

## Chapter 12

# Object-Oriented Design

---

### Objectives

- To understand the process of object-oriented design.
- To be able to read and understand object-oriented programs.
- To understand the concepts of encapsulation, polymorphism, and inheritance as they pertain to object-oriented design and programming.
- To be able to design moderately complex software using object-oriented design.

### 12.1 The Process of OOD

Now that you know some data-structuring techniques, it's time to stretch your wings and really put those tools to work. Most modern computer applications are designed using a data-centered view of computing. This so-called object-oriented design (OOD) process is a powerful complement to top-down design for the development of reliable, cost-effective software systems. In this chapter, we will look at the basic principles of OOD and apply them in a couple of case studies.

The essence of design is describing a system in terms of magical black boxes and their interfaces. Each component provides a set of services through its interface. Other components are users or *clients* of the services.

A client only needs to understand the interface of a service; the details of how that service is implemented are not important. In fact, the internal details may change radically and not affect the client at all. Similarly, the component providing the service does not have to consider how the service might be used. The black box just has to make sure that the service is faithfully delivered. This separation of concerns is what makes the design of complex systems possible.

In top-down design, functions serve the role of our magical black boxes. A client program can use a function as long as it understands what the function does. The details of how the task is accomplished are encapsulated in the function definition.

In object-oriented design, the black boxes are objects. The magic behind objects lies in class definitions. Once a suitable class definition has been written, we can completely ignore *how* the class works and just rely on the external interface—the methods. This is what allows you to draw circles in graphics windows without so much as a glance at the code in the `graphics` module. All the nitty-gritty details are encapsulated in the class definitions for `GraphWin` and `Circle`.

If we can break a large problem into a set of cooperating classes, we drastically reduce the complexity that must be considered to understand any given part of the program. Each class stands on its own. Object-oriented design is the process of finding and defining a useful set of classes for a given problem. Like all design, it is part art and part science.

There are many different approaches to OOD, each with its own special techniques, notations, gurus, and textbooks. I can't pretend to teach you all about OOD in one short chapter. On the other hand, I'm not convinced that reading many thick volumes will help much either. The best way to learn about design is to do it. The more you design, the better you will get.

Just to get you started, here are some intuitive guidelines for object-oriented design:

1. **Look for object candidates.** Your goal is to define a set of objects that will be helpful in solving the problem. Start with a careful consideration of the problem statement. Objects are usually described by nouns. You might underline all of the nouns in the problem statement and consider them one by one. Which of them will actually be represented in the program? Which of them have “interesting” behavior? Things that can be represented as primitive data types (numbers or strings) are probably not important candidates for objects. Things that seem to involve a grouping of related data items probably are.

2. **Identify instance variables.** Once you have uncovered some possible objects, think about the information that each object will need to do its job. What kinds of values will the instance variables have? Some object attributes will have primitive values; others might themselves be complex types that suggest other useful objects/classes. Strive to find good “home” classes for all the data in your program.
3. **Think about interfaces.** When you have identified a potential object/class and some associated data, think about what operations would be required for objects of that class to be useful. You might start by considering the verbs in the problem statement. Verbs are used to describe actions—what must be done. List the methods that the class will require. Remember that all manipulation of the object’s data should be done through the methods you provide.
4. **Refine the nontrivial methods.** Some methods will look like they can be accomplished with a couple of lines of code. Other methods will require considerable work to develop an algorithm. Use top-down design and stepwise refinement to flesh out the details of the more difficult methods. As you go along, you may very well discover that some new interactions with other classes are needed, and this might force you to add new methods to other classes. Sometimes you may discover a need for a brand-new kind of object that calls for the definition of another class.
5. **Design iteratively.** As you work through the design, you will bounce back and forth between designing new classes and adding methods to existing classes. Work on whatever seems to be demanding your attention. No one designs a program top to bottom in a linear, systematic fashion. Make progress wherever it seems progress needs to be made.
6. **Try out alternatives.** Don’t be afraid to scrap an approach that doesn’t seem to be working or to follow an idea and see where it leads. Good design involves a lot of trial and error. When you look at the programs of others, you are seeing finished work, not the process they went through to get there. If a program is well designed, it probably is not the result of a first try. Fred Brooks, a legendary software engineer, coined the maxim: “Plan to throw one away.” Often you won’t really know how a system should be built until you’ve already built it the wrong way.
7. **Keep it simple.** At each step in the design, try to find the simplest ap-

proach that will solve the problem at hand. Don't design in extra complexity until it is clear that a more complex approach is needed.

The next sections will walk you through a couple of case studies that illustrate aspects of OOD. Once you thoroughly understand these examples, you will be ready to tackle your own programs and refine your design skills.

