[CHAPTER] 4

CASE STUDY IN Object-Oriented Software Development

4.1 Introduction

The first three chapters described the steps involved in software development. Chapter 1 overviewed the software life cycle, and Chapters 2 and 3 went into much greater detail on the cycle's design phase—namely, object-oriented design in Java. Although this is certainly important material, we cannot appreciate the complexities of modern software development by focusing on an individual phase any more than we can appreciate the intricacies of home building by focusing only on electrical contracting or mortgage financing. Instead, we must observe the *complete* development of a program, from initial specifications through design and implementation to final acceptance testing.

In this chapter we specify, design, and implement a piece of software mentioned often in earlier chapters: a home heating system. We will not write the software that would be loaded into a real thermostat to control a real furnace. Instead, we will develop a *heating simulation program* that models the operational behavior of an arbitrary room-thermostat-furnace combination. Using this simulator, we could perform a number of interesting experiments, such as the following:

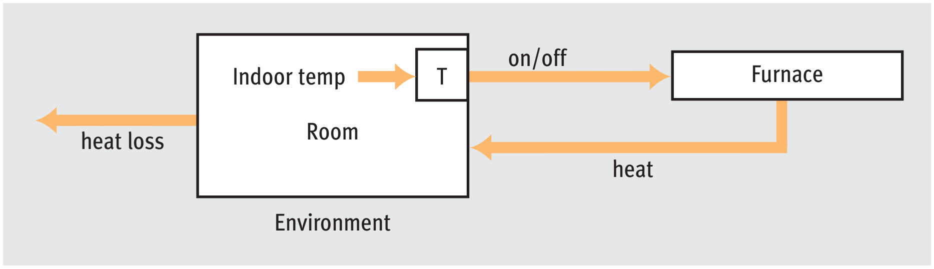
- Estimating how well a specific furnace works in a given climate
- Approximating annual heating costs for a particular living space
- Determining optimal settings for a specific thermostat and furnace

Our software development process parallels the life-cycle steps discussed in Chapter 1 and diagrammed in Figure 1-2. A careful reading of this case study can help to clarify and integrate the many ideas presented in the preceding pages. In the Challenge Work Exercises at the end of the chapter, we suggest projects that you can implement using the techniques shown in the following sections.

4.2 The Problem Requirements

Before we actually develop the specification document, let's clarify what the software is supposed to do. This requires decisions about program behavior as well as trade-offs between program efficiency, complexity, and cost.

Our home heating program simulates the operation of four components: a thermostat, a furnace, a living area (which we call the *room* even though it may encompass more than a single room), and the environment, which represents the space outside the room. These four components are diagrammed in Figure 4-1. (Because our goal in this case study is to demonstrate software development, not to construct a production program, we model a house with only a single thermostat and a single furnace, disregarding the issues of multi-zone heating systems.)



[FIGURE 4-1] The four components in our simulation model

The thermostat (labeled T in Figure 4-1) monitors room temperature and turns the furnace on when the temperature falls below a user-specified setting. It turns the furnace off when the temperature reaches the correct level. Most thermostats do not turn the furnace off as soon as the desired temperature is reached. Instead, they overheat the room by a few degrees to prevent the furnace from cycling on and off too frequently.

A furnace has two parameters of interest: its *capacity*, measured in BTUs per hour, and its *efficiency*. Capacity is the maximum amount of heat a furnace can produce, while efficiency is the percentage of that capacity it can actually deliver. The capacity of a typical home furnace is about 50,000 to 150,000 BTU/hour. The U.S. government requires all furnaces sold today to have efficiency ratings of at least 78%. Modern furnaces usually have much higher ratings, with efficiencies of 85 to 90%. The heat output of a furnace is determined by multiplying its capacity by its efficiency. The result is the number of BTUs generated in one hour.

Fluctuations in room temperature are caused by differences in the amount of heat pumped into a room by the furnace, which is zero if the furnace is off, and the amount of room heat lost to the surrounding environment. The formula for the new temperature at time period $(i+1)$ in a room with *FloorArea* square feet is:

$$RoomTemp_{i+1} = \frac{Q_{in} - Q_{loss}}{SHC \times FloorArea} + RoomTemp_i$$

Q_{in}, the *heat gain*, is the amount of heat coming in from the furnace, and Q_{loss}, the *heat loss*, is the amount of heat lost to the outdoor environment. *SHC*, the *specific heat capacity* of the room, measures the amount of heat stored in the room itself, including the walls, floors, and ceiling. The value of *SHC* for a typical house is 4.0 BTUs per square foot per degree Fahrenheit.

Q_{in}, the heat gain over a given period of time, is given by:

$$Q_{in} = \mathit{FurnaceOutput} \text{ (in BTUs/hour)} \times \mathit{time} \text{ (in hours)}$$

In a more sophisticated simulator, the amount of heat entering a room would include other factors such as solar radiation, electrical devices, and people, all of which generate heat. We neglect these other factors and assume that the only new heat produced comes from our furnace.

Q_{loss}, the heat loss to the surrounding environment over a given period of time, is given by:

$$Q_{loss} = \mathit{BLC} \times \mathit{FloorArea} \times (\mathit{InsideTemp} - \mathit{OutsideTemp}) \times \mathit{time} \text{ (in hours)}$$

BLC, the *basic load constant*, measures resistance to heat loss. A large value of BLC is indicative of a room that loses heat quite easily. A small value of BLC represents a room that is more resistant to heat loss, such as one that is very well insulated. In our simulation, BLC is 1.0 BTU per square foot per degree Fahrenheit per hour, a typical value for a well-built house.

We are now in a position to describe the problem we want to solve. We begin with a room at some initial temperature at time $T = 0$. We determine the room temperature at some future time $(T + \Delta)$ by computing how much heat has entered the room from the furnace and how much heat has been lost to the environment during the time increment Δ. We use this information to decide what action to take regarding the furnace. The possible operations are: (1) turning the furnace on if it is off and the temperature in the room has fallen below the thermostat setting, (2) turning the furnace off if it is on and the temperature in the room has risen above the thermostat setting plus some allowable amount of overheating, or (3) leaving the furnace in its current state and doing nothing. This process is repeated for the next time increment and continues until some maximum time limit is reached.

In the next section we turn this general problem statement into a complete, thorough, and unambiguous problem specification document. This document provides the information needed to design and implement a software package that solves the problem.

4.3 The Problem Specification Document

As we discussed in Chapter 1 and diagrammed in Figure 1-4, a specification is simply an input-output document that states exactly what outputs are produced for every possible input. Thus, our goal now is to describe exactly what inputs are required by the simulator and what results should be displayed.

Our home heating simulation program begins from the command line using the following command syntax:

```
java HeatingSimulation parameters...
```

The user can provide a number of optional parameters that control the characteristics of the simulation. The arguments take the form `paramName=paramValue`, where `paramName` is one of the eight parameters listed in Figure 4-2 and `paramValue` is the initial numerical value assigned to that parameter. If a user does not explicitly specify a value for any of these eight parameters, then the program uses the default value listed in Figure 4-2.

We selected this command-line format because it allows us to easily modify the program in the future. To add another model parameter—for example, to specify a different value for SHC, the specific heat capacity—we simply add a new parameter name and default value to the list in Figure 4-2. (After we complete our discussion of graphical user interfaces in Chapter 12, it would be interesting to return to this case study and create a visual, rather than command-line, interface to the software.)

PARAMETER NAME	DESCRIPTION	DEFAULT VALUE
`in`	The initial temperature of the room in degrees Fahrenheit	72.0
`out`	The outside environment temperature in degrees Fahrenheit	50.0
`set`	The desired room temperature (the thermostat setting) in degrees Fahrenheit	72.0
`cap`	The capacity of the furnace in BTUs/hour	50,000
`eff`	The efficiency of the furnace, given as a fraction of furnace capacity in the range of 0.0 to 1.0, inclusive	0.90
`size`	The size of the living area in square feet	250.0
`freq`	The number of ticks of the clock between successive lines of output	5
`length`	The total number of seconds the simulation will run	7200

[FIGURE 4-2] Command-line parameters to the simulation program

The program should ignore any invalid parameter names or out-of-range values. However, an illegal parameter value should terminate the program. For example, the following command line runs the simulation with a 1000-square foot living area and an initial room temperature of 65 degrees Fahrenheit. All other parameters are set to their default values:

```
java HeatingSimulation size=1000 in=65
```

Given the following command lines:

```
java HeatingSimulation lenth=200
java HeatingSimulation eff=1.05
```

the program will disregard the misspelled parameter `lenth` and instead use the default value of 120 for the parameter `length`. It also disregards the parameter value assigned to `eff`, as it is out of range. However, the following command:

```
java HeatingSimulation eff = zero point eight
```

terminates execution because the value provided for furnace efficiency has not been correctly expressed as a numeric value.

If all command-line values are legal and the program begins execution, then its output is a report (printed on the standard output device) that first displays the objects in the simulation and their initial state. Then, as the program runs, it displays every `freq` seconds of simulated time on a single line containing current room temperature, outdoor temperature, desired indoor temperature, and furnace status. A sample run of the program and the desired output is shown in Figure 4-3.

```
Created 4 Objects:
[ GasFurnace: cap=50000.0 eff=0.9 pilot on=true heating=false ]
[ Environment:  temperature=50.0 ]
[ Room:  temp=72.0 area=250.0 SHC=4.0 BLC=1.0 ]
[ Thermostat:  setting=72.0 overheat=3.0]

Starting simulation:   display frequency=5.0 runtime(sec.)= 7200.0

Time    Inside   Outside   Desired  Furnace (blank = Off)
----    ------   -------   -------  -------
0       72.00    50.00     72.00
300     73.79    50.00     72.00    On
600     75.52    50.00     72.00
900     74.99    50.00     72.00
1200    74.47    50.00     72.00
1500    73.97    50.00     72.00
1800    73.47    50.00     72.00
2100    72.99    50.00     72.00
2400    72.51    50.00     72.00
2700    72.05    50.00     72.00
3000    73.83    50.00     72.00    On
```

continued

```
3300    75.56    50.00    72.00
3600    75.03    50.00    72.00
3900    74.52    50.00    72.00
4200    74.01    50.00    72.00
4500    73.51    50.00    72.00
4800    73.03    50.00    72.00
5100    72.55    50.00    72.00
5400    72.09    50.00    72.00
5700    73.87    50.00    72.00    On
6000    75.60    50.00    72.00
6300    75.07    50.00    72.00
6600    74.55    50.00    72.00
6900    74.05    50.00    72.00
7200    73.55    50.00    72.00
```

[FIGURE 4-3] Sample program output

One of the most important parameters in the program is the time increment Δ, which is used to determine how often to recompute the room temperature.

This is not the same as the parameter `freq` *that determines how often to print a line of output. The parameter* `freq` *is expressed as a multiple of* Δ.

We are constructing a *discrete event simulator* that models the behavior of a system only at discrete points in time rather than continuously. In our simulation model we determine the room temperature at time T and then again at time $T + \Delta$. Any temperature change in the room between these two points in time is not part of the model. Thus, our clock will "tick" in units of Δ.

Obviously, we can achieve the highest level of accuracy by using a tiny value of Δ. However, the smaller the value of Δ, the greater the amount of computation we need to produce results, because we must run the program from time $T = 0$ to $T =$ `length` in steps of Δ. For example, to simulate one hour (3600 seconds) with $\Delta = 0.001$ second, the program must carry out 3.6 million recomputations of room temperature at times $T = 0.001$, 0.002, 0.003, ... , 3599.999, 3600.000. This may require a prohibitively large amount of machine time, causing the program to run quite slowly. On the other hand, if Δ is large, then the program does much less work, but it may not achieve a sufficient level of accuracy. For example, if $\Delta = 600$ seconds, or 10 minutes, we only need six iterations to simulate one hour (at $T = 600$, 1200, ... , 3600), but the program will not model the activities of either the thermostat or the furnace for an entire 10-minute period. The results will probably be highly inaccurate. So, we must make a trade-off between the resources consumed and the accuracy achieved.

We have two ways to handle this decision. One is to make Δ a user-specified parameter, exactly the same as the parameters listed in Figure 4-2. If we call the parameter `timeIncrement`, we might see a command line like the following:

```
java HeatingSimulation timeIncrement = 0.01
```

This is a reasonable approach. However, we have chosen not to do this because most users of our program will be unfamiliar with the technical concepts of discrete event simulation. Trying to explain how to select an appropriate time increment may confuse them and leave them unable to make a reasonable and informed choice. Instead, we will take another approach—we will set the value of this parameter (called SECS_BETWEEN_EVENTS in our model) to 60 seconds, or 1 minute. Thus, in our model, every clock tick represents the passing of one minute of simulated time. We hope that this choice of Δ produces accurate results without expending excessive amounts of time. For example, in the sample output of Figure 4-3, a line printed every 300 seconds of simulated time because the value of Δ was 60 seconds and the display frequency parameter `freq` was set to 5.

Figure 4-4 is a problem specification document that describes the behavior of the software we want to build. It is based on the many decisions, trade-offs, and assumptions made on the previous pages. It is now the user's job to carefully read and review these specifications to ensure that this document accurately describes the problem to be solved. If it does not, we can easily make changes to these specifications now; later in the development process, it will be much more difficult and expensive.

