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.

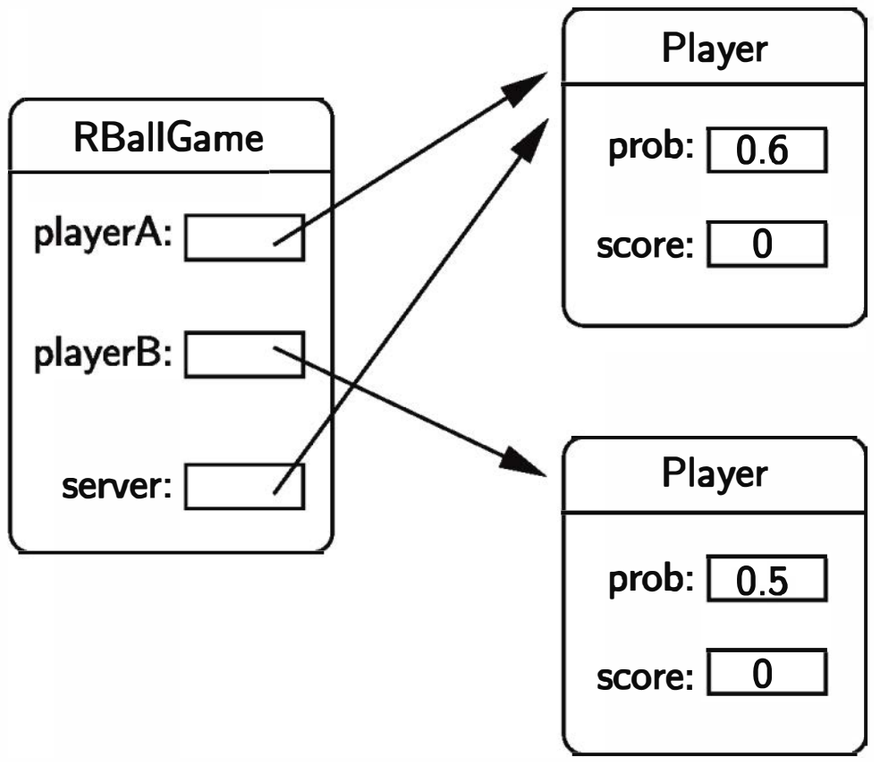

Figure 12.1: Abstract view of `RBallGame` object

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")
```

12.3 Case Study: Dice Poker

Back in Chapter 10, I suggested that objects are particularly useful for the design of graphical user interfaces. Let's finish up this chapter by looking at a graphical application using some of the widgets that we developed in previous chapters.

12.3.1 Program Specification

Our goal is to write a game program that allows a user to play video poker using dice. The program will display a hand consisting of five dice. The basic set of rules is as follows:

- The player starts with $100.
- Each round costs $10 to play. This amount is subtracted from the player's money at the start of the round.
- The player initially rolls a completely random hand (i.e., all five dice are rolled).
- The player gets two chances to enhance the hand by rerolling some or all of the dice.
- At the end of the hand, the player's money is updated according to the following payout schedule:

hand	pay
Two Pairs	$ 5
Three of a Kind	$ 8
Full House (A Pair and a Three of a Kind)	$ 12
Four of a Kind	$ 15
Straight (1–5 or 2–6)	$ 20
Five of a Kind	$ 30

Ultimately, we want this program to present a nice graphical interface. Our interaction will be through mouse clicks. The interface should have the following characteristics:

- The current score (amount of money) is constantly displayed.
- The program automatically terminates if the player goes broke.
- The player may choose to quit at appropriate points during play.
- The interface will present visual cues to indicate what is going on at any given moment and what the valid user responses are.

12.3.2 Identifying Candidate Objects

Our first step is to analyze the program description and identify some objects that will be useful in attacking this problem. This is a game involving dice and money. Are either of these good candidates for objects? Both the money and an individual die can be simply represented as numbers. By themselves, they

do not seem to be good object candidates. However, the game uses five dice, and this sounds like a collection. We will need to be able to roll all the dice or a selection of dice as well as analyze the collection to see what it scores.

We can encapsulate the information about the dice in a `Dice` class. Here are a few obvious operations that this class will have to implement:

constructor Creates the initial collection.

rollAll Assigns random values to each of the five dice.

roll Assigns a random value to some subset of the dice, while maintaining the current value of others.

values Returns the current values of the five dice.

score Returns the score for the dice.

We can also think of the entire program as an object. Let's call the class `PokerApp`. A `PokerApp` object will keep track of the current amount of money, the dice, the number of rolls, etc. It will implement a `run` method that we use to get things started and also some helper methods that are used to implement `run`. We won't know exactly what methods are needed until we design the main algorithm.

Up to this point, I have concentrated on the actual game that we are implementing. Another component to this program will be the user interface. One good way to break down the complexity of a more sophisticated program is to separate the user interface from the main guts of the program. This is often called the *model-view* approach. Our program implements some model (in this case, it models a poker game), and the interface is a view of the current state of the model.

One way of separating out the interface is to encapsulate the decisions about the interface in a separate interface object. An advantage of this approach is that we can change the look and feel of the program simply by substituting a different interface object. For example, we might have a text-based version of a program and a graphical version.