## **12.2 Case Study: Racquetball Simulation**

For our first case study, let's return to the racquetball simulation from Chapter 9. You might want to go back and review the program that we developed the first time around using top-down design.

The crux of the problem is to simulate multiple games of racquetball where the ability of the two opponents is represented by the probability that they win a point when they are serving. The inputs to the simulation are the probability for player A, the probability for player B, and the number of games to simulate. The output is a nicely formatted summary of the results.

In the version of the program in Chapter 9, we ended a game when one of the players reached a total of 15 points. This time around, let's also consider shutouts. If one player gets to 7 before the other player has scored a point, the game ends. Our simulation should keep track of both the number of wins for each player and the number of wins that are shutouts.

### **12.2.1 Candidate Objects and Methods**

Our first task is to find a set of objects that could be useful in solving this problem. We need to simulate a series of racquetball games between two players and record some statistics about the series of games. This short description already suggests one way of dividing up the work in the program. We need to do two basic things: simulate a game and keep track of some statistics.

Let's tackle simulation of the game first. We can use an object to represent a single game of racquetball. A game will have to keep track of information about two players. When we create a new game, we will specify the skill levels of the players. This suggests a class—let's call it `RBallGame`—with a constructor that requires parameters for the probabilities of the two players.

What does our program need to do with a game? Obviously, it needs to *play* it. Let's give our class a `play` method that simulates the game until it is over. We could create and play a racquetball game with two lines of code:

```
theGame = RBallGame(probA, probB)
theGame.play()
```

To play lots of games, we just need to put a loop around this code. That's all we really need in `RBallGame` to write the main program. Let's turn our attention to collecting statistics about the games.

Obviously, we will have to keep track of at least four counts in order to print a summary of our simulations: wins for A, wins for B, shutouts for A, and shutouts for B. We will also print out the number of games simulated, but this can be calculated by adding the wins for A and B. Here we have four related pieces of information. Rather than treating them independently, let's group them into a single object. This object will be an instance of a class called `SimStats`.

A `SimStats` object will keep track of all the information about a series of games. We have already analyzed the four crucial pieces of information. Now we have to decide what operations will be useful. For starters, we need a constructor that initializes all of the counts to 0.

We also need a way of updating the counts as each new game is simulated. Let's give our object an update method. The update of the statistics will be based on the outcome of a game. We will have to send some information to the statistics object so that the update can be done appropriately. An easy approach would be to just send the entire game and let update extract whatever information it needs.

Finally, when all of the games have been simulated, we need to print out a report of the results. This suggests a `printReport` method that prints out a nice report of the accumulated statistics.

We have now done enough design that we can actually write the main function for our program. Most of the details have been pushed off into the definition of our two classes.

```
def main():
    printIntro()
    probA, probB, n = getInputs()
    # Play the games
    stats = SimStats()
    for i in range(n):
        theGame = RBallGame(probA, probB) # create a new game
        theGame.play()                    # play it
        stats.update(theGame)              # get info about completed game
    # Print the results
    stats.printReport()
```

I have also used a couple of helper functions to print an introduction and get the inputs. You should have no trouble writing these functions.

Now we have to flesh out the details of our two classes. The `SimStats` class looks pretty easy—let’s tackle that one first.

### **12.2.2**    **Implementing SimStats**

The constructor for `SimStats` just needs to initialize the four counts to 0. Here is an obvious approach:

```
class SimStats:
    def __init__(self):
        self.winsA = 0
        self.winsB = 0
        self.shutsA = 0
        self.shutsB = 0
```

Now let’s take a look at the `update` method. It takes a game as a normal parameter and must update the four counts accordingly. The heading of the method will look like this:

```
def update(self, aGame):
```

But how exactly do we know what to do? We need to know the final score of the game, but this information resides inside `aGame`. Remember, we are not allowed to directly access the instance variables of `aGame`. We don’t even know yet what those instance variables will be.

Our analysis suggests the need for a new method in the `RBallGame` class. We need to extend the interface so that `aGame` has a way of reporting the final score. Let’s call the new method `getScores` and have it return the score for player A and the score for player B.