In the next section we begin designing the software that correctly and efficiently implements the specifications given in Figure 4-4.

Problem Specification Document

You are to build a home heating simulation program that models the behavior of four components: (1) a living area (or room) with `size` square feet and an initial temperature of `in` at time 0.0; (2) a single thermostat with a setting of `set` that measures room temperature; (3) a single furnace of capacity `cap` and efficiency `eff` that can be turned on and off by the thermostat and that is initially off; and (4) the environment outside the room that has a fixed temperature of `out`. The values of `size`, `in`, `set`, `cap`, `eff`, and `out` can be provided by the user on the command line. If a parameter is not specified, then use the default values given in Figure 4-2. If a parameter name is illegal, disregard it. If a parameter value is illegal, throw an exception and terminate execution.

continued

Advance the simulation clock by a value of $\Delta = 60$ seconds, and determine the new room temperature using the heat gain and heat loss formulas given in Section 4.2, with the two constants SHC and BLC set to 4.0 and 1.0, respectively. Based on the new temperature `t`, the thermostat should take one of the following three actions: if (`t < set`) and the furnace is off, then turn it on. If (`t ≥ (set + 3)`), where 3 represents the allowable amount of overheating, and the furnace is on, then turn it off. Otherwise, do nothing. Produce one line of output, as shown in Figure 4-3, if the current clock value is an even multiple of (Δ * `freq`), where `freq` is an input parameter that comes from the command line.

Continue this process until the simulation clock has reached or exceeded the value of `length`, which is also a command-line parameter. When that occurs, the program terminates.

[FIGURE 4-4] Problem specification document

4.4 Software Design

4.4.1 Identifying Classes

We are ready to begin designing a solution to the problem specified in Figure 4-4. The goals of the design phase are to: (1) identify the essential classes that are needed, (2) describe the state information and behaviors associated with these classes, and (3) diagram the relationships among these classes.

In Chapter 2, we showed that one of the best ways to identify the classes you need to solve a problem is to examine the nouns used most often in the problem specification document. Three nouns appear repeatedly in the document of Figure 4-4, and are excellent candidates to be made into classes:

- `Room`—A class representing a living area object
- `Thermostat`—A class representing objects that monitor temperature and take actions based on that temperature
- `Furnace`—A class representing objects that generate heat

Another noun that appears quite often is *environment*, which represents the world outside the room. Should that also be a class? It would certainly be possible to solve the problem without it. We could simply add an instance variable to our `Room` class to keep track of the temperature outside the room. The problem with this approach is that the outside temperature is not really part of the state of a room. This would be an example of a design that mixes state information from different sources into a single class, which can result in software that is both confusing and difficult to maintain. When designing classes you want to keep the states and behaviors of unrelated entities in separate classes. Therefore, if you find yourself doing such things as keeping employee salary information inside inventory objects or storing engine power data in driver objects, it is probably time to stop and rethink your entire design!

In this example the room object should obtain the outside temperature from another source, which argues strongly for `Environment` to be its own separate class. This approach also increases program flexibility. For example, in our simulation model the outside temperature is fixed and unchanging. However, in the future we might want the outside temperature to fluctuate during the day; for example, it might start out cool in the morning, increase during the afternoon hours, and cool off at night. To add this feature, we would only have to modify our `Environment` class. No other classes, including `Room`, would be affected. This is an example of how good object-oriented design can support and simplify maintenance.

Thus, we also include the following class in the design of our model:

- `Environment`—A class representing the world outside the room.

One other noun appears rather frequently in Figure 4-4, but is not as obvious a candidate for inclusion as the four classes identified so far. It is not as obvious because it is part of the simulator rather than the world being modeled. This is the *simulation clock*. One of the program's most important responsibilities is keeping track of time. This is necessary, for example, to determine when to compute a new room temperature, produce output, or terminate the program. A `Clock` class would be responsible for keeping track of the simulation time and notifying other objects when certain events are to occur. Thus, the next class we add to our design is:

- `Clock`—A class representing objects that can keep track of simulation time and inform other objects when certain events have occurred.

These, then, are the five classes we will include in the design of our home heating simulator. You should carefully review the specification document in Figure 4-4 to see how its contents—most importantly, its nouns—led us to select these five classes.

4.4.2 State and Behavior

There are many other nouns in the specification document of Figure 4-4, such as *capacity*, *efficiency*, *area*, and *temperature*. However, a little thought should convince you that these nouns do not represent separate classes, but *attributes* of the classes described in the previous section. For example, capacity and efficiency are not distinct entities or objects but instead are part of the state of furnace objects. Area is certainly part of the state of a room object, and temperature is a state variable of room objects (indoor temperature), environment objects (outdoor temperature), and thermostats (desired temperature). Thus, the problem specification document also helps a designer identify state information.

Figure 4-5 lists some of the important state information associated with the five classes we have chosen to include in our solution. We may discover others as we get further into the implementation.

CLASS	STATE INFORMATION (ATTRIBUTES)
`Environment`	Outdoor temperature
`Furnace`	Capacity
	Efficiency
	On/off state
`Room`	Current temperature
	Area in square feet
	Reference to the environment outside the room
	Reference to the furnace that heats the room
`Thermostat`	Desired temperature setting
	Amount of allowable overheating
	Reference to the furnace that it controls
	Reference to the room in which it is located
`Clock`	Current time
	Seconds between clock ticks

[FIGURE 4-5] Classes and state information

Now that we have identified (at least initially) the five classes in our simulator and the state information maintained by instances of these classes, we can begin to describe the behaviors, or actions, of these classes.

Three types of behavior are part of virtually every class, and can be viewed as boilerplate elements in every class you write. The first of these standard behaviors is the *constructors*. We must be able to create instances of a class, so we always need to have one or more constructors. Exactly how many are needed depends on the characteristics of the class. For example, `Room` may require two constructors—one to create a room where the square footage and the initial temperature are provided by the user, and a second constructor that creates a new instance of a room using default values for both of these parameters.

The second of these standard behaviors are *accessor* and *mutator* functions that allow access to (and possibly modification of) the instance variables of a class. Because instance variables are typically not accessible from outside the class, we must include methods of the form `getXXX()` that access and return the value of the private instance variable `XXX`, where `XXX` represents the variable name. If modifications to this variable are permitted, you should also include mutator methods of the form `setXXX(newVal)` that reset the value of the instance variable `XXX` to `newVal`.

For example, one of the state variables of `Environment` is `temperature`. A room object needs this temperature to compute Q_{loss}, the amount of heat lost to the environment. `Room` cannot access this value directly. Instead, it calls the accessor function `getTemperature()` in class `Environment`. Because our model does not allow changes to the outdoor temperature, a public mutator method is not needed. However, if this were possible, then `Environment` would include a method `setTemperature(double t)`, which resets the outdoor temperature to `t`.

The third standard behavior is to override the `toString()` method inherited by every object in Java from class `Object`. (This operation was highlighted in Figure 3-57 and Table 3-3.) The `toString()` method in class `Object` displays information about an object. However, this information is very general because it must apply to every object in the system. It is often helpful to override this method with a new method that prints more specific information about the particular object. This can be extremely helpful during debugging.

In addition to these three standard behaviors, objects also carry out actions appropriate only to them. How do we begin to identify these unique behaviors? The answer was again provided in Chapter 2, which said that we should look at the *verbs* in the problem specification document. These verbs can help us identify the actions that objects perform, and these actions frequently translate directly into methods in our classes.

For example, the problem specification states that we "determine the new room temperature." Obviously, one behavior of a room object is the ability to compute and store a new room temperature using the heat loss and heat gain formulas from Section 4.2. Using this method, the thermostat can query the room for its current temperature at selected intervals:

```
// determine the new room temperature t time
// units after the last determination
public void determineTemperature(double t);
```

Similarly, the specification document contains statements that the thermostat must turn the furnace on and off. These actions translate into a `Furnace` class method that looks like this:

```
// set the furnace heating state to the value of onOff
public void setHeating(boolean onOff);
```

Using this method, the thermostat can send a message to the furnace to turn itself on or off, depending on the value of its Boolean parameter.

Finally, a statement in the specification document reads "the thermostat should take one of the following three actions: if (`t < set`) and the furnace is off, then turn it on. If (`t ≥ (set + 3)`)... and the furnace is on, then turn it off. Otherwise, do nothing." These actions translate into a thermostat method that determines whether we need to change the state of the furnace:

```
// Return true if we need to change the furnace
// state based on the thermostat setting, current
// room temperature, and amount of allowable
// overheating, currently 3F. Otherwise, return false
public boolean determineStateChange();
```

If we do need to change the state of furnace `f`, then we invoke the `f.setHeating(b)` method, where `b` is the Boolean value returned by `determineStateChange()`.

In a similar fashion, we go through the entire specification document, identifying verbs that correspond directly to desired behaviors in our objects. Figure 4-6 lists the five classes contained in our solution and some behaviors we have identified for objects of those classes. Of course, we may add more behaviors as the testing process progresses.

```
• Environment
      // Create a new environment with the default temperature
      Environment();

      // Create a new environment with temperature t
      Environment(double t);

      // Return the current temperature
      double getTemperature();

      // Set the temperature to t
      void setTemperature(double t);


      // Return a string representation of the state of the environment
      String toString();

• Furnace
      // Create a new furnace with capacity c, efficiency eff,
      // that is initially turned off
      Furnace(double cap, double eff);
```

continued

```
        // Return the capacity of the furnace
        double getCapacity();

        // Return the efficiency of the furnace
        double getEfficiency();

        // Return the state of the furnace
        boolean isHeating();

        // Set the furnace state to onOff
        void setHeating(boolean onOff);

        // Determine the output of this furnace for the time period hrs
        double output(double hrs);

        // Return a string representation of the state of the furnace
        String toString();

  · Room
        // Create a room of size area and initial temperature initTemp
        // The room is heated by furnace f and is inside environment e
        Room(Environment e, Furnace f, double area, double initTemp);

        // Return the floor area of the room
        double getFloorArea();

        // Return a reference to the furnace heating this room
        Furnace getFurnace();

        // Return a reference to the environment outside the room
        Environment getEnvironment();

        // Determine the new temperature in the room after t
        // units of time
        void determineTemperatureChange(double t);

        // Return a string representation of the state of this room
        String toString();

  · Thermostat
        // Create a new thermostat in room r connected to furnace f
        Thermostat(Room r, Furnace f);

        // Return a reference to the room where the thermostat is located
        Room getRoom();

        // Return a reference to the furnace the thermostat controls
        Furnace getFurnace();
```

continued

```
        // Return the current temperature setting of the thermostat
        double getSetting();

        // Change the setting on the thermostat to newSetting
        void setSetting(double newSetting);

        // Get the value of the constant overheat, which is how much
        // a room is overheated before the furnace is turned off
        double getOverHeat();

        // Determine if we need to change the state of the
        // furnace based on the thermostat setting, the current room
        // temperature, and the amount of allowable overheating
        void determineStateChange();

        // Return a string representation of the state of this thermostat
        String toString();

· Clock
        // Create a clock that advances tickInterval seconds each tick
        Clock(int tickInterval);

        // Return a string representation of the state of the clock
        String toString();
```

[FIGURE 4-6] Classes and their behaviors

4.4.3 Inheritance and Interfaces

Now that we have selected our classes and identified their states and behaviors, we should be ready to begin implementation. In fact, we could build a good home heating simulator, given the design work done so far.

However, we want to add a couple of features to our design before jumping into implementation. These features will help make the finished program more flexible and maintainable and will demonstrate the advantages that accrue from the intelligent use of inheritance. We will use both the specification and specialization forms of inheritance described in Section 2.2.3 and highlighted in Table 2-1.

Our first design change comes from recognizing that there are many furnaces in the marketplace with a wide range of operating characteristics. Although they all share certain properties, such as a given heating capacity, different models have specialized features. This is an example of the *specialization* form of inheritance, in which the subclass is a more specialized form of its superclass.

For example, one popular type of furnace is the gas furnace. We can confirm that this is a specialization form of inheritance by noting that furnaces and gas furnaces satisfy the is-a relationship introduced in Section 2.2.3; that is, "A gas furnace is a furnace." A gas furnace and a furnace share the same state information listed in Figure 4-5, namely heat capacity, efficiency, and an on/off state. They also share the behaviors in Figure 4-6—they can be turned on and off and, when on, they both generate heat. However, a gas furnace has one feature not found in general `Furnace` class objects: a *pilot light*. A gas furnace cannot generate heat unless its pilot light is on. (New homeowners often discover this fact on the first cold day of winter!) Thus, to implement the concept of a gas furnace, we can keep all the capabilities of our existing `Furnace` class. Then we only need to add new information about the state of the pilot light and new behaviors to check and set its state.

Instead of creating a subclass called `GasFurnace`, maybe it would be simpler to add the following state variable:

```
boolean pilotLight;
```

to our existing `Furnace` class, along with new instance methods that access and modify this variable. This would be a serious mistake. In effect, we would be saying that *every* furnace has a pilot light, which is incorrect, of course. Later, we might want to model the behavior of a furnace type that does not have a pilot light, such as a solar panel. Because of our poor design, making this small change could require modifying a good deal of code, increasing both maintenance costs and the likelihood of errors.