Let's assume that our program will make use of an interface object, call it a `PokerInterface`. It's not clear yet exactly what behaviors we will need from this class, but as we refine the `PokerApp` class, we will need to get information from the user and also display information about the game. These will correspond to methods implemented by the `PokerInterface` class.

12.3.3 Implementing the Model

So far, we have a pretty good picture of what the `Dice` class will do and a starting point for implementing the `PokerApp` class. We could proceed by working on either of these classes. We won't really be able to try out the `PokerApp` class until we have dice, so let's start with the lower-level `Dice` class.

Implementing Dice

The `Dice` class implements a collection of dice, which are just changing numbers. The obvious representation is to use a list of five ints. Our constructor needs to create a list and assign some initial values:

```
class Dice:
    def __init__(self):
        self.dice = [0]*5
        self.rollAll()
```

This code first creates a list of five zeroes. These need to be set to some random values. Since we are going to implement a `rollAll` function anyway, calling it here saves duplicating that code.

We need methods to roll selected dice and also to roll all of the dice. Since the latter is a special case of the former, let's turn our attention to the `roll` function, which rolls a subset. We can specify which dice to roll by passing a list of indexes. For example, `roll([0,3,4])` would roll the dice in positions 0, 3 and 4 of the dice list. We just need a loop that goes through the parameter and generates a new random value for each listed position:

```
    def roll(self, which):
        for pos in which:
            self.dice[pos] = randrange(1,7)
```

Next, we can use `roll` to implement `rollAll` as follows:

```
    def rollAll(self):
        self.roll(range(5))
```

I used `range(5)` to generate a sequence of all the indexes.

The `values` function is used to return the values of the dice so that they can be displayed. Another one-liner suffices:

```
    def values(self):
        return self.dice[:]
```

Notice that I created a copy of the dice list by slicing it. That way, if a `Dice` client modifies the list that it gets back from `values`, it will not affect the original copy stored in the `Dice` object. This defensive programming prevents other parts of the code from accidentally messing with our object.

Finally, we come to the `score` method. This is the function that will determine the worth of the current dice. We need to examine the values and determine whether we have any of the patterns that lead to a payoff, namely five of a kind, four of a kind, full house, three of a kind, two pairs, or straight. Our function will need some way to indicate what the payoff is. Let's return a string labeling what the hand is and an int that gives the payoff amount.

We can think of this function as a multi-way decision. We simply need to check for each possible hand. If we do so in a sensible order, we can guarantee giving the correct payout. For example, a full house also contains a three of a kind. We need to check for the full house before checking for three of a kind, since the full house is more valuable.

One simple way of checking the hand is to generate a list of the counts of each value. That is, `counts[i]` will be the number of times that the value `i` occurs in dice. If the dice are: `[3,2,5,2,3]` then the count list would be `[0,0,2,2,0,1,0]`. Notice that `counts[0]` will always be zero, since dice values are in the range 1–6. Checking for various hands can then be done by looking for various values in `counts`. For example, if `counts` contains a 3 and a 2, the hand contains a triple and a pair; hence, it is a full house.

Here's the code:

