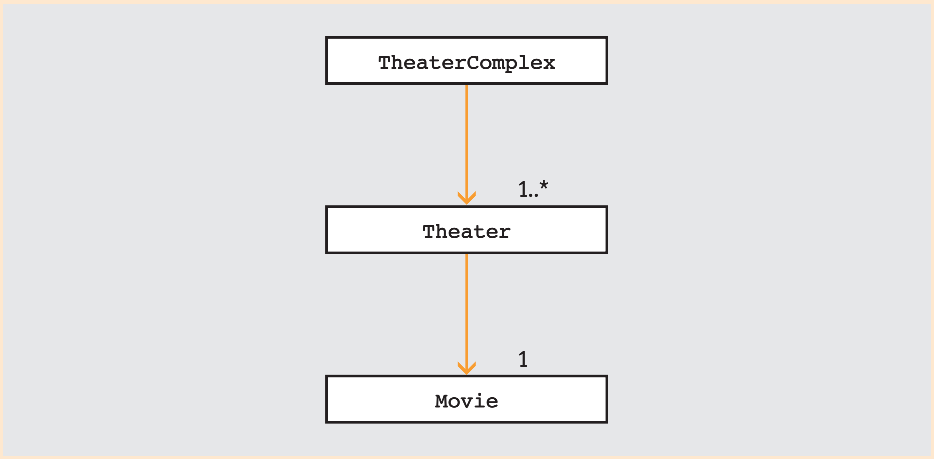
14 Big Oil Inc. has developed a new technique for locating undersea oil. By analyzing information from two different types of sensors—sonar and teledensitometers (TDs)—Big Oil scientists can pinpoint undersea oil fields. In an effort to lower risk to their employees, Big Oil plans to use robotic drones to explore interesting areas of the seabed. Each drone can physically mount one sensor module. Drones can be instructed to move along the ocean surface, report their position via the global positioning system, and hold their position while measurements are taken. All sensor modules allow data retrieval, but different sensor types have different controls, as follows:

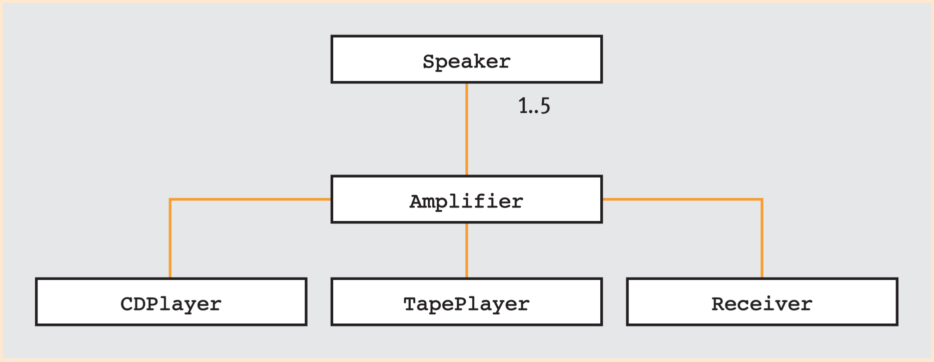
a A sonar sensor generates a pulse of sonic energy and listens for the echoes. Sonar readings are storage intensive, and so sonar modules cannot store more than one reading at a time. The current reading may be retrieved by the operator or overwritten by a new reading.