One of the important goals of design is to *localize* changes that must be made to the software. That is, if a change is needed to some portion of code X, then only the code that deals with X should have to be reviewed and modified. The sections of code that have nothing to do with X should be unaffected by this change. In this example, if we implement a new furnace type, we should not need to worry about a pilot light state variable because the new furnace may not even have one.

We can achieve this localized behavior using inheritance. We let the superclass `Furnace` represent the shared state and behavior common to *all* furnaces. Then, whenever we want to model a new furnace type, we create a subclass that inherits these common states and behaviors and only adds or modifies characteristics that are unique to the particular furnace type. That is, we only need to write new code to (1) implement specialized behaviors and (2) override existing behaviors that work differently on this new system. All other behaviors are inherited from the superclass and used as is. This approach should simplify maintenance and minimize the chance for unexpected errors to creep into the code.

Thus, we will add the following new `GasFurnace` subclass to our design.

```
/**
   * This new GasFurnace subclass extends Furnace and has
   * the following new states or changed behavior
   */
Subclass     Extends    New state    New or changed behaviors

GasFurnace   Furnace    pilotLight   // true if pilot light is on,
                                     // false otherwise
                                     boolean isPilotOn();

                                     // set pilot state to onOff
                                     void setPilot(onOff);
                                     // modify to produce heat only
                                     // if the pilot light is on
                                     double output(double hours);
```

We will follow the same steps described previously to add other specialized furnace types to our simulation model.

Our second design change involves how the simulation clock communicates with other objects in our software. We are building a discrete event simulator, a model in which events happen only at discrete points in time. The clock is responsible for keeping track of simulation time and informing other objects when certain things must be done. It is like having a single timekeeper in a sporting contest (the clock) that informs the referees (the other objects) by a horn or siren when certain events have occurred, such as the end of the game.

In our software, two classes need to be informed of the current time—`Room` and `Thermostat`. The room object must recompute room temperature at explicit points in time (every one minute in our model), and the thermostat must access that new temperature and decide what, if anything, should be done regarding the state of the furnace. What is the best way for the clock to inform the room and thermostat objects of the current time?

One technique would be for the room and thermostat objects to repeatedly send messages to the clock asking for the time. This is horribly inefficient and could lead to the type of behavior we see in small children on a long car trip—"Are we there yet? Are we there yet?" In a similar vein, the room and thermostat objects would be sending messages saying, in effect: "Is it time to recompute temperature? Is it time to recompute temperature?"

A better way to handle this issue is to reverse the direction of the communications; that is, the clock sends a message to the room and thermostat objects when it is time for them to perform some operation, such as recomputing the new room temperature. In our design we permit an object to put itself on a list of objects that want to receive timing information from the clock. The question we must answer, though, is how we can be sure that these objects can receive the messages that will be sent to them? Sending a message `f()` to an object implies that the object contains a public instance method called `f()`. How can we be sure that `Room` and `Thermostat` have such methods?

The answer is to create an interface that specifies all of the messages that `Clock` can possibly send out. Then, any class that wants to receive timing information can do so by implementing this interface, thus guaranteeing that they contain all the methods necessary to receive the clock information. This is an example of the *specification form* of inheritance listed in Table 2-1, in which the superclass (the interface) defines behaviors implemented in the subclass (the classes implementing the interface) but not in the superclass.

Our design includes an interface called `ClockListener` that must be implemented by any class that wants to receive timing information. The `ClockListener` interface specifies the following two behaviors:

- `preEvent(double timeInterval)`—This method is sent out by the `Clock` just *before* an event is about to take place. This provides the object receiving the message with a chance to update its state since the last event occurred, which was `timeInterval` ticks ago.
- `event()`—This method indicates that the event has just taken place.

For example, let's assume that `Room` implements the `ClockListener` interface described above. A room object needs to recompute its temperature based on what has happened since the last event. To do this, it must check the on/off status of the furnace, the outdoor temperature, and the time interval `t` since the last computation. It then completes the heat loss and heat gain formulas of Section 4.2 using these values. All these operations are performed when the room object receives a `preEvent(t)` message. The room temperature state variable is not changed until the `event()` method is invoked because we want to be sure that the recomputations were done with the previous values, not the newly recomputed ones.

Events are handled this way to avoid any problems that might occur because of the order in which `event()` messages are sent to specific objects. For example, in our model we want to ensure that the thermostat checks the temperature in the room *after* the temperature has been updated, not before. If we sent an `event()` message to both the room and thermostat telling them it was time to carry out an event, we could not be sure of the order in which the operations would be performed. The thermostat might access either the old room temperature or, if `Room` finishes its computations first, the newly recomputed value. By sending the `preEvent()` message first, we ensure that all preparations for the upcoming event are made by every object receiving this message. Then, these new values are assigned to the appropriate state variables when the `event()` operation is invoked.

Another nice feature is that if we add new classes that also need timing information, it is easy for these new classes to obtain it. They just have to implement the `ClockListener` interface.

THE OPEN SOURCE MOVEMENT

In this case study we have implied that software is designed and implemented by a team of professionals working within a single organization to create a product they own and sell for a profit. In fact, most software from companies like Microsoft, IBM, and Oracle is developed in just such a proprietary manner.

However, massive software projects such as Java compilers, Windows XP, and Microsoft Office—which are much larger than our home heating simulator—can take thousands of person-years to complete. The likelihood of getting everything in such a large program to work perfectly can be rather small; we have all been frustrated by freezes, errors, and crashes in our application software and operating systems.

Many people around the world are addressing the development of correct, efficient, and elegant software through the **open source movement**. They believe that the best way to develop bug-free software is to enlist the cooperation of skilled, altruistic programmers who agree to work for free. These programmers are inspired by the goals of producing high-quality software and working cooperatively with like-minded people. The source code is freely available, and changes and improvements can be made by anyone with a good idea. Programs that result from this group effort are widely distributed for both personal and commercial use, which is quite different from the proprietary approach of IBM or Microsoft. There, the software design is kept secret and the source code is not shared outside the development team or the corporation.

The open source movement encourages contributions to software development from anyone in the world, in the belief that a more open process increases the likelihood that errors will be located, oversights will be corrected, and improvements will not be overlooked. The popular Linux operating system and the Apache Web server were both developed using the open source model, as was the successful free encyclopedia Wikipedia. The latter project began in 2001 and is a collaborative effort of tens of thousands of volunteers who freely contribute articles. Today, Wikipedia includes about 3 million articles in 10 languages, including more than 1.4 million articles in English alone. (By comparison, the *Encyclopedia Britannica* contains about 65,000 articles.)

4.4.4 UML Diagrams

The last thing we do before implementation is diagram the relationships among the various classes, subclasses, and interfaces in our solution. We have already identified some of these relationships during our preliminary specification and design work. For example, the following sentences describe important interactions between classes; they come from the discussions in the preceding sections:

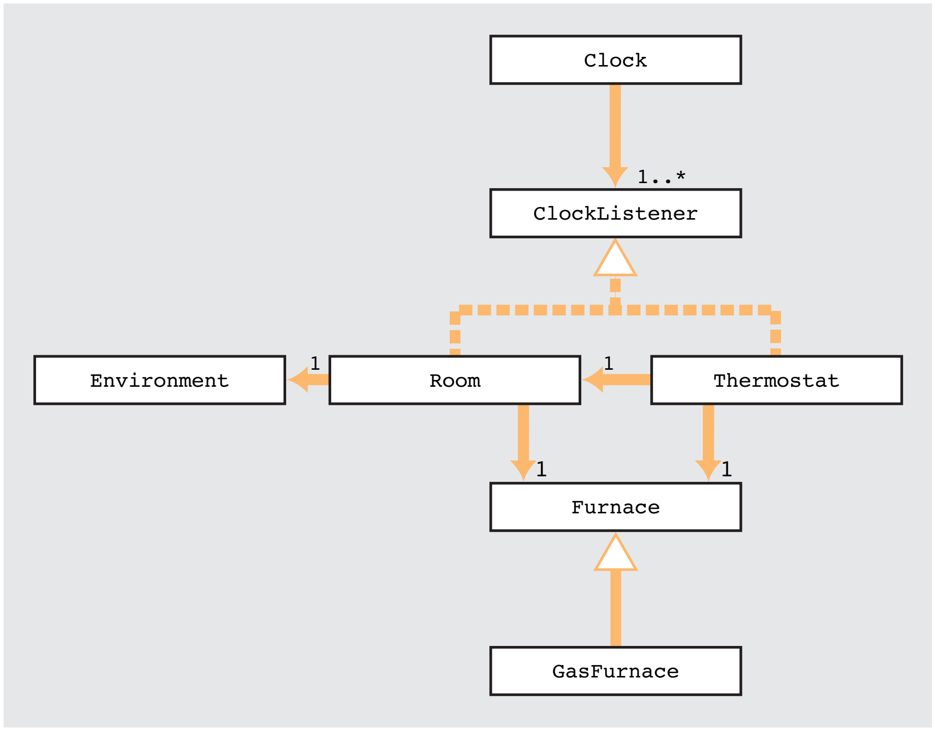
- "The room obtains the outside temperature from the environment..."
- "The thermostat queries the room to learn the indoor temperature..."
- "The thermostat turns the furnace on or off depending on its setting and the current indoor temperature..."

Because of the small size of this project—only five classes, one subclass, and one interface—we probably do not need to formally diagram the relationships between entities. We could probably keep them all in our head.

The finished home heating simulation program is about 1100 lines long, including comments. According to Figure 1-1, this puts the program in the Small category.

However, as we emphasized in Chapter 1, real-world packages are quite large, incorporating hundreds of thousands of lines of code and dozens or hundreds of packages, classes, subclasses, and interfaces. In this software development environment, UML diagrams are invaluable in helping the programmer to write and maintain code. For this reason, we present some important UML diagrams for this software project.

Figure 4-7 is a UML class diagram that includes the seven major components in our design. This figure contains five classes—`Clock`, `Room`, `Thermostat`, `Furnace`, and `Environment`—one subclass, `GasFurnace`, and one interface, `ClockListener`. `Room` must know about the single `Environment` in which it is contained and the single `Furnace` that heats it. `Thermostat` must know about the `Room` it is in and which `Furnace` it controls. `Clock` does not need to know directly about any other class. Instead, it simply sends messages to every class that implements the `ClockListener` interface.



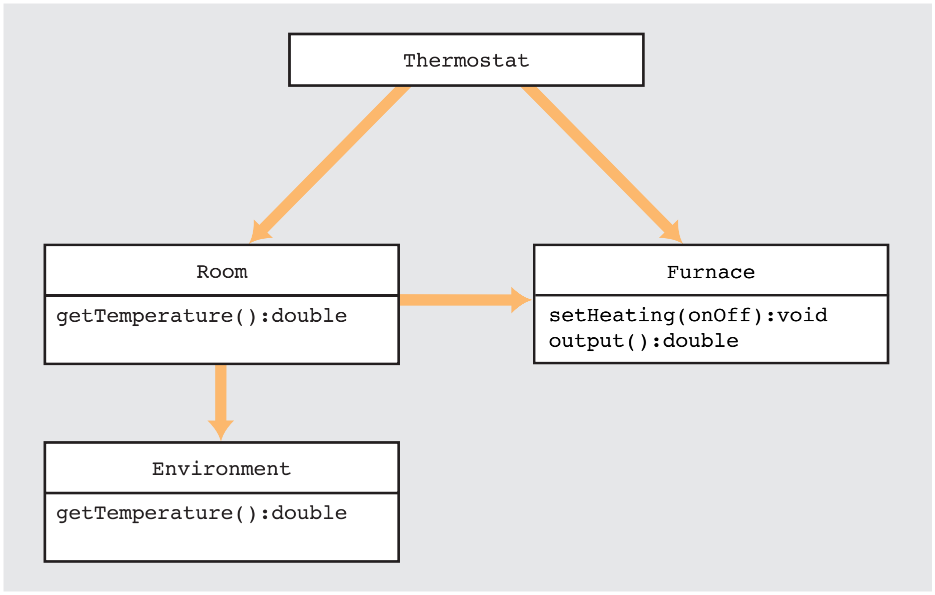
[FIGURE 4-7] Major classes in the home heating simulation

The dotted line in Figure 4-7 indicates that, in our design, the two classes `Room` and `Thermostat` implement this interface and can communicate with `Clock` by receiving its timing messages. Finally, Figure 4-7 specifies that `GasFurnace` is a subclass of `Furnace`.

This UML diagram captures all of the major relationships that exist among the classes, subclasses, and interfaces in our program. However, it does not specify the behaviors or states that exist within these classes. To capture that information, we can create additional UML diagrams that focus on smaller segments of the diagram in Figure 4-7 and that contain additional information about the state and behavior of individual classes.

For example, Figure 4-8 is a UML diagram that focuses on the heating component section of our solution—the `Furnace`, `Thermostat`, `Room`, and `Environment` classes. This diagram identifies not only the relationships that exist between these components but the methods used to communicate between classes.