```
    def score(self):
        # Create the counts list
        counts = [0] * 7
        for value in self.dice:
            counts[value] = counts[value] + 1

        # score the hand
        if 5 in counts:
            return "Five of a Kind", 30
        elif 4 in counts:
            return "Four of a Kind", 15
        elif (3 in counts) and (2 in counts):
```

```
        return "Full House", 12
    elif 3 in counts:
        return "Three of a Kind", 8
    elif not (2 in counts) and (counts[1]==0 or counts[6] == 0):
        return "Straight", 20
    elif counts.count(2) == 2:
        return "Two Pairs", 5
    else:
        return "Garbage", 0
```

The only tricky part is the testing for straights. Since we have already checked for 5, 4, and 3 of a kind, checking that there are no pairs—`not (2 in counts)`—guarantees that the dice show five distinct values. If there is no 6, then the values must be 1–5; likewise, no 1 means the values must be 2–6.

At this point, we could try out the `Dice` class to make sure that it is working correctly. Here is a short interaction showing some of what the class can do:

```
>>> from dice import Dice
>>> d = Dice()
>>> d.values()
[6, 3, 3, 6, 5]
>>> d.score()
('Two Pairs', 5)
>>> d.roll([4])
>>> d.values()
[6, 3, 3, 6, 4]
>>> d.roll([4])
>>> d.values()
[6, 3, 3, 6, 3]
>>> d.score()
('Full House', 12)
```

We would want to be sure that each kind of hand scores properly.

Implementing PokerApp

Now we are ready to turn our attention to the task of actually implementing the poker game. We can use top-down design to flesh out the details and also suggest what methods will have to be implemented in the `PokerInterface` class.

Initially, we know that the `PokerApp` will need to keep track of the dice, the amount of money, and some user interface. Let's initialize these values in the constructor:

```
class PokerApp:
    def __init__(self):
        self.dice = Dice()
        self.money = 100
        self.interface = PokerInterface()
```

To run the program, we will create an instance of this class and call its `run` method. Basically, the program will loop, allowing the user to continue playing hands until he or she is either out of money or chooses to quit. Since it costs $10 to play a hand, we can continue as long as `self.money >= 10`. Determining whether the user actually wants to play another hand must come from the user interface. Here is one way we might code the `run` method:

```
def run(self):
    while self.money >= 10 and self.interface.wantToPlay():
        self.playRound()
    self.interface.close()
```

Notice the call to `interface.close` at the bottom. This will allow us to do any necessary cleaning up such as printing a final message for the user or closing a graphics window.

Most of the work of the program has now been pushed into the `playRound` method. Let's continue the top-down process by focusing our attention here. Each round will consist of a series of rolls. Based on these rolls, the program will have to adjust the player's score:

```
def playRound(self):
    self.money = self.money - 10
    self.interface.setMoney(self.money)
    self.doRolls()
    result, score = self.dice.score()
    self.interface.showResult(result, score)
    self.money = self.money + score
    self.interface.setMoney(self.money)
```

This code really handles only the scoring aspect of a round. Anytime new information must be shown to the user, a suitable method from `interface` is invoked.

The $10 fee to play a round is first deducted and the interface is updated with the new amount of money remaining. The program then processes a series of rolls (`doRolls`), shows the user the result, and updates the amount of money accordingly.

Finally, we are down to the nitty-gritty details of implementing the dice-rolling process. Initially, all of the dice will be rolled. Then we need a loop that continues rolling user-selected dice until either the user chooses to quit rolling or the limit of three rolls is reached. Let's use a local variable `rolls` to keep track of how many times the dice have been rolled. Obviously, displaying the dice and getting the list of dice to roll must come from interaction with the user through `interface`.