Now the algorithm for `update` is straightforward:

```
def update(self, aGame):
    a, b = aGame.getScores()
    if a > b:                                # A won the game
        self.winsA = self.winsA + 1
        if b == 0:
            self.shutsA = self.shutsA + 1
    else:                                    # B won the game
```

```

self.winsB = self.winsB + 1
if a == 0:
    self.shutsB = self.shutsB + 1

```

We can complete the `SimStats` class by writing a method to print out the results. Our `printReport` method will generate a table that shows the wins, win percentage, shutouts, and shutout percentage for each player. Here is a sample output:

Summary of 500 games:

	wins (% total)	shutouts (% wins)
Player A:	411 82.2%	60 14.6%
Player B:	89 17.8%	7 7.9%

It is easy to print out the headings for this table, but the formatting of the lines takes a little more care. We want to get the columns lined up nicely, and we must avoid division by zero in calculating the shutout percentage for a player who didn't get any wins. Let's write the basic method, but procrastinate a bit and push off the details of formatting the line into another method, `printLine`. The `printLine` method will need the player label (A or B), number of wins and shutouts, and the total number of games (for calculation of percentages).

```

def printReport(self):
    # Print a nicely formatted report
    n = self.winsA + self.winsB
    print("Summary of", n, "games:\n")
    print("          wins (% total)    shutouts (% wins)  ")
    print("-----")
    self.printLine("A", self.winsA, self.shutsA, n)
    self.printLine("B", self.winsB, self.shutsB, n)

```

To finish out the class, we implement the `printLine` method. This method will make heavy use of string formatting. A good start is to define a template for the information that will appear in each line:

```

def printLine(self, label, wins, shuts, n):
    template = "Player {0}:{1:5}  ({2:5.1%}) {3:11}  ({4})"
    if wins == 0:          # Avoid division by zero!
        shutStr = "-----"

```



```

else:
    shutStr = "{0:4.1%}".format(float(shuts)/wins)
    print(template.format(label, wins, float(wins)/n, shuts, shutStr))

```

Notice how the shutout percentage is handled. The main template includes it as a fifth slot, and the `if` statement takes care of formatting this piece to prevent division by zero.

### 12.2.3 Implementing RBallGame

Now that we have wrapped up the `SimStats` class, we need to turn our attention to `RBallGame`. Summarizing what we have decided so far, this class needs a constructor that accepts two probabilities as parameters, a `play` method that plays the game, and a `getScores` method that reports the scores.

What will a racquetball game need to know? To actually play the game, we have to remember the probability for each player, the score for each player, and which player is serving. If you think about this carefully, you will see that probability and score are properties related to particular *players*, while the server is a property of the *game* between the two players. That suggests that we might simply consider that a game needs to know who the players are and which is serving. The players themselves can be objects that know their probability and score. Thinking about the `RBallGame` class this way leads us to design some new objects.

If the players are objects, then we will need another class to define their behavior. Let's name that class `Player`. A `Player` object will keep track of its probability and current score. When a `Player` is first created the probability will be supplied as a parameter, but the score will just start out at 0. We'll flesh out the design of `Player` class methods as we work on `RBallGame`.

We are now in a position to define the constructor for `RBallGame`. The game will need instance variables for the two players and another variable to keep track of which player is serving:

```

class RBallGame:
    def __init__(self, probA, probB):
        self.playerA = Player(probA)
        self.playerB = Player(probB)
        self.server = self.playerA # Player A always serves first

```

Sometimes it helps to draw a picture of the relationships among the objects that we are creating. Suppose we create an instance of `RBallGame` like this:



```
theGame = RBallGame(.6,.5)
```

Figure 12.1 shows an abstract picture of the objects created by this statement and their interrelationships.