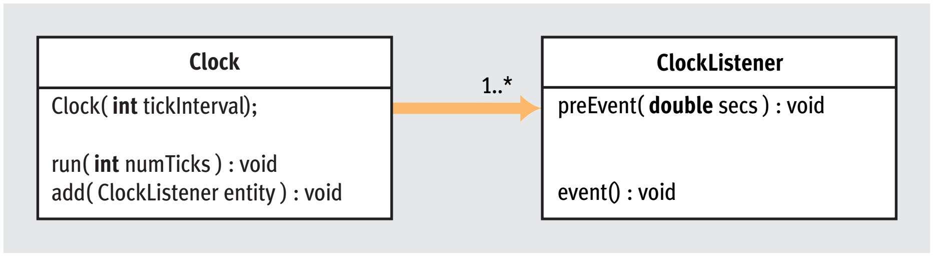
The `Thermostat` class uses the `getTemperature()` method of `Room` to access the current room temperature. If the state of the `Furnace` must be changed, then `Thermostat` uses the `setHeating(onOff)` method to turn the `Furnace` on or off. `Room` uses the `output()` method of `Furnace` to determine the amount of furnace heat entering the room, and the `getTemperature()` method of `Environment` to determine the outdoor temperature. From these two values `Room` is able to compute the new room temperature.



[FIGURE 4-8] The heating portion of the home simulator

Figure 4-9 is a UML diagram that highlights the timing section of our solution, namely the `Clock` class and the `ClockListener` interface. The diagram specifies that the two key timing operations of the clock are the following:

- Create a new `Clock` that ticks every `tickInterval` seconds. In our design, `tickInterval` is set to 60 seconds. (We referred to `tickInterval` as Δ in earlier discussions.) In a different design, a user might enter this value.
- Run the clock in steps of `tickInterval` from time 0 to time `numTicks`.



Clock
Clock(int tickInterval);
run(int numTicks) : void
add(ClockListener entity) : void
1..*
ClockListener
preEvent(double secs) : void
event() : void

[FIGURE 4-9] The `Clock` class and the `ClockListener` interface

In addition, our Clock keeps a list of all entities that want to receive timing information. There can be as many as we want, the only restriction being that every entity that wants to receive timing information must implement the ClockListener interface. New entities are added to the list using the add(entity) method of Clock.

Finally, Figure 4-9 shows that this timing information is sent to all entities on the list using the two methods preEvent(secs) and event()discussed earlier. Because these entities have implemented the ClockListener interface, we are guaranteed that they can correctly receive these messages.

These are the only UML diagrams we present in this case study. However, if this were a more complex software project, we would likely prepare additional diagrams that would help to clarify and explain our design. Among the most important are the *sequence diagrams* introduced in Section 2.3.2.3. Sequence diagrams describe the dynamic interactions and exchanges between classes. For example, Figure 4-8 specified that Thermostat must query both Room and Furnace to determine what actions to take. After determining what to do, it interacts with Furnace to change its state. Although these interactions are clearly identified in Figure 4-8, that diagram does not capture the order in which these actions take place. A sequence diagram provides just this type of information, and it can be a great help to the programmers implementing these classes.

4.5 Implementation Details

Finally, after many pages of requirements, specification, and design, we have arrived at implementation—the phase that many students mistakenly identify as the most creative and important part of software development. We hope that the discussions preceding this section have helped to correct this view and have suggested how much work you must do before even one line of code is written. The previous sections should also help you better understand Figure 1-10, which states that only 15 to 20 percent of a software development project is spent on coding and debugging, while 30 to 40 percent is spent on requirements, specifications, and design. Finally, we hope these discussions have demonstrated that the truly creative and innovative part of software development is design, not implementation. Looking back at the many decisions already made—classes, subclasses, interfaces, states, behaviors, interactions—it is obvious that we have already completed much of the truly interesting and challenging work. Our task now is to take this design and translate it into correct and efficient Java code. The job of the implementation phase is much like the responsibilities of a building contractor, who must take an architect's creative ideas and turn them into floors, walls, and roofs.

The first class we implement is Environment.java, as shown in Figure 4-10. There is not much to say about this class, as it is basically a container for the outside temperature. Therefore, you might consider it unnecessary. However, as we said earlier, the advantage of making `Environment` a separate class is that, at some future time, it will be much easier to modify the program to allow the outdoor temperature to fluctuate in interesting and complex ways. Note also that we have overridden the `toString()` method inherited from class `Object` to provide more descriptive information about the `Environment`.

```java
/**
 * This class represents the external environment that a room
 * is contained in.  The primary role of this class is to maintain
 * the outside temperature.  We will be using a fixed outdoor
 * temperature, but it could be modified to vary according to
 * some environmental rules.
 */
public class Environment {
    // Initial temperature.  Used by the default constructor.
    public static final double DEFAULT_TEMPERATURE = 72.;

    // The current temperature in this environment
    private double temperature;

    /**
     * Create a new environment using the default temperature.
     */
    public Environment() {
        this( DEFAULT_TEMPERATURE );
    }

    /**
     * Create a new environment with the specified temperature.
     *
     * @param initialTemp initial temperature of the environment.
     */
    public Environment( double initialTemp ) {
        temperature = initialTemp;
    }

    /**
     * Return the environment's current temperature.
     *
     * @return the current temperature.
     */
    public double getTemperature() {
        return temperature;
    }

    /**
     * Set the temperature of the environment.
```

continued

```
     *
     * @param newTemp temperature to set the environment to.
     */
    public void setTemperature( double newTemp ) {
        temperature = newTemp;
    }

    /**
     * Return a string representation of the environment.
     *
     * @return a string representation of the environment.
     */
    public String toString() {
        return "[ Environment: " +
               " temperature=" + temperature +
               "]";
    }
}
```

[FIGURE 4-10] Code for `Environment.java`

The next class we implement is `Furnace.java`. Its code is shown in Figure 4-11.

This class contains three private instance variables—`capacity`, `efficiency`, and `heating`, a Boolean value that specifies the on/off state of the furnace. There are accessor functions for each of these state variables as well as a mutator function, `setHeating(boolean onOff)`, which allows us to change the furnace's state. The other interesting method is `output(double hours)`, which determines the total amount of heat generated by a furnace over a given time period using the formula from Section 4.2:

$$\textit{heat output} = (\textit{capacity} \times \textit{efficiency} \times \textit{time})$$

Our `Furnace` class represents an idealized furnace object, in that it includes only the characteristics common to all furnace types, but no specialized characteristics for a particular type.

```
/**
 * A furnace in the home heating simulation. When on, a furnace
 * will produce a certain number of BTUs/hour of heat. The furnace
 * has a capacity that gives the maximum number of BTUs it can
 * generate and an efficiency that determines what percentage of the
 * furnace's capacity is actually generated as output.
 */
```

continued

```
public class Furnace {
    private double capacity;    // Capacity of the furnace in BTUs/hour
    private double efficiency;  // The efficiency of the furnace
    private boolean heating;    // Is the furnace producing heat?

    /**
     * Create a new furnace.
     *
     * @param cap the capacity of the furnace in BTUs/hr
     * @param eff the efficiency of the furnace
     */
    public Furnace( double cap, double eff ) {
        // Store the state of the furnace
        capacity = cap;
        efficiency = eff;

        // Make sure the furnace is off
        heating = false;
    }

    /**
     * Turn the furnace on (i.e., it will produce heat) or off.
     *
     * @param onOff if true the furnace will produce heat.
     */
    public void setHeating( boolean onOff ) {
        heating = onOff;
    }

    /**
     * Return the capacity of this furnace.
     *
     * @return the capacity of the furnace.
     */
    public double getCapacity() {
        return capacity;
    }

    /**
     * Return the efficiency of this furnace.
     *
     * @return the efficiency of this furnace.
     */
    public double getEfficiency() {
        return efficiency;
    }
```

continued

```
    /**
     * Return the state of the furnace.
     *
     * @return true if the furnace is producing heat.
     */
    public boolean isHeating() {
        return heating;
    }

    /**
     * Return the number of BTUs produced by the furnace during the
     * specified period of time.
     *
     * @param hours number of hours the furnace has been on.
     * @return the number of BTUs produced by the furnace during
     *         the specified time period.
     */
    public double output( double hours ) {
        double btusGenerated = 0.0;

        if ( heating ) {
            btusGenerated = capacity * efficiency * hours;
        }

        return btusGenerated;
    }

    /**
     * Return a string representation of this furnace.
     *
     * @return a string representation of this furnace.
     */
    public String toString() {
        return "[ Furnace:" + " cap=" + capacity +
               " eff=" + efficiency + " heating=" +
               heating + " ]";
    }
}
```

[FIGURE 4-11] Code for `Furnace.java`

We can handle the enormous range of furnace types by implementing them as subclasses of the `Furnace` class of Figure 4-11. The only type we include in this case study is `GasFurnace.java`, whose code is shown in Figure 4-12. However, it would be easy to add more subclasses to implement the unique heating properties of devices such as solar panels and heat pumps. We would implement them as subclasses of `Furnace`, exactly as shown in Figure 4-12.

The only state variable added to this subclass is `pilotLight`. In addition, we have included accessor and mutator functions called `isPilotOn()` and `setPilot(onOff)`, which allow us to check and set the value of this new state variable.

Note how this new subclass overrides the `output()` method inherited from `Furnace`. This was necessary because a gas furnace only generates heat if its pilot light is on. Also note that instead of rewriting the entire method, it uses `super` to invoke the `output()` method in the superclass once we are sure the pilot light is on.

```java
/**
 * A gas furnace.  A gas furnace has a pilot light that must be on
 * for the furnace to produce heat.
 */
public class GasFurnace extends Furnace {
    private boolean pilotLight; //State of the pilot light

    /**
     * Create a new furnace.
     *
     * @param cap the capacity of the furnace in BTUs/hr
     * @param eff the efficiency of the furnace
     */
    public GasFurnace( double cap, double eff ) {
        super( cap, eff );

        // Pilot light is off
        pilotLight = false;
    }

    /**
     * Turn the pilot light on.
     *
     * @param onOff if true the pilot light is on.
     */
    public void setPilot( boolean onOff ) {
        pilotLight = onOff;
    }

    /**
     * Return the state of the pilot light.
     *
     * @return true if the pilot light is on.
     */
    public boolean isPilotOn() {
        return pilotLight;
    }
```

continued

```
    /**
     * Return the number of BTUs produced by the furnace during the
     * specified period of time.
     *
     * @param hours number of hours the furnace has been on.
     * @return the number of BTUs produced by the furnace during
     *         the specified time period.
     */
    public double output( double hours ) {
        double btusGenerated = 0.0;

        if ( pilotLight ) {
            btusGenerated = super.output( hours );
        }

        return btusGenerated;
    }

    /**
     * Return a string representation of this furnace.
     *
     * @return a string representation of this furnace.
     */
    public String toString() {
        return "[ GasFurnace:" + " cap=" + getCapacity() +
               " eff=" + getEfficiency() + " pilot on=" +
               pilotLight + " heating=" + isHeating() + " ]";
    }
}
```

[FIGURE 4-12] Code for `GasFurnace.java`

The next piece of code we show is the `ClockListener` interface. This interface is implemented by any object that wants to receive timing information from the `Clock`. It includes the two methods described earlier: `preEvent(double secs)` and `event()`. The `preEvent` method is invoked *prior* to an event taking place. The idea is that when this method is called, an object is given the opportunity to perform any necessary preliminary updates and computations. Only when the `event()` method is invoked is a new value actually assigned to a state variable. The `ClockListener` interface is shown in Figure 4-13.

```
/**
 * Simulation objects that implement this interface are interested
 * in being notified when the time recorded by a clock changes.
 *
```

continued

```
 * There are two methods defined in this interface.  The preEvent()
 * method will be invoked before an event is about to take place.
 * The purpose of this method is to provide a simulation object
 * with the opportunity to update its state before the next event
 * takes place. For example, a room may want to calculate the change
 * in room temperature that occurred since the last event.
 *
 * The event() method indicates that an event has taken place.
 */

public interface ClockListener {

    /**
     * This method is called before the next event occurs so that
     * a simulation object can update its state based on what has
     * occurred since the last event. A simulation object will not
     * change its state when this method is called. It prepares itself
     * for the state change that will occur when the next event occurs.
     *
     * @param interval the number of seconds that have elapsed since
     *        the last event (i.e., the length of the interval
     *        between events).
     */
    public void preEvent( double interval );

    /**
     * Called when the next event occurs in the simulation.
     */
    public void event();

}
```

[FIGURE 4-13] Code for `ClockListener.java`

Now let's take a look at `Room.java`. The code for this class is given in Figure 4-14. First, notice that `Room` implements the `ClockListener` interface of Figure 4-13. This is necessary because `Room` must be sent information from `Clock` when it is time to update its internal room temperature.

The `Room` class includes constructors that use any input values the user provides and default values for quantities not provided. For example, the following constructor:

```
    public Room(Environment theWorld, Furnace heatSource,
                double area);
```

sets the `area` of the room to the user-specified value, but it sets the constants SHC, BLC, and the initial room temperature to default values. Additional constructors exist for other combinations of defaults and user-provided inputs.

The implementation of `Room` includes the following accessor functions for the many state variables associated with objects of this class:

```
getEnvironment(); // Return a reference to the outside Environment
getFurnace();     // Return a reference to the Furnace in this room
getSHC();         // Return the SHC of the room
getBLC();         // Return the BLC of the room
getFloorArea();   // Return the area of the room
getTemperature(); // Return the current room temperature
```

The most interesting implementation issue is how to handle communications between `Room` and `Clock`, which specify when to compute a new temperature. Figure 4-6 identified a method, `determineTemperatureChange(double t)`, that performs this computation. The interesting question is what causes `Room` to execute this method? The answer is the two methods `preEvent` and `event` that are invoked by `Clock`. Because we have specified that `Room` implements `ClockListener`, we can be sure that the code for these two routines exists in `Room`.