```
def doRolls(self):
    self.dice.rollAll()
    roll = 1
    self.interface.setDice(self.dice.values())
    toRoll = self.interface.chooseDice()
    while roll < 3 and toRoll != []:
        self.dice.roll(toRoll)
        roll = roll + 1
        self.interface.setDice(self.dice.values())
        if roll < 3:
            toRoll = self.interface.chooseDice()
```

At this point, we have completed the basic functions of our interactive poker program. That is, we have a model of the process for playing poker. We can't really test out this program yet, however, because we don't have a user interface.

12.3.4 A Text-Based UI

In designing `PokerApp`, we have also developed a specification for a generic `PokerInterface` class. Our interface must support the methods for displaying information: `setMoney`, `setDice`, and `showResult`. It must also have methods that allow for input from the user: `wantToPlay` and `chooseDice`. These methods can be implemented in many different ways, producing programs that look quite different even though the underlying model, `PokerApp`, remains the same.

Usually, graphical interfaces are much more complicated to design and build than text-based ones. If we are in a hurry to get our application running, we might first try building a simple text-based interface. We can use this for testing

and debugging of the model without all the extra complication of a full-blown GUI.

First, let's tweak our `PokerApp` class a bit so that the user interface is supplied as a parameter to the constructor:

```
class PokerApp:
    def __init__(self, interface):
        self.dice = Dice()
        self.money = 100
        self.interface = interface
```

Then we can easily create versions of the poker program using different interfaces.

Now let's consider a bare-bones interface to test out the poker program. Our text-based version will not present a finished application, but rather, it provides a minimalist interface solely to get the program running. Each of the necessary methods can be given a trivial implementation.

Here is a complete `TextInterface` class using this approach:

```
# textpoker

class TextInterface:

    def __init__(self):
        print("Welcome to video poker.")

    def setMoney(self, amt):
        print("You currently have ${0}.".format(amt))

    def setDice(self, values):
        print("Dice:", values)

    def wantToPlay(self):
        ans = input("Do you wish to try your luck? ")
        return ans[0] in "yY"

    def close(self):
        print("\nThanks for playing!")

    def showResult(self, msg, score):
```

```
            print("{0}. You win ${1}.".format(msg, score))

    def chooseDice(self):
        return eval(input("Enter list of which to change ([] to stop) "))
```

As is usual for test code, I have tried to implement each required method in the simplest possible way. Notice especially the use of `eval` in `chooseDice` as a simple (though potentially unsafe) way of directly inputting the list of indexes for the dice that should be rolled again. Using this interface, we can test out our `PokerApp` program to see whether we have implemented a correct model. Here is a complete program making use of the modules that we have developed:

```
# textpoker.py -- video dice poker using a text-based interface.

from pokerapp import PokerApp
from textpoker import TextInterface

inter = TextInterface()
app = PokerApp(inter)
app.run()
```

Basically, all this program does is create a text-based interface and then build a `PokerApp` using this interface and start it running. Instead of creating a separate module for this, we could also just add the necessary launching code at the end of our `textpoker` module.

When running this program, we get a rough but usable interaction:

```
Welcome to video poker.
Do you wish to try your luck? y
You currently have $90.
Dice: [6, 4, 4, 2, 4]
Enter list of which to change ([] to stop) [0,4]
Dice: [1, 4, 4, 2, 2]
Enter list of which to change ([] to stop) [0]
Dice: [2, 4, 4, 2, 2]
Full House. You win $12.
You currently have $102.
Do you wish to try your luck? y
You currently have $92.
Dice: [5, 6, 4, 4, 5]
```

```
Enter list of which to change ([] to stop) [1]
Dice: [5, 5, 4, 4, 5]
Enter list of which to change ([] to stop) []
Full House. You win $12.
You currently have $104.
Do you wish to try your luck? y
You currently have $94.
Dice: [3, 2, 1, 1, 1]
Enter list of which to change ([] to stop) [0,1]
Dice: [5, 6, 1, 1, 1]
Enter list of which to change ([] to stop) [0,1]
Dice: [1, 5, 1, 1, 1]
Four of a Kind. You win $15.
You currently have $109.
Do you wish to try your luck? n