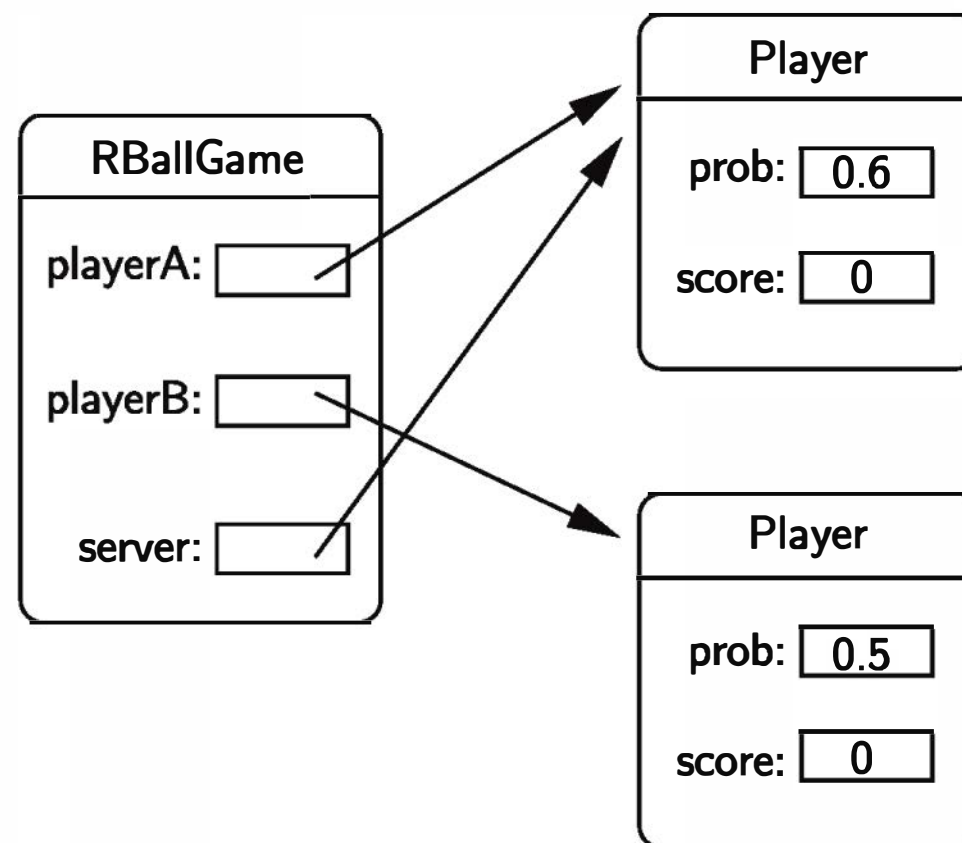


Figure 12.1: Abstract view of RBallGame object

OK, now that we can create an RBallGame, we need to figure out how to play it. Going back to the discussion of racquetball from Chapter 9, we need an algorithm that continues to serve rallies and either award points or change the server as appropriate until the game is over. We can translate this loose algorithm almost directly into our object-based code.

First, we need a loop that continues as long as the game is not over. Obviously, the decision of whether the game has ended can only be made by looking at the game object itself. Let's just assume that an appropriate `isOver` method can be written. The beginning of our `play` method can make use of this (yet-to-be-written) method:

```
def play(self):
    while not self.isOver():
```

Inside the loop, we need to have the serving player serve and, based on the result, decide what to do. This suggests that Player objects should have a method that performs a serve. After all, whether the serve is won or not depends on the probability that is stored inside of each player object. We'll just ask the server if the serve is won or lost:

```
if self.server.winsServe():
```

Based on this result, we either award a point or change the server. To award a point, we need to change a player's score. This again requires the player to do something, namely increment the score. Changing servers, on the other hand, is done at the game level, since this information is kept in the `server` instance variable of `RBallGame`.

Putting it all together, here is our `play` method:

```
def play(self):
    while not self.isOver():
        if self.server.winsServe():
            self.server.incScore()
        else:
            self.changeServer()
```

As long as you remember that `self` is an `RBallGame`, this code should be clear. While the game is not over, if the server wins a serve, award a point to the server; otherwise change the server.

Of course, the price we pay for this simple algorithm is that we now have two new methods (`isOver` and `changeServer`) that need to be implemented in the `RBallGame` class and two more (`winsServe` and `incScore`) for the `Player` class.

Before attacking these new methods, let's go back and finish up the other top-level method of the `RBallGame` class, namely `getScores`. This one just returns the scores of the two players. Of course, we run into the same problem again. It is the player objects that actually know the scores, so we will need a method that asks a player to return its score.

```
def getScores(self):
    return self.playerA.getScore(), self.playerB.getScore()
```

This adds one more method to be implemented in the `Player` class. Make sure you put that on our list to complete later.

To finish out the `RBallGame` class, we need to write the methods `isOver` and `changeServer`. Given what we have developed already and our previous version of this program, these methods are straightforward. I'll leave those as an exercise for you at the moment. If you're looking for my solutions, skip to the complete code at the end of this section.

### 12.2.4 Implementing Player

In developing the `RBallGame` class, we discovered the need for a `Player` class that encapsulates the service probability and current score for a player. The `Player` class needs a suitable constructor and methods for `winsServe`, `incScore`, and `getScore`.

If you are getting the hang of this object-oriented approach, you should have no trouble coming up with a constructor. We just need to initialize the instance variables. The player's probability will be passed as a parameter, and the score starts at 0:

```
def __init__(self, prob):
    # Create a player with this probability
    self.prob = prob
    self.score = 0
```

The other methods for our `Player` class are even simpler. To see whether a player wins a serve, we compare the probability to a random number between 0 and 1:

```
def winsServe(self):
    return random() < self.prob
```

To give a player a point, we simply add one to the score:

```
def incScore(self):
    self.score = self.score + 1
```

The final method just returns the value of the score:

```
def getScore(self):
    return self.score
```

Initially, you may think that it's silly to create a class with a bunch of one- or two-line methods. Actually, it's quite common for a well-modularized, objected-oriented program to have lots of trivial methods. The point of design is to break a problem down into simpler pieces. If those pieces are so simple that their implementations are obvious, that gives us confidence that we must have gotten it right.