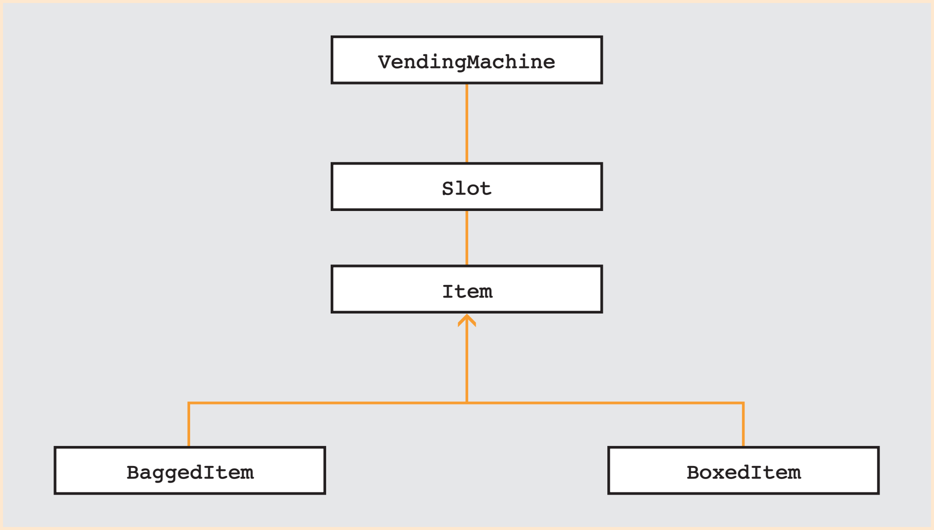
b A teledensitometer (TD) sensor generates readings using a continuous process. The TD sensor is switched on when it is over an area of interest. While switched on, data generated with confidence values past a certain threshold are added to the TD's temporary buffer. After collection, the operator can instruct the TD to verify that the buffer contains sufficient data for an accurate measurement. Depending on the result, the operator must either discard the buffer data, retain it to add more (by reactivating the TD sensor), or archive it to one of the TD module's five long-term storage locations (thus emptying the buffer). Appending data from one target point to a buffer that contains data from another target point corrupts the buffer's data.

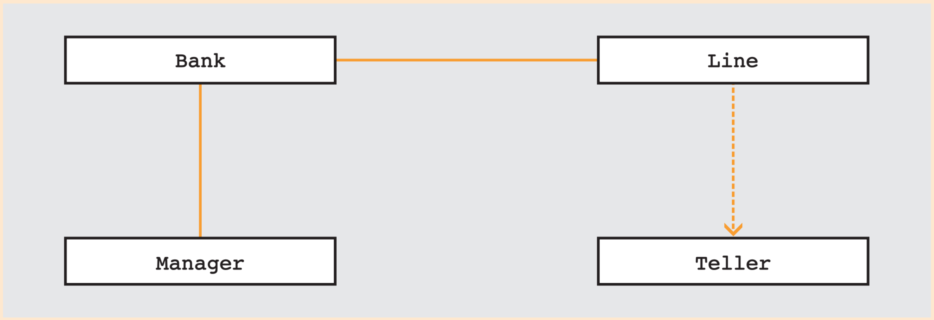
Write English instructions for using each of the two types of drones to survey an area for the presence of oil. Based on these instructions, design a class for a drone and each of the two sensor types. Document your design using a UML class diagram.

15 Describe in simple English the systems represented by the following four UML diagrams:






VendingMachine
Slot
Item
BaggedItem
BoxedItem


Bank
Line
Manager
Teller

CHALLENGE WORK EXERCISES

1 You now know that Java is not the only object-oriented programming language. Compile a list of all known object-oriented languages. Use the list to create a lineage that illustrates the order in which the languages were developed. Select two languages other than Java, and create a chart that compares their features.

2 Visual Basic is a popular language for developing programs that use a graphical user interface. Would you consider Visual Basic to be object oriented? To answer this question, list what you believe are the object-oriented features of the language. Look at the list you just developed and compare it to the topics discussed in this chapter. Is something missing?

3 The Java programming language is constantly evolving. What process would you have to follow to have a new feature added to the language? When will the next release of Java be available? What features will be added to the language in the next release?

4 Two types of software development processes are the software life cycle, which is often called the waterfall model, and extreme programming, which is described earlier in this chapter. See if you can find other development processes used in the industry. Compare the models you find.

5 The term *design pattern* describes a solution to a common problem in software development. The book *Design Patterns: Elements of Reusable Object-Oriented Software*, by Gamma et al., is a compilation of more than 20 design patterns. This book is often called the "gang of four" (often abbreviated *gof*) book in reference to the four authors. Use your favorite Internet search engine to search for "gang of four design patterns." Look at the search results and find a Web page that describes the design patterns in the book. Study five of the design patterns, then give a short description of each, and list two situations in which each would be useful.