The method `preEvent(double secs)` inside `Room` is where we implement the heat gain and heat loss formulas of Section 4.2. The invocation of this method by `Clock` causes the `Room` object to call `determineTemperatureChange`. This method computes the heat gain (by querying `Furnace` to determine its output) and heat loss (by querying `Environment` as to its outdoor temperature) and uses these two values to determine `deltaTemp`, the change in room temperature since the last computation. However, this change is not actually assigned to the state variable temperature until `Clock` invokes the second method, `event()`, which causes `Room` to update its temperature. As we mentioned in Section 4.4.3, this two-step process prevents errors caused by the order in which new values are computed and accessed.

The final thing you should notice about `Room` is that we have again overridden the `toString()` method to provide more descriptive information about the characteristics of `Room` objects. The code for `Room` is shown in Figure 4-14.

```
/**
 * A room in the heating simulation.
 */
public class Room implements ClockListener {
```

continued

```
/**
 * The specific heat capacity (SHC) gives the amount of heat
 * stored in the room.  The SHC for a typical tract house in the
 * United States is 4 BTUs per square foot per degree Fahrenheit.
 */
public static final double DEFAULT_SHC = 4.0;

/**
 * The basic load constant (BLC) gives the resistance of heat flow
 * out of the room.  A large BLC value is indicative of a room
 * that loses heat very easily.
 */
public static final double DEFAULT_BLC = 1.0;
public static final double MINIMUM_BLC = 1.0;
public static final double MAXIMUM_BLC = 10.0;

// State of the room
private Environment outside;  // The outside world
private Furnace myFurnace;    // The furnace heating this room

private double shc;           // The SHC for this room
private double blc;           // The BLC for this room
private double floorArea;     // The floor area of this room
private double temperature;   // The current room temperature

private double deltaTemp;     // The change in room temperature
                              // during the past time interval

/**
 * Create a new room.  The SHC and BLC for the room will be set
 * to the default values.
 *
 * @param theWorld the environment that the room will get the
 *        outside temperature from.
 * @param heatSource the furnace providing heat to the room.
 * @param area the floor area of the room in square feet.
 * @param initialTemperature the initial temperature of the room in
 *        degrees Fahrenheit.
 */
public Room( Environment theWorld, Furnace heatSource,
             double area, double initialTemperature ) {

    this( theWorld, heatSource, DEFAULT_SHC,
          DEFAULT_BLC, area, initialTemperature );
}

/**
 * Create a new room. The SHC and BLC for the room will be set
```

continued

```
     * to the default values. The initial temperature of the room will
     * be set to the current outside temperature.
     *
     * @param theWorld the environment that the room will get the
     *        outside temperature from.
     * @param heatSource the furnace providing heat to the room.
     * @param area the floor area of the room in square feet.
     */
    public Room( Environment theWorld, Furnace heatSource,
                 double area ) {

        this( theWorld, heatSource, DEFAULT_SHC,
              DEFAULT_BLC, area,
              theWorld.getTemperature() ); // outside temperature
    }

    /**
     * Create a new room.
     *
     * @param theWorld the environment that the room will get the
     *        outside temperature from.
     * @param heatSource the furnace providing heat to the room.
     * @param specificHeatCapacity the SHC for this room.
     * @param basicLoadConstant the BLC for this room.  If the value of
     *        this parameter falls outside the valid range for the BLC,
     *        the default BLC value will be used.
     * @param area the floor area of the room in square feet.
     * @param initialTemperature the initial temperature of the room in
     *        degrees Fahrenheit.
     */
    public Room( Environment theWorld, Furnace heatSource,
                 double specificHeatCapacity, double basicLoadConstant,
                 double area, double initialTemperature ) {

        // Initialize the state of the room
        outside = theWorld;
        myFurnace = heatSource;
        shc = specificHeatCapacity;
        floorArea = area;
        temperature = initialTemperature;

        // Make sure the requested BLC is in range
        if ( basicLoadConstant < MINIMUM_BLC ||
             basicLoadConstant > MAXIMUM_BLC ) {
            blc = DEFAULT_BLC;
        }
```

continued

```
        else {
            blc = basicLoadConstant;
        }
    }

    /**
     * Return the environment in which this room is located.
     *
     * @return the environment the room is located in.
     */
    public Environment getEnvironment() {
        return outside;
    }

    /**
     * Return the primary heat source for this room.
     *
     * @return the furnace heating this room.
     */
    public Furnace getFurnace() {
        return myFurnace;
    }

    /**
     * Return the specific heat capacity for this room.
     *
     * @return the SHC for this room.
     */
    public double getSHC() {
        return shc;
    }

    /**
     * Return the basic load constant for this room.
     *
     * @return the BLC for this room.
     */
    public double getBLC() {
        return blc;
    }

    /**
     * Return the floor area of this room.
     *
     * @return the floor area of this room.
     */
```

continued

```
    public double getFloorArea() {
        return floorArea;
    }

    /**
     * Return the current temperature of this room.
     *
     * @return the current temperature of thisroom.
     */
    public double getTemperature(){
        return temperature;
    }

    /**
     * This method will compute the change in room temperature
     * since the last event.
     *
     * Heat into the room comes from the furnace:
     *   Qin = FurnaceOutputBTUsPerHr * timeHours
     *
     * Heat loss is calculated as follows:
     *   Qloss = BLC * FloorArea * ( InsideTemp - OutsideTemp ) *
     *           timeHours
     *
     * The basic load constant (BLC) gives the resistance to heat flow
     * out of the room. The BLC can range from 1 to 10. A large BLC
     * value is indicative of a room that loses heat very quickly
     * (like a car).
     *
     * The change in room temperature during the past interval is given
     * by the formula:
     *   deltaTemp = ( Qin - Qloss ) / ( SHC * floorArea )
     *
     * SHC stands for specific heat capacity of the room and gives
     * the amount of heat that is stored in the room itself (i.e.,
     * in the walls, floors, etc.).  The SHC for a typical tract house
     * in the US is 4 BTUs per square foot per degree Fahrenheit.
     *
     * @param interval the number of seconds of elapsed time.
     */
    public void determineTemperatureChange( double interval ) {
        // The number of hours that have passed since the last event
        double elapsedTimeInHours = interval / Clock.SECS_PER_HOUR;

        // Only heat into the room comes from the furnace
        double qIn = myFurnace.output( elapsedTimeInHours );
```

continued

```
        // Compute the heat that has left the room
        double qLoss = blc * floorArea *
                       ( temperature - outside.getTemperature() )  *
                       elapsedTimeInHours;

        // Compute the change in temperature
        deltaTemp = ( qIn - qLoss ) / ( shc * floorArea );
    }

    /**
     * This method is called before the next event occurs so that it
     * can determine the temperature change that has occurred
     * since the last event.  The room will not change its state when
     * this method is called.  It prepares itself for the state change
     * that will occur when the next event occurs.
     *
     * @param interval the number of seconds that have elapsed since
     *        the last event.
     */
    public void preEvent( double interval ) {
        // Determine the change in room temperature
        determineTemperatureChange( interval );
    }

    /**
     * The room will update its current temperature when
     * this method is called.
     */
    public void event() {
        // Adjust the room temperature
        temperature = temperature + deltaTemp;
    }

    /**
     * Return a string representation of the room.
     *
     * @return a string representation of the room.
     */
    public String toString() {
        return "[ Room: " + " temp=" + temperature +
               " area=" + floorArea + " SHC=" + shc +
               " BLC=" + blc + " ]";
    }
}
```

[FIGURE 4-14] Code for `Room.java`

The Thermostat class is similar in structure and layout to the Room class. It has a number of constructors that handle different combinations of user-specified and default parameter values. For example, the following constructor:

```
public Thermostat(Room theRoom, Furnace theFurnace);
```

creates a new Thermostat inside theRoom that controls a specified furnace called theFurnace. The initial thermostat setting and the amount of allowable overheating are both set to default values.

The Thermostat class contains the following accessor and mutator functions for all of its private instance variables:

```
getRoom();          // Return a reference to the room monitored
                    // by this thermostat

getFurnace();       // Return a reference to the Furnace controlled
                    // by this thermostat

getSetting();       // Return the current thermostat setting

setSetting(double newSetting);  // Reset current thermostat setting

getOverHeat();      // Return the allowable amount of overheating
```

Finally, Thermostat, like Room, also implements the ClockListener interface. When its preEvent() method is invoked, it determines what, if any, changes should be made in the state of the Furnace by calling determineStateChange. This method queries Room to get the current room temperature and determines if the current temperature is either above (setting + overHeat) and the Furnace is on, or if it is below setting and the Furnace is off. It uses the Boolean variable activateFurnace to record the action to be taken. When event() is invoked, Thermostat invokes myFurnace.setHeating(activateFurnace) to set the state of myFurnace.

The code for the Thermostat class is shown in Figure 4-15.

```
/**
 * This class represents a thermostat in the home heating simulation
 * program.  The thermostat monitors a single room and turns on the
 * furnace whenever the room temperature falls below the desired
 * setting.  The thermostat will overheat the room slightly so that
 * the furnace will not cycle on and off too quickly.
 */
```

continued

```
public class Thermostat implements ClockListener {
    // Default overheat setting
    public static final double DEFAULT_OVERHEAT = 3.0;

    // Default temperature setting
    public static final double DEFAULT_SETTING = 72.0;

    private Room myRoom;        // The room being monitored
    private Furnace myFurnace;  // The furnace that will heat the room

    private double setting;     // The desired room temperature
    private double overHeat;    // The amount the room will be
                                // overheated

    private boolean activateFurnace;  // Used to determine if the
                                      // furnace should be turned on

    /**
     * Create a new thermostat with the default settings.
     *
     * @param theRoom the room that will be monitored by this
     *        thermostat.
     * @param theFurnace the furnace that will add heat to the room.
     */
    public Thermostat( Room theRoom, Furnace theFurnace ) {
        this( theRoom, theFurnace, DEFAULT_SETTING, DEFAULT_OVERHEAT );
    }

    /**
     * Create a new thermostat with the default overheat amount.
     *
     * @param theRoom the room that will be monitored by this
     *        thermostat.
     * @param theFurnace the furnace that will add heat to the room.
     * @param desiredTemp the desired room temperature.
     */
    public Thermostat( Room theRoom, Furnace theFurnace,
                       double desiredTemp ) {
        this( theRoom, theFurnace, desiredTemp, DEFAULT_OVERHEAT );
    }

    /**
     * Create a new thermostat.
     *
     * @param theRoom the room that will be monitored by this
     *        thermostat.
     * @param theFurnace the furnace that will add heat to the room.
```

continued

```
     * @param desiredTemp the desired room temperature.
     * @param overheatAmount the amount that the room will be
     *        overheated.
     */
    public Thermostat( Room theRoom, Furnace theFurnace,
                       double desiredTemp, double overheatAmount ) {
        myRoom = theRoom;
        myFurnace = theFurnace;
        setting = desiredTemp;
        overHeat = overheatAmount;
    }

    /**
     * Return the room being monitored by this thermostat.
     *
     * @return the room monitored by this thermostat.
     */
    public Room getRoom() {
        return myRoom;
    }

    /**
     * Return the furnace that will heat the room.
     *
     * @return the furnace that the thermostat will use to add heat to
     *         the room.
     */
    public Furnace getFurnace() {
        return myFurnace;
    }

    /**
     * Return the setting of the thermostat.
     *
     * @return the current setting of the thermostat.
     */
    public double getSetting() {
        return setting;
    }

    /**
     * Return the overheat setting for the thermostat.
     *
     * @return the overheat setting for the thermostat.
     */
```

continued

```
    public double getOverHeat() {
        return overHeat;
    }

    /**
     * Set the desired temperature setting for the thermostat.
     *
     * @param newSetting the new temperature setting for the
     *        thermostat.
     */
    public void setSetting( double newSetting ) {
        setting = newSetting;
    }

    /**
     * Base the decision to turn on/off the furnace on the temperature
     * of the room during the past time period.
     *
     * @param interval the number of seconds that have elapsed since
     *        the last event.
     */
    public void determineStateChange( double interval ) {
        double roomTemp = myRoom.getTemperature();

        if ( activateFurnace ) {
            // If the furnace is on, leave it on until the room
            // temperature is equal to or greater than the desired
            // setting plus the overheat amount
            activateFurnace = roomTemp < setting + overHeat;

        }
        else {
            // If the furnace is currently off, it should stay off
            // until the room temperature falls below the desired
            // setting.
            activateFurnace = roomTemp < setting;
        }
    }

    /**
     * This method is called before the next event occurs so that
     * the thermostat can determine what it should do to the
     * furnace (i.e., turn it on, turn it off, or leave it alone) based
     * on what has happened since the last event.  The thermostat
     * will not change its state when this method is called.
     * It prepares itself for the state change that will occur when the
     * next event occurs.
```

continued

```
     *
     * @param interval the number of seconds that have elapsed since
     *        the last event.
     */
    public void preEvent( double interval ) {
        determineStateChange( interval );
    }

    /**
     * Turn on/off the furnace based on the temperature of the room.
     */
    public void event() {
        myFurnace.setHeating( activateFurnace );
    }

    /**
     * Return a string representation of the thermostat.
     *
     * @return a string representation of the thermostat.
     */
    public String toString() {
        return "[ Thermostat: " + " setting=" + setting +
               " overheat=" + overHeat + "]";
    }
}
```

[FIGURE 4-15] Code for `Thermostat.java`

Our final piece of code (except for `main()`) is the master clock of our simulation program, which is an instance of class `Clock`. Each increment of the clock represents the passing of `tickInterval` seconds, a constant that is set when the clock is created:

```
public Clock(int tickInterval); // num of seconds per tick
```

In our simulation, `tickInterval` is automatically set to 60 seconds. However, as discussed earlier, we may want to allow users to set this parameter themselves, either to increase accuracy or decrease the computational load.