### 12.2.5 The Complete Program

That pretty much wraps up our object-oriented version of the racquetball simulation. The complete program follows. You should read through it and make sure you understand exactly what each class does and how it does it. If you have questions about any parts, go back to the discussion above to figure it out.

```
# objrball.py -- Simulation of a racquet game.
#               Illustrates design with objects.

from random import random

class Player:
    # A Player keeps track of service probability and score

    def __init__(self, prob):
        # Create a player with this probability
        self.prob = prob
        self.score = 0

    def winsServe(self):
        # Returns a Boolean that is true with probability self.prob
        return random() < self.prob

    def incScore(self):
        # Add a point to this player's score
        self.score = self.score + 1

    def getScore(self):
        # Returns this player's current score
        return self.score

class RBallGame:
    # A RBallGame represents a game in progress. A game has two players
    # and keeps track of which one is currently serving.

    def __init__(self, probA, probB):
        # Create a new game having players with the given probs.
        self.playerA = Player(probA)
```

---

```
    self.playerB = Player(probB)
    self.server = self.playerA # Player A always serves first

def play(self):
    # Play the game to completion
    while not self.isOver():
        if self.server.winsServe():
            self.server.incScore()
        else:
            self.changeServer()

def isOver(self):
    # Returns game is finished (i.e. one of the players has won).
    a,b = self.getScores()
    return a == 15 or b == 15 or \
        (a == 7 and b == 0) or (b==7 and a == 0)

def changeServer(self):
    # Switch which player is serving
    if self.server == self.playerA:
        self.server = self.playerB
    else:
        self.server = self.playerA

def getScores(self):
    # Returns the current scores of player A and player B
    return self.playerA.getScore(), self.playerB.getScore()

class SimStats:
    # SimStats handles accumulation of statistics across multiple
    # (completed) games. This version tracks the wins and shutouts for
    # each player.

    def __init__(self):
        # Create a new accumulator for a series of games
        self.winsA = 0
        self.winsB = 0
        self.shutsA = 0
```

---

```

        self.shutsB = 0

def update(self, aGame):
    # Determine the outcome of aGame and update statistics
    a, b = aGame.getScores()
    if a > b:                                # A won the game
        self.winsA = self.winsA + 1
        if b == 0:
            self.shutsA = self.shutsA + 1
    else:                                    # B won the game
        self.winsB = self.winsB + 1
        if a == 0:
            self.shutsB = self.shutsB + 1

def printReport(self):
    # Print a nicely formatted report
    n = self.winsA + self.winsB
    print("Summary of", n , "games:\n")
    print("          wins (% total)    shutouts (% wins)  ")
    print("-----")
    self.printLine("A", self.winsA, self.shutsA, n)
    self.printLine("B", self.winsB, self.shutsB, n)

def printLine(self, label, wins, shuts, n):
    template = "Player {0}:{1:5}  ({2:5.1%}) {3:11}    ({4})"
    if wins == 0:                # Avoid division by zero!
        shutStr = "-----"
    else:
        shutStr = "{0:4.1%}".format(float(shuts)/wins)
    print(template.format(label, wins, float(wins)/n, shuts, shutStr))

def printIntro():
    print("This program simulates games of racquetball between two")
    print('players called "A" and "B."  The ability of each player is')
    print("indicated by a probability (a number between 0 and 1) that")
    print("the player wins the point when serving. Player A always")
    print("has the first serve.\n")