Thanks for playing!
```

You can see how this interface provides just enough so that we can test out the model. In fact, we've got a game that's already quite a bit of fun to play!

12.3.5 Developing a GUI

Now that we have a working program, let's turn our attention to a graphical interface. Our first step must be to decide exactly how we want our interface to look and function. The interface will have to support the various methods found in the text-based version and will also probably have some additional helper methods.

Designing the Interaction

Let's start with the basic methods that must be supported and decide exactly how interaction with the user will occur. Clearly, in a graphical interface, the faces of the dice and the current score should be continuously displayed. The `setDice` and `setMoney` methods will be used to change those displays. That leaves one output method, `showResult`, that we need to accommodate. One common way to handle this sort of transient information is with a message at the bottom of the window. This is sometimes called a *status bar*.

To get information from the user, we will make use of buttons. In `wantToPlay`, the user will have to decide between either rolling the dice or quitting. We could include "Roll Dice" and "Quit" buttons for this choice. That leaves us with figuring out how the user should choose dice.

To implement `chooseDice`, we could provide a button for each die and have the user click the buttons for the dice they want to roll. When the user is done choosing the dice, they could click the "Roll Dice" button again to roll the selected dice. Elaborating on this idea, it would be nice if we allowed the user to change his or her mind while selecting the dice. Perhaps clicking the button of a currently selected die would cause it to become deselected. The clicking of the button will serve as a sort of toggle that selects/unselects a particular die. The user commits to a certain selection by clicking on "Roll Dice."

Our vision for `chooseDice` suggests a couple of tweaks for the interface. First, we should have some way of showing the user which dice are currently selected. There are lots of ways we could do this. One simple approach would be to change the color of the dice. Let's "gray out" the pips on the dice selected for rolling. Second, we need a good way for the user to indicate that they wish to stop rolling. That is, they would like the dice scored just as they stand. We could handle this by having them click the "Roll Dice" button when no dice are selected, hence asking the program to roll no dice. Another approach would be to provide a separate button to click that causes the dice to be scored. The latter approach seems a bit more intuitive/informative. Let's add a "Score" button to the interface.

Now we have a basic idea of how the interface will function. We still need to figure out how it will look. What is the exact layout of the widgets? Figure 12.2 is a sample of how the interface might look. I'm sure those of you with a more artistic eye can come up with a more pleasing interface, but we'll use this one as our working design.

Managing the Widgets

The graphical interface that we are developing makes use of buttons and dice. Our intent is to reuse the `Button` and `DieView` classes for these widgets that were developed in previous chapters. The `Button` class can be used as is, and since we have quite a number of buttons to manage, we can use a list of `Buttons`, similar to the approach we used in the calculator program from Chapter 11.

Unlike the buttons in the calculator program, the buttons of our poker interface will not be active all of the time. For example, the dice buttons will only

Figure 12.2: GUI interface for video dice poker

be active when the user is actually in the process of choosing dice. When user input is required, the valid buttons for that interaction will be set to active and the others will be inactive. To implement this behavior, we can add a helper method called `choose` to the `PokerInterface` class.

The `choose` method takes a list of button labels as a parameter, activates them, and then waits for the user to click one of them. The return value of the function is the label of the button that was clicked. We can call the `choose` method whenever we need input from the user. For example, if we are waiting for the user to choose either the "Roll Dice" or "Quit" button, we would use a sequence of code like this:

```
choice = self.choose(["Roll Dice", "Quit"])
if choice == "Roll Dice":
    ...
```

Assuming the buttons are stored in an instance variable called `buttons`, here is one possible implementation of `choose`:

```
    def choose(self, choices):
        buttons = self.buttons

        # activate choice buttons, deactivate others
        for b in buttons:
            if b.getLabel() in choices:
                b.activate()
            else:
                b.deactivate()