The `run(int numTicks)` method of `Clock` activates the simulation, as it loops from `i` = 0 to `i` = `numTicks`, where each of these individual ticks represents the passing of `tickInterval` seconds. It is the `run()` method that truly starts the simulation process.

The last interesting aspect of `Clock` is the `listeners[]` array. This array of objects implements the `ClockListener` interface of Figure 4-13 and is notified every time the clock ticks. Each time the clock ticks, it first calls the method `listeners[j].preEvent(secsPerTick)`on every object in the `listeners[]` array. Next, it calls `listeners[j].event()` on every object in the `listeners[]` array. The value `secsPerTick` is a state variable that specifies how many seconds have passed since the last tick of the clock (that is, it has the same value as `tickInterval`).

We can see from the UML diagram in Figure 4-7 that the two methods that implement the `ClockListener` interface are `Room` and `Thermostat`. The problem now is how to indicate that these two objects should be placed in the `listeners[]` array.

The answer is that we **register** these two methods with the `Clock`. Registration is a common programming technique that invokes a method that adds a specific object name to a collection. This collection of objects is then given certain privileges or is treated in some special way. In our case, registration allows objects to receive timing information via the `preEvent` and `event` methods.

In this example, our registration method is `add(ClockListener entity)`, which adds `entity` to our `listeners[]` array. Because `listeners[]` is an array structure, it has a fixed size: 10 elements in our case. If we tried to add an 11th object, there wouldn't be any room, and the array would overflow. Instead of generating a run-time error and terminating the program, we handle this problem by *resizing* the array. We dynamically create a new array that is twice its current size, and then copy the elements of the current array into the new one. This allows the simulation to continue executing and produce the desired results. Dynamic array resizing is a common technique used by many of the library routines in Java.

The next section discusses a data structure—the linked list—that can hold an arbitrarily large number of objects. In addition, we will see that this data structure is part of an existing package called the Java Collection Framework, which is freely available to any Java programmer. Taking advantage of this feature eliminates the need for such mundane tasks as resizing arrays.

The code for `Clock` is shown in Figure 4-16.

```
/**
 * The master clock for the home heating simulation. Generates tick
 * events for ClockListeners that have been registered with the
 * clock.  The interval between ticks is determined at the time the
 * clock is constructed.
 */
public class Clock {

    // Number of seconds in one minute
    public static final int SECS_PER_MINUTE = 60;

    // Number of seconds in one hour
    public static final int SECS_PER_HOUR = SECS_PER_MINUTE * 60;
```

continued

```
// Initial size of the array that holds the listeners
private static final int INITIAL_SIZE = 10;

// Objects to be notified when tick events occur
private ClockListener listeners[];

// The number of listeners registered with this clock
private int numListeners;

// The number of seconds that pass between ticks
private int secsPerTick;

/**
 * Create a new clock that will generate tick events at
 * the specified interval.
 *
 * @param tickInterval the number of seconds between ticks.
 */
public Clock( int tickInterval ) {
    // Set up the array that will hold the listeners
    listeners = new ClockListener[ INITIAL_SIZE ];
    numListeners = 0;

    // The time interval for this clock
    secsPerTick = tickInterval;
}

/**
 * Run the clock for the specified number of ticks.  The state of
 * the clock is valid between invocations of run().  This makes it
 * possible to call run multiple times within the simulation.
 *
 * @param numTicks the number of ticks to generate
 */
public void run( int numTicks ) {
    for ( int i = 0; i < numTicks; i++ ) {

        // Notify the listeners that an event is about to happen
        for ( int j = 0; j < numListeners; j++ ) {
            listeners[ j ].preEvent( secsPerTick );
        }

        // The event occurs
        for ( int j = 0; j < numListeners; j++ ) {
            listeners[ j ].event();
        }
    }
}
```

continued

```
    /**
     * Add a listener to the collection of objects that are
     * notified when tick events occur.
     *
     * @param entity the listener to add to the collection
     */
    public void add( ClockListener entity ) {
        // Resize the array if it is full.  A new array will be created
        // with twice the capacity and the contents of the old array
        // will be copied into the new array.
        if ( numListeners == listeners.length ) {

            // Create a new array that is twice as large
            ClockListener newListeners[] =
            new ClockListener[ listeners.length * 2 ];

            // Copy the old array into the new array
            for ( int i = 0; i < listeners.length; i++ ) {
                newListeners[ i ] = listeners[ i ];
            }

            // Make the class use the new array to keep track of the
            // listeners
            listeners = newListeners;
        }

        // Add the listener to the array
        listeners[ numListeners ] = entity;
        numListeners = numListeners + 1;
    }
}
```

[FIGURE 4-16] Code for `Clock.java`

The final class we develop is `HeatingSimulation.java`, the class that includes the `main()` method for the entire program. (Remember that we initiate the program with the command `java HeatingSimulation`.) This class contains mostly initialization and administrative tasks. For example, this is where we process the command line, access the user-specified parameters, make sure they are legal, and store them in an array called `simParams`. This class is responsible for producing the output report shown in Figure 4-3. The program imports and uses the Java class `DecimalFormat` in package `java.text` to assist in printing the output in an elegant and appropriate format.

`HeatingSimulation` also creates the objects used in the simulation—in this example, one room, one furnace, one thermostat, one environment, and `masterClock`, the master simulation clock. It also registers the room and the thermostat with `Clock` and turns on the furnace's pilot light so that it can produce heat. Finally, it executes the statement that starts the simulation:

```
masterClock.run( (int)simParams[DISPLAY_FREQ] );
```

This statement does not run the entire simulation. Instead, it runs the simulation for one "output display" unit of time. In the sample output in Figure 4-3, this is 5 minutes, the time interval between successive output lines. (Because the clock ticks in 1-minute intervals, this represents five ticks of the simulation clock.) When the simulation time has passed, the program prints a single line of output and invokes the run method again. This run-output loop continues until the master simulation clock has reached or exceeded the value of the parameter `length`, shown in Figure 4-2, which is the total time the simulation is to run.

The code for `HeatingSimulation` is shown in Figure 4-17. This code completes the implementation of our home heating simulation case study.

```
import java.text.DecimalFormat;

/**
 * The program that runs the heating simulation.  The heating
 * system modeled by this program consists of a single room heated
 * by one furnace which is controlled by a single thermostat.
 */
public class HeatingSimulation {

    // The simulation will always advance time in 60 second units
    private static final int SECS_BETWEEN_EVENTS = 60;

    // An array will be used to store the simulation parameters.  This
    // will make it easier to write the code that parses the command
    // line and sets these parameters.  The constants below identify
    // the parameter that is stored in the corresponding position in
    // the array

    // Inside temperature
    private static final int INSIDE_TEMP = 0;
    // Outside temperature
    private static final int OUTSIDE_TEMP = 1;
    // Desired temperature
    private static final int DESIRED_TEMP = 2;
    // Furnace capacity
    private static final int FURNACE_CAPACITY = 3;
    // Furnace efficiency
```

continued

```
    private static final int FURNACE_EFFICIENCY = 4;
    // Room size (sq ft)
    private static final int ROOM_SIZE = 5;
    // Display freq (mins)
    private static final int DISPLAY_FREQ = 6;
    // Time to run (mins)
    private static final int SIM_LENGTH = 7;

    // The array to hold the values of the parameters.  The initializer
    // is used to set the default value for each parameter.
    private static double simParams[] = {
                                    72.0,    // Inside temperature
                                    50.0,    // Outside temperature
                                    72.0,    // Desired temperature
                                    50000.0,// Furnace capacity
                                    .90,     // Furnace efficiency
                                    250.0,   // Room size
                                    5.0,     // Ticks between output
                                    7200.0   // Time to run (secs)
    };

    // This array holds the names of the parameters that will be used
    // on the command line.  Each name is stored in the same position
    // as the corresponding value in the simParams[] array.
    private static String simNames[] = {"in",    // Inside temperature
                                    "out",   // Outside temperature
                                    "set",   // Desired temperature
                                    "cap",   // Furnace capacity
                                    "eff",   // Furnace efficiency
                                    "size", // Room size
                                    "freq", // Display frequency
                                    "length"// Time to run
    };

    public static void main( String args[] ) {
        // Format used to print report
        DecimalFormat fmt = new DecimalFormat( "###0.00" );

        // The references to the objects that make up the simulation.
        // In this simulation there is one room controlled by one
        // thermostat and heated by one furnace.
        GasFurnace theFurnace = null;
        Environment theWorld = null;
        Room theRoom = null;
        Thermostat theThermostat = null;
```

continued

```
// Process the command line arguments
processCommandLine( args );

// Create a furnace, a room, a thermostat, and an environment.
theFurnace = new GasFurnace( simParams[ FURNACE_CAPACITY ],
                             simParams[ FURNACE_EFFICIENCY ] );

theWorld = new Environment( simParams[ OUTSIDE_TEMP ] );

theRoom = new Room( theWorld, theFurnace,
                    simParams[ ROOM_SIZE ],
                    simParams[ INSIDE_TEMP ] );

theThermostat = new Thermostat( theRoom, theFurnace,
                                simParams[ DESIRED_TEMP ] );

// Create the clock that will drive the simulation and register
// the room and the thermostat with the clock so that they
// will be notified when events occur within the simulation
Clock masterClock = new Clock( SECS_BETWEEN_EVENTS );
masterClock.add( theRoom );
masterClock.add( theThermostat );

// Turn on the pilot light so the furnace will produce heat
theFurnace.setPilot( true );

// Print out the objects that were created
System.out.println( "Created 4 Objects:" );
System.out.println( "   " + theFurnace );
System.out.println( "   " + theWorld );
System.out.println( "   " + theRoom );
System.out.println( "   " + theThermostat );
System.out.println();

// Run the simulation for the requested time period. When
// displayFrequency seconds of simulated time have passed, the
// current state of the objects within the simulation will be
// displayed
System.out.println( "Starting simulation:  " +
                    " display frequency=" +
                    simParams[ DISPLAY_FREQ ] +
                    " runtime(sec.)= " +
                    simParams[ SIM_LENGTH ] + "\n" );

System.out.println("Time\tInside\tOutside\tDesired\tFurnace" );
System.out.println("----\t------\t-------\t-------\t-------" );
```

continued

```
        for ( int simTime = 0; simTime <= (int)simParams[ SIM_LENGTH ];
              simTime = simTime + (int)simParams[ DISPLAY_FREQ ] *
              SECS_BETWEEN_EVENTS) {

            // Print the statistics
            System.out.print( simTime + "\t" +
                              fmt.format(theRoom.getTemperature()) +
                              "\t" +
                              fmt.format(theWorld.getTemperature()) +
                              "\t" +
                              fmt.format(theThermostat.getSetting()) );

            if ( theFurnace.isHeating() ) {
                System.out.print( "\tOn" );
            }

            System.out.println();

            // Run the simulation for display frequencies in seconds
            masterClock.run( (int)simParams[ DISPLAY_FREQ ] );
        }
    }

    /**
     * Scan the command line arguments and set any simulation
     * parameters as specified by the user.  Invalid parameter
     * settings will be ignored.
     * Note that if an invalid numeric value is specified on the
     * command line a run-time exception will be thrown and the program
     * will terminate.
     *
     * @param args the parameter settings to parse.
     */
    private static void processCommandLine( String args[] ) {
        // Step through the settings...
        for ( int i = 0; i < args.length; i++ ) {
            // Parameter settings take the form:  name=value
            int equals = args[ i ].indexOf( '=' );

            // If there is an equals sign in the setting then it might
            // be valid
            if ( equals != -1 ) {
                // Extract the name and the value
                String paramName = args[ i ].substring( 0, equals );
                String paramValue = args[ i ].substring( equals + 1 );
```

continued

```
                    // The index into the simParams array where the setting
                    // is to be made. A value of -1 indicates that the name
                    // is invalid
                    int loc = -1;

                    // Search for the name in the names array. Because the
                    // name is stored in the same position as the
                    // corresponding value, once the location of the name
                    // is determined we know where the value is stored
                    for (int j = 0; loc == -1 && j < simNames.length; j++){
                        if ( paramName.equals( simNames[ j ] ) ) {
                            loc = j;
                        }
                    }

                    // If the name is valid set the parameter.  Note that
                    // an invalid value entered on the command line will
                    // cause a run-time exception and terminate the program.
                    if ( loc != -1 && paramValue.length() > 0 ) {
                        simParams[ loc ] = Double.parseDouble(paramValue);
                    }
                }
            }
        }
    }
```

[FIGURE 4-17] Code for `HeatingSimulation.java`

SIMULA—THE ORIGINAL OBJECT-ORIENTED LANGUAGE

Object-oriented design is a fairly recent development, becoming popular only in the last 10 to 20 years. Similarly, most object-oriented languages are relatively new. Java, the language of this text, became available for widespread use in October 1994. Visual BASIC (VB) hit the market in 1991. Python, a multiparadigm language that is gaining in popularity, was developed in 1990 using the open source model, as described earlier in this chapter. C++, a popular object-oriented language, had its commercial release in 1985, while a more recent object-oriented update of C, C#, appeared in 2001. Thus, most object-oriented languages are only 5 to 20 years old.

continued

However, the idea of a language based on classes and objects is not new, and the languages just mentioned—Java, C#, VB, Python—all stem from work originally done more than 40 years ago by Kristen Nygaard and Ole-Johan Dahl at the Norwegian Computer Center in Oslo. Nygaard and Dahl were interested in creating a language for building discrete event simulations. They designed a language called Simula, which was based on the concept of classes and objects originally proposed by C.A.R. Hoare of Oxford University in 1966. (We will read more about Professor Hoare in the Challenge Work Exercise at the end of Chapter 5.) Nygaard and Dahl had the idea that each entity being simulated in the program could be described as a class that encapsulated its state and behavior. They worked on the design of this new language for more than a year and presented it to the computer science community at a conference in May 1967, which is why the language is more popularly known as Simula-67.