```

```

def getInputs():
    # Returns the three simulation parameters
    a = float(input("What is the prob. player A wins a serve? "))
    b = float(input("What is the prob. player B wins a serve? "))
    n = int(input("How many games to simulate? "))
    return a, b, n

def main():
    printIntro()

    probA, probB, n = getInputs()

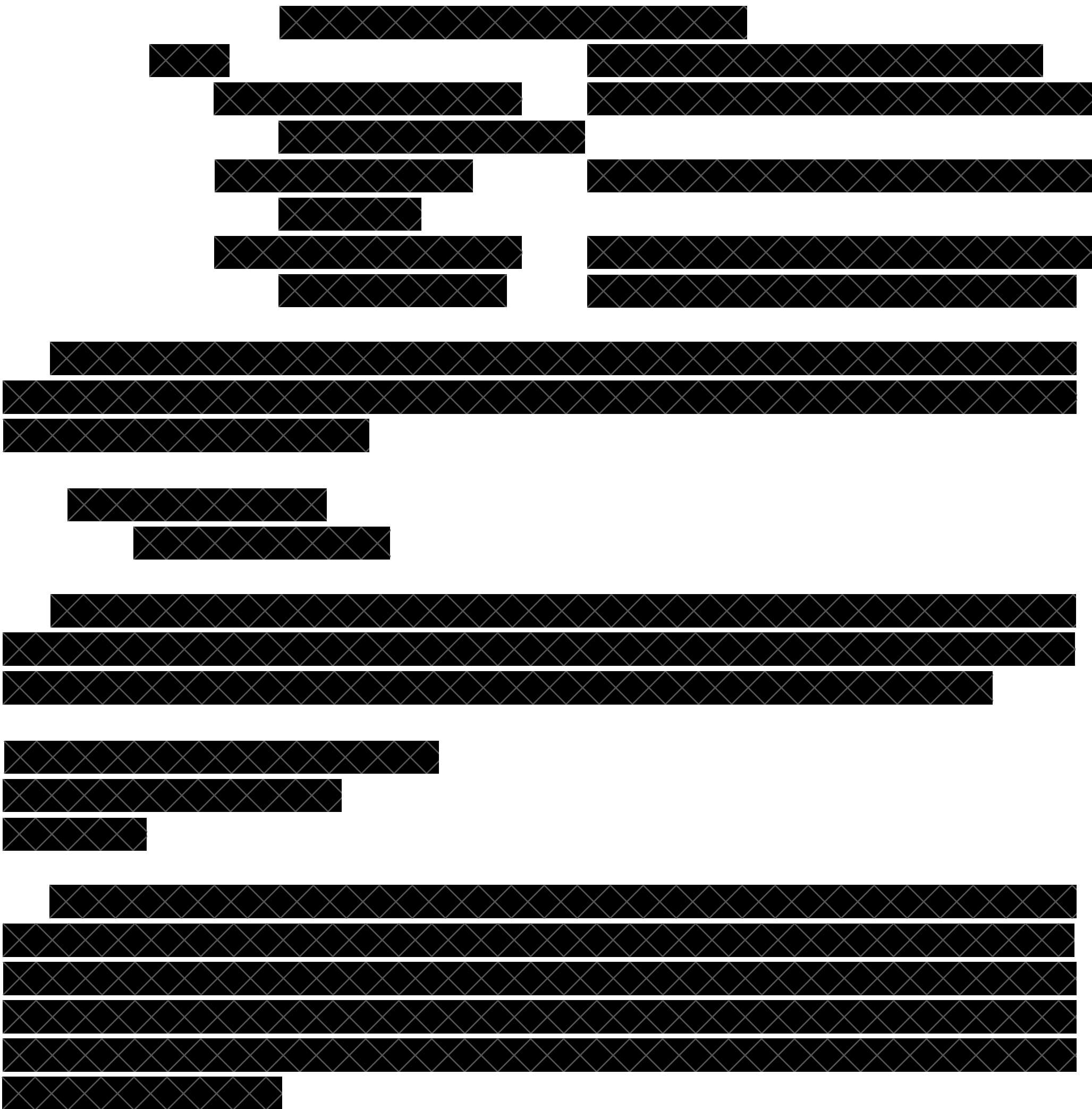
    # Play the games
    stats = SimStats()
    for i in range(n):
        theGame = RBallGame(probA, probB) # create a new game
        theGame.play()                    # play it
        stats.update(theGame)              # extract info

    # Print the results
    stats.printReport()

main()
input("\nPress <Enter> to quit")

```





**12.4** OO Concepts

My goal for the racquetball and video poker case studies was to give you a taste for what OOD is all about. Actually, what you’ve seen is only a distillation of the design process for these two programs. Basically, I have walked you through the algorithms and rationale for two completed designs. I did not document every single decision, false start, and detour along the way. Doing so would have at least tripled the size of this (already long) chapter. You will learn best by

making your own decisions and discovering your own mistakes, not by reading about mine.

Still, these smallish examples illustrate much of the power and allure of the object-oriented approach. Hopefully, you can see why OO techniques have become standard practice in software development. The bottom line is that the OO approach helps us to produce complex software that is more reliable and cost-effective. However, I still have not defined exactly what counts as object-oriented development.

Most OO gurus talk about three features that together make development truly object-oriented: *encapsulation*, *polymorphism*, and *inheritance*. I don't want to belabor these concepts too much, but your introduction to object-oriented design and programming would not be complete without at least some understanding of what is meant by these terms.

#### **12.4.1**   **Encapsulation**

I have already mentioned the term *encapsulation* in previous discussion of objects. As you know, objects know stuff and do stuff. They combine data and operations. This process of packaging some data along with the set of operations that can be performed on the data is called encapsulation.

Encapsulation is one of the major attractions of using objects. It provides a convenient way to compose complex solutions, a way that corresponds to our intuitive view of how the world works. We naturally think of the world around us as consisting of interacting objects. Each object has its own identity, and knowing what kind of object it is allows us to understand its nature and capabilities. I look out my window and I see houses, cars, and trees, not a swarming mass of countless molecules or atoms.