        # get mouse clicks until an active button is clicked
        while True:
            p = self.win.getMouse()
            for b in buttons:
                if b.clicked(p):
                    return b.getLabel()  # function exit here.
```

The other widgets in our interface will be our `DieView` that we developed in the last two chapters. Basically, we will use the same class as before, but we need to add just a bit of new functionality. As discussed above, we want to change the color of a die to indicate whether it is selected for rerolling.

You might want to go back and review the `DieView` class. Remember, the class constructor draws a square and seven circles to represent the positions where the pips of various values will appear. The `setValue` method turns on the appropriate pips to display a given value. To refresh your memory a bit, here is the `setValue` method as we left it:

```
    def setValue(self, value):
        # Turn all the pips off
        for pip in self.pips:
            pip.setFill(self.background)

        # Turn the appropriate pips back on
        for i in self.onTable[value]:
            self.pips[i].setFill(self.foreground)
```

We need to modify the `DieView` class by adding a `setColor` method. This method will be used to change the color that is used for drawing the pips. As you can see in the code for `setValue`, the color of the pips is determined by the value of the instance variable `foreground`. Of course, changing the value of

`foreground` will not actually change the appearance of the die until it is redrawn using the new color.

The algorithm for `setColor` seems straightforward. We need two steps:

```
change foreground to the new color
redraw the current value of the die
```

Unfortunately, the second step presents a slight snag. We already have code that draws a value, namely `setValue`. But `setValue` requires us to send the value as a parameter, and the current version of `DieView` does not store this value anywhere. Once the proper pips have been turned on, the actual value is discarded.

In order to implement `setColor`, we need to tweak `setValue` so that it remembers the current value. Then `setColor` can redraw the die using its current value. The change to `setValue` is easy; we just need to add a single line:

```
self.value = value
```

This line stores the value parameter in an instance variable called `value`.

With the modified version of `setValue`, implementing `setColor` is a breeze.

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

Notice how the last line simply calls `setValue` to (re)draw the die, passing along the value that was saved from the last time `setValue` was called.

Creating the Interface

Now that we have our widgets under control, we are ready to actually implement our GUI poker interface. The constructor will create all of our widgets, setting up the interface for later interactions:

```
class GraphicsInterface:
    def __init__(self):
        self.win = GraphWin("Dice Poker", 600, 400)
        self.win.setBackground("green3")
        banner = Text(Point(300,30), "Python  Poker  Parlor")
        banner.setSize(24)
        banner.setFill("yellow2")
```

```
        banner.setStyle("bold")
        banner.draw(self.win)
        self.msg = Text(Point(300,380), "Welcome to the Dice Table")
        self.msg.setSize(18)
        self.msg.draw(self.win)
        self.createDice(Point(300,100), 75)
        self.buttons = []
        self.addDiceButtons(Point(300,170), 75, 30)
        b = Button(self.win, Point(300, 230), 400, 40, "Roll Dice")
        self.buttons.append(b)
        b = Button(self.win, Point(300, 280), 150, 40, "Score")
        self.buttons.append(b)
        b = Button(self.win, Point(570,375), 40, 30, "Quit")
        self.buttons.append(b)
        self.money = Text(Point(300,325), "$100")
        self.money.setSize(18)
        self.money.draw(self.win)
```

You should compare this code to Figure 12.2 to make sure you understand how the elements of the interface are created and positioned.

I hope you noticed that I pushed the creation of the dice and their associated buttons into a couple of helper methods. Here are the necessary definitions:

```
    def createDice(self, center, size):
        center.move(-3*size,0)
        self.dice = []
        for i in range(5):
            view = DieView(self.win, center, size)
            self.dice.append(view)
            center.move(1.5*size,0)

    def addDiceButtons(self, center, width, height):
        center.move(-3*width, 0)
        for i in range(1,6):
            label = "Die {0}".format(i)
            b = Button(self.win, center, width, height, label)
            self.buttons.append(b)
            center.move(1.5*width, 0)
```

These two methods are similar in that they employ a loop to draw five similar

widgets. In both cases, a `Point` variable, `center`, is used to calculate the correct position of the next widget.

Implementing the Interaction

You might be a little scared at this point that the constructor for our GUI interface was so complex. Even simple graphical interfaces involve many independent components. Getting them all set up and initialized is often the most tedious part of coding the interface. Now that we have that part out of the way, actually writing the code that handles the interaction will not be too hard, provided we attack it one piece at a time.

Let's start with the simple output methods `setMoney` and `showResult`. These two methods display some text in our interface window. Since our constructor took care of creating and positioning the relevant `Text` objects, all our methods have to do is call the `setText` methods for the appropriate objects:

```
def setMoney(self, amt):
    self.money.setText("${0}".format(amt))

def showResult(self, msg, score):
    if score > 0:
        text = "{0}! You win ${1}".format(msg, score)
    else:
        text = "You rolled {0}".format(msg)
    self.msg.setText(text)
```

In a similar spirit, the output method `setDice` must make a call to the `setValue` method of the appropriate `DieView` objects in `dice`. We can do this with a `for` loop:

```
def setDice(self, values):
    for i in range(5):
        self.dice[i].setValue(values[i])
```

Take a good look at the line in the loop body. It sets the *i*th die to show the *i*th value.

As you can see, once the interface has been constructed, making it functional is not overly difficult. Our output methods are completed with just a few lines of code. The input methods are only slightly more complicated.

The `wantToPlay` method will wait for the user to click either "Roll Dice" or "Quit." We can use our `choose` helper method to do this.

```
def wantToPlay(self):
    ans = self.choose(["Roll Dice", "Quit"])
    self.msg.setText("")
    return ans == "Roll Dice"
```

After waiting for the user to click an appropriate button, this method then clears out any message—such as the previous results—by setting the `msg` text to the empty string. The method then returns a Boolean value by examining the label returned by `choose`.

That brings us to the `chooseDice` method. Here we must implement a more extensive user interaction. The `chooseDice` method returns a list of the indexes of the dice that the user wishes to roll.

In our GUI, the user will choose dice by clicking on corresponding buttons. We need to maintain a list of which dice have been chosen. Each time a die button is clicked, that die is either chosen (its index is appended to the list) or unchosen (its index is removed from the list). In addition, the color of the corresponding `DieView` reflects the status of the die. The interaction ends when the user clicks either the roll button or the score button. If the roll button is clicked, the method returns the list of currently chosen indexes. If the score button is clicked, the function returns an empty list to signal that the player is done rolling.

Here is one way to implement the choosing of dice. The comments in this code explain the algorithm:

```
def chooseDice(self):
    # choices is a list of the indexes of the selected dice
    choices = []                        # No dice chosen yet
    while True:
        # wait for user to click a valid button
        b = self.choose(["Die 1", "Die 2", "Die 3", "Die 4", "Die 5",
                         "Roll Dice", "Score"])

        if b[0] == "D":                 # User clicked a die button
            i = int(b[4]) - 1           # Translate label to die index
            if i in choices:            # Currently selected, unselect it
                choices.remove(i)
                self.dice[i].setColor("black")
            else:                       # Currently deselected, select it
                choices.append(i)
```

```
            self.dice[i].setColor("gray")
        else:                           # User clicked Roll or Score
            for d in self.dice:         # Revert appearance of all dice
                d.setColor("black")
            if b == "Score":            # Score clicked, ignore choices
                return []
            elif choices != []:         # Don't accept Roll unless some
                return choices          #   dice are actually selected
```

That about wraps up our program. The only missing piece of our interface class is the `close` method. To close up the graphical version, we just need to close the graphics window:

```
    def close(self):
        self.win.close()
```

Finally, we need a few lines to actually get our graphical poker-playing program started. This code is exactly like the start code for the textual version, except that we use a `GraphicsInterface` in place of the `TextInterface`:

```
inter = GraphicsInterface()
app = PokerApp(inter)
app.run()
```

We now have a complete, usable video dice poker game. Of course, our game is lacking a lot of bells and whistles such as printing a nice introduction, providing help with the rules, and keeping track of high scores. I have tried to keep this example relatively simple, while still illustrating important issues in the design of GUIs using objects. Improvements are left as exercises for you. Have fun with them!

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 objected-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 objected-*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 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, Breakout, Tetris, etc. and create your own version using the animation techniques from Chapter 11.