It became an extremely important language that had an impact far beyond its original use for simulation. Its design influenced virtually all the object-oriented languages that came after it. In November 2001, Nygaard and Dahl were awarded the IEEE John Von Neumann Medal by the Institute of Electrical and Electronic Engineers "for the introduction of the concepts underlying object-oriented programming through the design and implementation of Simula-67." In February 2002, they received the A.M. Turing Award from the Association for Computing Machinery (ACM) for ideas fundamental to the emergence of object-oriented programming. Sadly, Nygaard and Dahl died within six weeks of each other, a few months after winning this prestigious award.

4.6 Testing

Although this section comes at the end of the chapter, following the discussion on implementation, we do *not* imply that testing should be delayed until all the code has been written. Postponing testing until the end of a software development project is a sure recipe for disaster—including budget overruns, missed delivery dates, and "buggy" code. Thorough, complete, and intense testing of every piece of code is an essential part of software development.

The initial phase of testing is called **unit testing**, as first described in Section 1.2.6. During this testing you thoroughly test each unit of code (class, method, or subclass) that you write, as soon as you write it. You never place any source code into a library until it has been thoroughly checked using a carefully planned and well-designed set of test cases, and it has successfully passed 100% of these unit tests.

For example, the first piece of code we developed was `Environment`, shown in Figure 4-10. Next, we discussed the implementation of class `Furnace`. However, in a real-world project, we would have thoroughly tested all aspects of the `Environment` class to ensure they worked correctly before ever starting to write the code for `Furnace`. Even though `Environment` is small and quite simple, a number of cases still must be carefully checked. In the case of `Environment`, we must test each of the following:

- The *default constructor*, to ensure that it sets the temperature to the correct default value
- The *one-parameter constructor*, to ensure that it sets the temperature to the specified parameter
- The *accessor method*, `getTemperature()`
- The *mutator method*, `setTemperature(t)`
- The *overridden method* `toString()`

Figure 4-18 shows the code for a class called `TestEnvironment`, whose task is to test each of these five cases.

```
/**
 * A test program for the environment class.
 */
public class TestEnvironment {

    public static void main( String args[] ) {
      // First let's use both the default constructor
      // and the one-parameter constructor
      Environment e1 = new Environment();
      Environment e2 = new Environment( 85.0 );

      // Determine if the constructors and
      // the getTemperature method works
      System.out.println("Temperature of e1 = " + e1.getTemperature());
      System.out.println("Temperature of e2 = " + e2.getTemperature());

      // See if we can change the temperature
      e1.setTemperature( 75.0 );
      System.out.println( "New temperature of e1 = " +
                          e1.getTemperature() );

      e2.setTemperature( 90.0 );
      System.out.println( "New temperature of e2 = " +
                          e2.getTemperature() );
```

continued

```
        // Determine if toString() works as expected
        System.out.println( "Object e1 = " + e1 );
        System.out.println( "Object e2 = " + e2 );

        System.out.println( "End of unit test of Environment" );
      }

  }
```

[FIGURE 4-18] Test program for the `TestEnvironment` class

When this program is run, the expected output is the following:

```
Temperature of e1 = 72.0
Temperature of e2 = 85.0
New temperature of e1 = 75.0
New temperature of e2 = 90.0
Object e1 = [ Environment: temperature=75.0 ]
Object e2 = [ Environment: temperature=90.0 ]
End of unit test of Environment
```

Because we are using default formatting, your output might look slightly different. For now we are only concerned with correctness, not the exact layout.

If the expected output is produced when the test program is executed, we can be reasonably confident that class `TestEnvironment` is working according to specifications.

Because of its simplicity, we can probably stop after one test suite. With more complex classes, we would run multiple test programs.

If the expected output does not appear, we must immediately locate the bugs in this unit, correct them, and rerun the test suite. However, this should be relatively simple because we are only examining a single class, not the 1100 lines of code in the entire software package.

When we feel completely confident that the unit is correct, we put it into our library and then write and test the next program unit, `Furnace`, assuming that these two classes are being developed by the same individual or team. If separate groups were working on these two processes, they would be coding and testing these components in parallel.

The `Furnace` class of Figure 4-11 would be tested in a similar way. We must test each of the following conditions:

- The two-parameter constructor
- The three accessor methods `getCapacity()`, `getEfficiency()`, and `isHeating()`
- The mutator method `setHeating(onOff)`
- The method `output(double hours)`, which computes heat output
- The overridden method `toString()`

Furthermore, the `output` method has two distinct cases we must test:

- Computation of the heat output when the furnace is on
- Computation of the heat output when the furnace is off

Figure 4-19 shows a test program `TestFurnace` for our `Furnace` class.

```
/**
 * Program to test the furnace class.
 */
public class TestFurnace {

    public static void main( String args[] ) {
        // First let's invoke the constructor
        Furnace f1 = new Furnace( 10000.0, 0.78 );
        Furnace f2 = new Furnace( 30000.0, 0.85 );
        Furnace f3 = new Furnace( 50000.0, 0.93 );

        // Determine if the constructors worked and if the
        // getCapacity and getEfficiency methods work
        System.out.println( "f1 Capacity = " + f1.getCapacity() +
                            " Efficiency = " + f1.getEfficiency() +
                            " Heating state = " + f1.isHeating() );

        System.out.println( "f2 Capacity = " + f2.getCapacity() +
                            " Efficiency = " + f2.getEfficiency() +
                            " Heating state = " + f2.isHeating() );

        System.out.println( "f3 Capacity = " + f3.getCapacity() +
                            " Efficiency = " + f3.getEfficiency() +
                            " Heating state = " + f3.isHeating() );

        // Now see if we can change the state of a Furnace
        f1.setHeating( true );
        System.out.println( "New heating state of f1 = " +
                            f1.isHeating() );
```

continued

```
        f2.setHeating( true );
        System.out.println( "New heating state of f2 = " +
                            f2.isHeating() );

        f3.setHeating( true );
        System.out.println( "New heating state of f3 = " +
                            f3.isHeating() );

        // See if we can turn it off again
        f3.setHeating( false );
        System.out.println( "New heating state of f3 = " +
                            f3.isHeating() );

        // Now let's compute the output of each of these
        // furnaces for 1 hour, 2 hours, and 3 hours. Because
        // furnace f3 is off, it won't produce any heat
        System.out.println( f1.output( 1.0 ) );
        System.out.println( f1.output( 2.0 ) );
        System.out.println( f1.output( 3.0 ) );

        System.out.println( f2.output( 1.0 ) );
        System.out.println( f2.output( 2.0 ) );
        System.out.println( f2.output( 3.0 ) );

        System.out.println( f3.output( 1.0 ) );
        System.out.println( f3.output( 2.0 ) );
        System.out.println( f3.output( 3.0 ) );
        // Determine if toString() works as we would expect
        System.out.println( "Object f1 = " + f1 );
        System.out.println( "Object f2 = " + f2 );
        System.out.println( "Object f3 = " + f3 );

        System.out.println("End of unit test of Furnace");
    }
}
```

[FIGURE 4-19] Test program for the `TestFurnace` class

When it is run, the output of the program in Figure 4-19 should be the following:

```
f1 Capacity = 10000.0 Efficiency = 0.78 Heating state = false
f2 Capacity = 30000.0 Efficiency = 0.85 Heating state = false
f3 Capacity = 50000.0 Efficiency = 0.93 Heating state = false
New heating state of f1 = true
New heating state of f2 = true
New heating state of f3 = true
New heating state of f3 = false
```

```
7800.0
15600.0
23400.0
25500.0
51000.0
76500.0
0.0
0.0
0.0
Object f1 = [ Furnace: cap=10000.0 eff=0.78 heating=true ]
Object f2 = [ Furnace: cap=30000.0 eff=0.85 heating=true ]
Object f3 = [ Furnace: cap=50000.0 eff=0.93 heating=false ]
   End of unit test of Furnace
```

If this is the output of the test program, we can again feel confident about the correctness of our code. We would put `Furnace` into our library and move on to the implementation and testing of the next class. We leave the design of the remaining test programs as an exercise for the reader.

Looking back at the `Furnace` test program in Figure 4-19, it may seem like overkill to write a 45-line program to test a class that contains only about 100 lines. However, one of the most important rules of modern software development is: *Never skimp on testing!*

If you must write a 50-line test program to adequately test 50 lines of developed software, then so be it. If your code is not correct, it doesn't matter how efficient, elegant, maintainable, and robust it may be. It won't be of use to anyone.

The second phase of testing is called **acceptance testing**, and during this phase the software is placed into the environment in which it will be regularly used. Rather than selecting data based on testing specific classes, methods, and flow paths, the program is tested with real-world data that reflects the typical operating conditions the program will experience when used on a daily basis. These test cases are usually selected by the user and are frequently included in the program specifications.

If the program operates successfully on these acceptance cases, it is deemed to be finished and is delivered to the user. However, we all know that software is never truly finished, and it will likely be updated, modified, and adapted for many years to come. That is why maintenance is such a critically important part of the software development life cycle.

4.7 Summary

We hope that this extended case study clarified the many important ideas presented in this section of the book. By observing the development of a program from its initial problem statement to implementation and testing, you can better understand the many steps of the software development process. However, to truly appreciate this process, it is not enough to observe it; you must also *try* it. The end of this chapter includes suggestions for projects you can work on, either individually or as part of a development team. We strongly encourage you to design and implement one of these projects using the techniques presented in this chapter.

In the next section of the book, we will investigate a new subject that is critical to the success of any significant software project—the topics of algorithms and data structures. The case study we just completed did not use any interesting data structures, except for a single one-dimensional array. However, virtually all real-world problems require more complex data structures, such as lists, stacks, priority queues, binary trees, hash tables, or graphs. The intelligent use of these structures allows us to create faster and more efficient programs.

We will take an in-depth look at these data structures along with the algorithms required to manipulate them and the mathematical tools needed to analyze them. We will also introduce the Java Collection Framework, a set of classes and methods that makes many of these algorithms and data structures available to every Java programmer.

EXERCISES

In Exercises 1, 2, and 3, use the existing home heating simulator code in this chapter to design and run experiments to answer the following questions:

1. Set the outdoor temperature to 32 degrees Fahrenheit, the initial and desired room temperature to 70 degrees Fahrenheit, and the room size to 1000 square feet. Test each of the following four furnaces and determine how long the furnace is in the ON state during a 5-hour period. Print one line of output every five minutes and assume that if the output line states that the furnace is ON, then it has been on for the entire 5-minute period.

   a. Capacity = 70,000 Efficiency = 0.90
   b. Capacity = 73,000 Efficiency = 0.88
   c. Capacity = 75,000 Efficiency = 0.82
   d. Capacity = 80,000 Efficiency = 0.78

   From the output produced by the simulator, answer the question "Which of these furnaces is most cost efficient at keeping the house at 70 degrees Fahrenheit?", where efficiency means that the furnace has been on for the least amount of time.

2. Determine what percentage of savings we could expect if we used the same model parameters from Exercise 1 but lowered the initial and desired room temperatures to:

   a. 68 degrees Fahrenheit
   b. 65 degrees Fahrenheit

   Assume that costs are directly proportional to how much time the furnace is on. You only need to test the furnace from Exercise 1 that you determined to be most efficient.

3. In the chapter, we stated that making a good choice for the time increment Δ is critical to the efficient behavior of our software. If it is too small, then the program performs an excessive amount of computation to solve the problem. If it is too large, then we may get highly inaccurate results. In our simulator we set Δ to 60 seconds.