From a design standpoint, encapsulation also provides a critical service of separating the concerns of “what” vs. “how.” The actual implementation of an object is independent of its use. The implementation can change, but as long as the interface is preserved, other components that rely on the object will not break. Encapsulation allows us to isolate major design decisions, especially ones that are subject to change.

Another advantage of encapsulation is that it supports code reuse. It allows us to package up general components that can be used from one program to the next. The `DieView` class and `Button` classes are good examples of reusable components.

Encapsulation is probably the chief benefit of using objects, but alone it only

makes a system *object-based*. To be truly object-oriented, the approach must also have the characteristics of *polymorphism* and *inheritance*.

### 12.4.2 Polymorphism

Literally, the word *polymorphism* means “many forms.” When used in object-oriented literature, this refers to the fact that what an object does in response to a message (a method call) depends on the type or class of the object.

Our poker program illustrated one aspect of polymorphism. The `PokerApp` class was used both with a `TextInterface` and a `GraphicsInterface`. There were two different forms of interface, and the `PokerApp` class could function quite well with either. When the `PokerApp` called the `showDice` method, for example, the `TextInterface` showed the dice one way and the `GraphicsInterface` did it another way.

In our poker example, we used either the text interface or the graphics interface. The remarkable thing about polymorphism, however, is that a given line in a program may invoke a completely different method from one moment to the next. As a simple example, suppose you had a list of graphics objects to draw on the screen. The list might contain a mixture of `Circle`, `Rectangle`, `Polygon`, etc. You could draw all the items in a list with this simple code:

```
for obj in objects:
    obj.draw(win)
```

Now ask yourself, what operation does this loop actually execute? When `obj` is a circle, it executes the `draw` method from the circle class. When `obj` is a rectangle, it is the `draw` method from the rectangle class, etc.

Polymorphism gives object-oriented systems the flexibility for each object to perform an action just the way that it should be performed for that object. Before object orientation, this kind of flexibility was much harder to achieve.

### 12.4.3 Inheritance

The third important property for object-oriented approaches, *inheritance*, is one that we have not yet used. The idea behind inheritance is that a new class can be defined to borrow behavior from another class. The new class (the one doing the borrowing) is called a *subclass*, and the existing class (the one being borrowed from) is its *superclass*.

For example, if we are building a system to keep track of employees, we might have a class `Employee` that contains the general information that is common to all employees. One example attribute would be a `homeAddress` method that returns the home address of an employee. Within the class of all employees, we might distinguish between `SalariedEmployee` and `HourlyEmployee`. We could make these subclasses of `Employee`, so they would share methods like `homeAddress`. However, each subclass would have its own `monthlyPay` function, since pay is computed differently for these different classes of employees.

Inheritance provides two benefits. One is that we can structure the classes of a system to avoid duplication of operations. We don't have to write a separate `homeAddress` method for the `HourlyEmployee` and `SalariedEmployee` classes. A closely related benefit is that new classes can often be based on existing classes, promoting code reuse.

We could have used inheritance to build our poker program. When we first wrote the `DieView` class, it did not provide a way of changing the appearance of the die. We solved this problem by modifying the original class definition. An alternative would have been to leave the original class unchanged and create a new subclass `ColorDieView`. A `ColorDieView` is just like a `DieView` except that it contains an additional method that allows us to change its color. Here is how it would look in Python:

```
class ColorDieView(DieView):  
  
    def setValue(self, value):  
        self.value = value  
        DieView.setValue(self, value)  
  
    def setColor(self, color):  
        self.foreground = color  
        self.setValue(self.value)
```

The first line of this definition says that we are defining a new `ColorDieView` class that is based on (i.e., a subclass of) `DieView`. Inside the new class, we define two methods. The second method, `setColor`, adds the new operation. Of course, in order to make `setColor` work, we also need to modify the `setValue` operation slightly.

The `setValue` method in `ColorDieView` redefines or *overrides* the definition of `setValue` that was provided in the `DieView` class. The `setValue` method in the new class first stores the value and then relies on the `setValue` method

of the superclass `DieView` to actually draw the pips. Notice especially how the call to the method from the superclass is made. The normal approach `self.setValue(value)` would refer to the `setValue` method of the `ColorDieView` class, since `self` is an instance of `ColorDieView`. In order to call the original `setValue` method from the superclass, it is necessary to put the class name where the object would normally go.

```
DieView.setValue(self, value)
```

The actual object to which the method is applied is then sent as the first parameter.

## 12.5 Chapter Summary

This chapter has not introduced very much in the way of new technical content. Rather it has illustrated the process of object-oriented design through the racquetball simulation and dice poker case studies. The key ideas of OOD are summarized here:

- Object-oriented design (OOD) is the process of developing a set of classes to solve a problem. It is similar to top-down design in that the goal is to develop a set of black boxes and associated interfaces. Where top-down design looks for functions, OOD looks for objects.
- There are many different ways to do OOD. The best way to learn is by doing it. Some intuitive guidelines can help:
  1. Look for object candidates.
  2. Identify instance variables.
  3. Think about interfaces.
  4. Refine nontrivial methods.
  5. Design iteratively.
  6. Try out alternatives.
  7. Keep it simple.
- In developing programs with sophisticated user interfaces, it's useful to separate the program into model and view components. One advantage of this approach is that it allows the program to sport multiple looks (e.g., text and GUI interfaces).

- There are three fundamental principles that make software object oriented:

**Encapsulation** Separating the implementation details of an object from how the object is used. This allows for modular design of complex programs.

**Polymorphism** Different classes may implement methods with the same signature. This makes programs more flexible, allowing a single line of code to call different methods in different situations.

**Inheritance** A new class can be derived from an existing class. This supports sharing of methods among classes and code reuse.

## **12.6 Exercises**

### **Review Questions**

#### **True/False**

1. Object-oriented design is the process of finding and defining a useful set of functions for solving a problem.
2. Candidate objects can be found by looking at the verbs in a problem description.
3. Typically, the design process involves considerable trial and error.
4. GUIs are often built with a model-view architecture.
5. Hiding the details of an object in a class definition is called instantiation.
6. Polymorphism literally means “many changes.”
7. A superclass inherits behaviors from its subclasses.
8. GUIs are generally easier to write than text-based interfaces.

#### **Multiple Choice**

1. Which of the following was not a class in the racquetball simulation?  
a) Player   b) SimStats   c) RBallGame   d) Score

2. What is the data type of `server` in an `RBallGame`?  
a) `int`   b) `Player`   c) `bool`   d) `SimStats`
3. The `isOver` method is defined in which class?  
a) `SimStats`   b) `RBallGame`   c) `Player`   d) `PokerApp`
4. Which of the following is not one of the fundamental characteristics of object-oriented design/programming?  
a) inheritance   b) polymorphism  
c) generality   d) encapsulation
5. Separating the user interface from the “guts” of an application is called a(n) \_\_\_\_ approach.  
a) abstract                      b) object-oriented  
c) model-theoretic   d) model-view

### Discussion

1. In your own words, describe the process of OOD.
2. In your own words, define *encapsulation*, *polymorphism*, and *inheritance*.

### Programming Exercises

1. Modify the Dice Poker program from this chapter to include any or all of the following features:
  - a) **Splash Screen.** When the program first fires up, have it print a short introductory message about the program and buttons for “Let’s Play” and “Exit.” The main interface shouldn’t appear unless the user selects “Let’s Play.”
  - b) Add a “Help” button that pops up another window displaying the rules of the game (the payoffs table is the most important part).
  - c) Add a high score feature. The program should keep track of the 10 best scores. When a user quits with a good enough score, he/she is invited to type in a name for the list. The list should be printed in the splash screen when the program first runs. The high-scores list will have to be stored in a file so that it persists between program invocations.



2. Using the ideas from this chapter, implement a simulation of another racquet game. See the programming exercises from Chapter 9 for some ideas.
3. Write a program to keep track of conference attendees. For each attendee, your program should keep track of name, company, state, and email address. Your program should allow users to do things such as add a new attendee, display information on an attendee, delete an attendee, list the names and email addresses of all attendees, and list the names and email addresses of all attendees from a given state. The attendee list should be stored in a file and loaded when the program starts.
4. Write a program that simulates an automatic teller machine (ATM). Since you probably don't have access to a card reader, have the initial screen ask for user ID and a PIN. The user ID will be used to look up the information for the user's accounts (including the PIN to see whether it matches what the user types). Each user will have access to a checking account and a savings account. The user should be able to check balances, withdraw cash, and transfer money between accounts. Design your interface to be similar to what you see on your local ATM. The user account information should be stored in a file when the program terminates. This file is read in again when the program restarts.
5. Find the rules to an interesting dice game and write an interactive program to play it. Some examples are craps, yacht, greed, and skunk.
6. Write a program that deals four bridge hands, counts how many points they have, and gives opening bids. You will probably need to consult a beginner's guide to bridge to help you out.
7. Find a simple card game that you like and implement an interactive program to play that game. Some possibilities are war, blackjack, various solitaire games, and crazy eights.
8. Write an interactive program for a board game. Some examples are Othello (reversi), Connect Four, Battleship, Sorry!, and Parcheesi.
9. (Advanced) Look up a classic video game such as Asteroids, Frogger, Break-out, Tetris, etc. and create your own version using the animation techniques from Chapter 11.