Run the program using the same parameters in Figure 4-3. Now make the following two changes and answer the following questions.

a Set $\Delta = 0.0001$ seconds and run the simulation for two hours (7200 seconds), keeping all other values unchanged. How much longer does it take to run the program and produce the same report shown in Figure 4-3? How much did the accuracy change? Was it worthwhile to use such a small value of Δ?

b Set $\Delta = 600$ seconds (10 minutes) and run the simulation for two hours, keeping all other values unchanged. How accurate were the results? Did the model show any strange or unusual behavior? Was this value of Δ acceptable?

4 Modify the model so that the user can provide input values for the two constants SHC and BLC on the command line. Here are the specifications:

```
parameter name: shc  Default value: 4.0
parameter name: blc  Default value: 1.0
```

Your model should now be able to accept command lines that look like this:

```
java HeatingSimulation cap = 10000 shc = 5.5 blc = 1.5
```

5 Carefully review the problem specification document shown in Figure 4-4. Are any important pieces of information omitted from this document that could cause future errors or omissions? Are there any ambiguities or inconsistencies that could create problems during design and implementation? Critique the quality of the specifications in this document, and discuss how you might have written them differently.

6 Modify the `Environment` class so that, instead of a fixed outdoor temperature, the temperature varies as a function of the `Clock` value. Assume that $\Delta = 60$ seconds, and that the outdoor temperature varies in the following way:

- For the first 480 clock ticks (8 hours), the temperature increases by 0.04 degrees Fahrenheit at each tick.
- For the next 480 clock ticks (8 hours), the temperature remains constant.
- For the next 480 clock ticks (8 hours), the temperature decreases by 0.04 degrees Fahrenheit at each tick.

If the running time of the model is more than 24 hours, simply repeat this cycle. If it is less than 24 hours, then run as many ticks as the user specified, even if you don't get through the cycle completely. Compare the behavior of the model with the output of the constant temperature `Environment` shown in Figure 4-3.

7 Assume that our model is modified to use radiators to heat the living area. The main difference between radiators and forced-air heaters is that a radiator stays warm for a while, even after the furnace is turned off. Thus, there is some continuing heat gain for a period of time.

Write a new class called `RadiatorHeat`, which is a subclass of `Furnace`. When this radiator-based furnace is on, the heat output is computed exactly as described in the chapter. However, when the `Furnace` is off, rather than producing a heat output of 0, the heat output is computed as follows:

- One clock tick after the furnace was turned off, the heat output becomes two-thirds of what it was when the furnace was on.
- Two clock ticks after the furnace was turned off, the heat output becomes one-third of what it was when the furnace was on.
- Three clock ticks after the furnace was turned off and continuing until the furnace is turned back on, the heat output becomes 0.

8 a Write a UML sequence diagram for the `Thermostat` class.

b Discuss the benefits that such diagrams can have for a programmer who is trying to implement code.

9 Explain why the values "1 .. *" appear with the arrow from class `Clock` to the `ClockListener` interface in the UML diagram shown in Figure 4-7. Why are all other values in that diagram set to 1?

10 Write test programs to thoroughly test the following classes in our simulator. Explain why you designed the test program as you did, and give the output you would expect to see from your test program when it is run.

a `Room.java`, as shown in Figure 4-14

b `Thermostat.java`, as shown in Figure 4-15

CHALLENGE WORK EXERCISES

The following projects describe simulations that individual students or teams can implement using the techniques described in this chapter. The following specifications represent only a preliminary set of requirements. You must produce a complete specification document before design can begin.

Simulation Project 1

You are to write a program to simulate the control tower for a local airport. Although the airport does not operate exactly like a normal airport, your experience with airports can help you to visualize the system operation.

The model is made up of an arbitrary number of independent runways; each runway has two associated waiting lines. One of the lines, the arrival queue, contains a list of all the airplanes waiting to land on this runway, and the other, the departure queue, contains a list of all the airplanes waiting to use the runway to leave the airport. Each of the runways is functionally equivalent. None are reserved for landings or departures or for particular planes, for example. In addition, planes can land on different runways at the same time. The number of runways is determined at run time by having the program read a number from standard input.

When an airplane is generated to arrive or depart at this airport (for example, enter a waiting line), the control tower looks at the queues for each of the runways and determines which has the shortest wait before access to the runway can be granted—that is, which runway has the fewest planes waiting. Airplanes that want to land at the airport have priority over airplanes that want to depart.

The waiting time for an airplane that wants to land on a runway is the sum of the following:

1. The amount of time it takes for the plane currently using the runway to complete its arrival or departure
2. The amount of time associated with each plane already in the arrival queue

The wait for an airplane that wants to leave the airport is the sum of the following:

1. The amount of time it takes for the plane currently using the runway to complete its arrival or departure
2. The amount of time associated with each plane in the arrival queue. (Note that this is an approximation, as new arrivals may enter the system while we are waiting, which could increase the waiting time.)
3. The amount of time associated with each plane already in the departure queue

Whenever a runway is not busy, the next airplane in the arrival queue is assigned to the runway; that is, the airplane is removed from the arrival queue and proceeds directly to the runway. If no airplanes are in the arrival queue, the first airplane in the departure queue is removed from the queue and proceeds directly to the runway.

The first airplane that requests a runway in the simulation is assigned to runway 0. The system remembers that runway 0 was the last runway to have an airplane assigned to it. When assigning other airplanes to runways, the control tower looks first at the next runway numerically after the one that last had an airplane assigned to it—in this case, runway 1. The search process is circular, which means that when the last runway is examined, the next runway to examine is runway 0.

The simulation is driven by the following five pieces of input:

- A seed to be used with the random number generator
- The average arrival rate of airplanes requesting arrival or departure
- The average amount of time for an airplane to arrive or depart
- The number of runways at this airport
- The amount of time that the simulation should run

After all input has been read and all objects in the system have been created, the program enters the simulation loop. Each iteration through the simulation loop represents the passage of one unit of time. During each single time period, or tick, the program performs the following steps:

1. The airplane generator is asked for a possibly empty list of airplanes that want to arrive or depart.
2. Each airplane is assigned to the runway that allows it to arrive or land at the soonest available time. At the first tick, the control tower starts checking at runway 0.
3. Each runway determines if the current airplane requires any more time on the runway. If the current airplane has left the runway, then statistics are updated; if another airplane is in either queue, the next airplane proceeds to the runway.
4. The clock used by the simulation is updated to indicate that one unit of time has passed.

After the simulation is completed, the following information is printed:

- Average time an airplane is on a runway
- Average time an airplane must wait to use a runway (the time the plane is in a queue)
- Average idle time of all the runways
- The final contents of both queues associated with each runway, starting with runway 0

After this information is printed, the simulator can terminate.

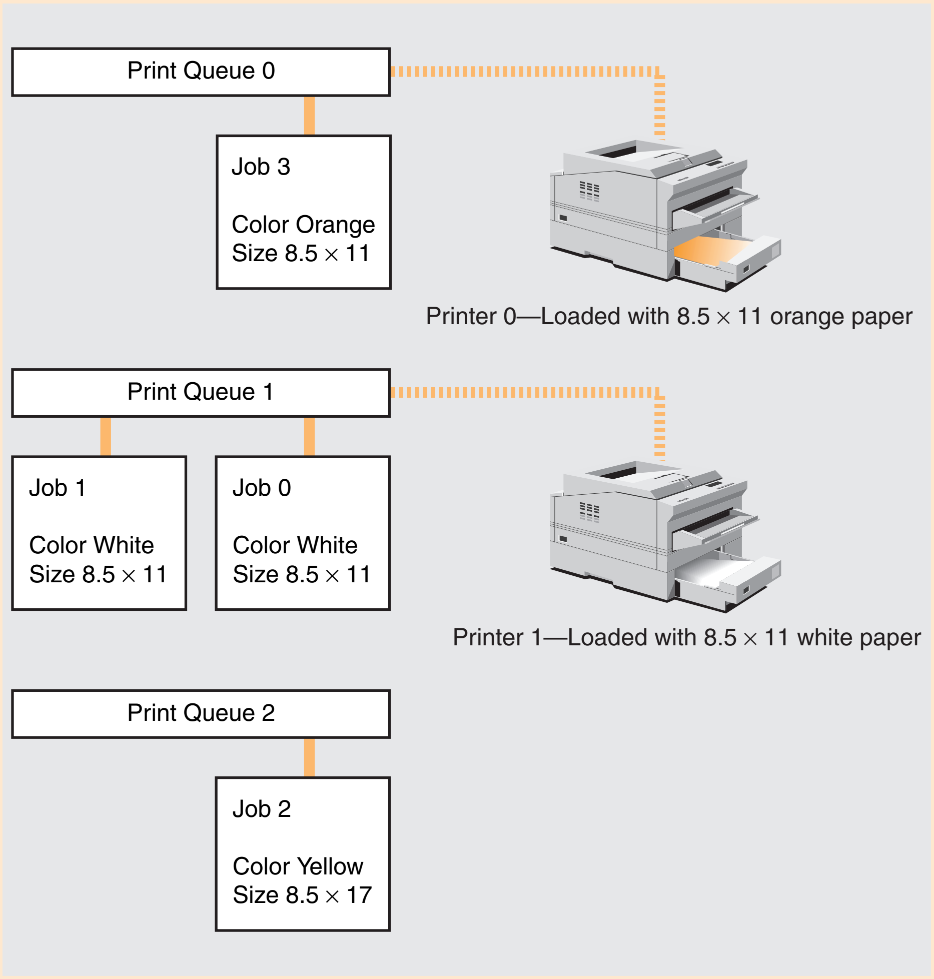
Simulation Project 2

You are to write a program that simulates the running of a large print shop. Users make electronic print requests in the form of jobs. Each job is picked up and routed to a printer that suits the job's needs. We refer to these needs as resources; they include paper size and color, stapling or binding options, sorting options, and ink or toner colors.

Some resources are basic capabilities of the printer and are not easy to change. Others, such as paper and ink, can be changed as frequently as needed. This leads to some problems. What if:

- A job request is made for paper that is not loaded into any printer?
- A job is queued up to print, but the operator changes the printer so it no longer can do the job?

This is where the idea of print queues comes in. A print queue is created for every combination of resources that any job needs. If a printer offers this set of resources, then the print queue feeds jobs to it. If not, the print queue simply queues up jobs, waiting for some printer to be reconfigured to fit its needs, as shown in Figure 4-20.



[FIGURE 4-20] Printer simulation

This diagram shows two printers: one is loaded with orange paper and the second with white paper. The queues that are holding requests for orange and white paper are connected to printers. The third queue is not connected to a printer, because no printer is loaded with yellow paper. Before the job waiting for yellow paper can print, the paper in one of the printers must be changed.

Simulation ■ The simulation is driven by a data file that is logically divided into a printer and simulation section. The number of printers and their initial configuration are defined in the printer section. The simulation section contains information about print requests, in the form of jobs, made by system users. As job information is processed, the program assigns an expected duration time to the job, creates the job, and places it in the appropriate queue.

When the printer simulation begins, the program opens the data file and processes the printer section. At this point, all printers in the system have been created and are ready to accept jobs. The program then enters the simulation loop.

Each iteration through the simulation loop represents the passage of one unit of time. During a single time period, or tick, the printers are informed that one time unit has passed, and then lines from the simulation section of the data file are processed. The loop repeats when all of the lines from the simulation section have been processed or a line indicates that all new jobs for this time period have been processed. The loop terminates when the entire simulation section has been processed and all the jobs have finished printing.

In this project, no actual printing occurs. Instead the process is simulated. In the simulation, a piece of software receives the print request and claims to have printed it, even though nothing came out. Because no real printing is occurring, the program produces diagnostic output to indicate what is happening. The main program takes care of changing the paper in the printers as needed.

Simulation Project 3

You are to write a program that simulates the behavior of a theater complex. Although the complex is simpler than a typical theater complex, your experiences at movie complexes can help you to visualize the system operation.

In this simulation a theatre complex consists of an arbitrary number of theatres, each with its own movie to be shown and its own schedule. A movie may be shown in more than one theatre if it is popular enough. Customers may buy any number of tickets to any movie playing at the complex. The regular price of a ticket is $7.00, but the price is reduced to $4.00 if the ticket is purchased before 4 p.m.

The theatres are all controlled by a master clock that is overseen by the complex manager. When the clock changes, all theatres are notified so that they can update their state accordingly. The time on the master clock always changes in 15-minute units (standard clocks change their time in one-second units).

Movie schedules include the number of times the movie is shown, the time of its first showing during the day, and the break (in minutes) between consecutive showings. For example, the movie *Grumpy Cat* runs for 45 minutes, shows four times, starts at 7 p.m., and has a break of 15 minutes between showings; therefore, it has the following schedule: 7 p.m., 8 p.m., 9 p.m., and 10 p.m. The length of a movie and the length of the break between consecutive showings is always at least 15 minutes each.

Simulation ■ The number of theatres in the complex, the names of the movies being shown, and schedules for each movie are determined at run time from an input file. The sale of tickets, the passing of time, and management queries are input to the program from a second input file.

After all input from the first data file is read and all objects in the system are created, the program reads the contents of the second data file and executes the corresponding commands.

In this project, no work is actually done to sell tickets or show movies. Instead the process is simulated. In the simulation, tickets are sold, the clock advances time, the schedule and status of all the movies are printed, and a summary for the day is printed.