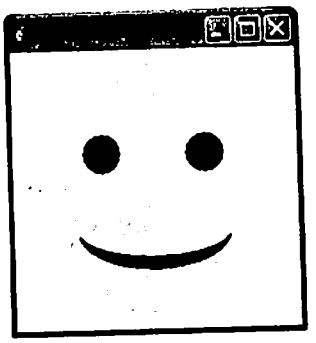

Fig. 6.17 | Creating `JFrame` to display a smiley face. (Part 2 of 2.)

GUI and Graphics Case Study Exercises

6.1 Using method `fillOval`, draw a bull's-eye that alternates between two random colors, as in Fig. 6.18. Use the constructor `Color( int r, int g, int b )` with random arguments to generate random colors.

6.2 Create a program that draws 10 random filled shapes in random colors and positions (Fig. 6.19). `Method paintComponent` should contain a loop that iterates 10 times. In each iteration, the loop should determine whether to draw a filled rectangle or an oval, create a random color and choose coordinates and dimensions at random. The coordinates should be chosen based on the panel's width and height. Lengths of sides should be limited to half the width or height of the window. What happens each time `paintComponent` is called (i.e., the window is resized, uncovered, etc.)? We will resolve this issue in Chapter 8.

6.14 (Optional) Software Engineering Case Study: Identifying Class Operations

In the "Software Engineering Case Study" sections at the ends of Chapters 3, 4 and 5, we performed the first few steps in the object-oriented design of our ATM system. In Chapter 3, we identified the classes that we will need to implement and created our first class diagram. In Chapter 4, we described some attributes of our classes. In Chapter 5, we examined objects' states and modeled objects' state transitions and activities. In this section, we determine some of the class operations (or behaviors) needed to implement the ATM system.

Fig. 6.18 | A bull's-eye with two alternating, random colors.

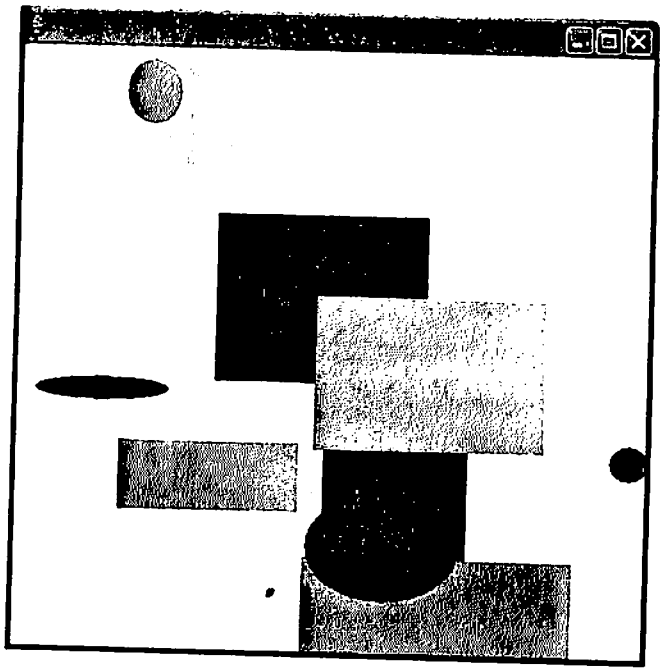

Fig. 6.19 | Randomly generated shapes.

Identifying Operations

An operation is a service that objects of a class provide to clients (users) of the class. Consider the operations of some real-world objects. A radio's operations include setting its station and volume (typically invoked by a person adjusting the radio's controls). A car's operations include accelerating (invoked by the driver pressing the accelerator pedal), decelerating (invoked by the driver pressing the brake pedal or releasing the gas pedal), turning and shifting gears. Software objects can offer operations as well—for example, a software graphics object might offer operations for drawing a circle, drawing a line, drawing a square and the like. A spreadsheet software object might offer operations like printing the spreadsheet, totaling the elements in a row or column and graphing information in the spreadsheet as a bar chart or pie chart.

We can derive many of the operations of each class by examining the key verbs and verb phrases in the requirements document. We then relate each of these to particular classes in our system (Fig. 6.20). The verb phrases in Fig. 6.20 help us determine the operations of each class.

Modeling Operations

To identify operations, we examine the verb phrases listed for each class in Fig. 6.20. The "executes financial transactions" phrase associated with class `ATM` implies that class `ATM` instructs transactions to execute. Therefore, classes `BalanceInquiry`, `Withdrawal` and `Deposit` each need an operation to provide this service to the ATM. We place this operation (which we have named `execute`) in the third compartment of the three transaction classes in the updated class diagram of Fig. 6.21. During an ATM session, the `ATM` object will invoke the `execute` operation of each transaction object to tell it to execute.

`ATM`	executes financial transactions
`BalanceInquiry`	[none in the requirements document]
`Withdrawal`	[none in the requirements document]
`Deposit`	[none in the requirements document]
`BankDatabase`	authenticates a user, retrieves an account balance, credits a deposit amount to an account, debits a withdrawal amount from an account
`Account`	retrieves an account balance, credits a deposit amount to an account, debits a withdrawal amount from an account
`Screen`	displays a message to the user
`Keypad`	receives numeric input from the user
`CashDispenser`	dispenses cash, indicates whether it contains enough cash to satisfy a withdrawal request
`DepositSlot`	receives a deposit envelope

Fig. 6.20 | Verbs and verb phrases for each class in the ATM system.

The UML represents operations (which are implemented as methods in Java) by listing the operation name, followed by a comma-separated list of parameters in parentheses, a colon and the return type:

operationName(*parameter1*, *parameter2*, ..., *parameterN*) : *return type*

Each parameter in the comma-separated parameter list consists of a parameter name, followed by a colon and the parameter type:

parameterName : *parameterType*

For the moment, we do not list the parameters of our operations—we will identify and model the parameters of some of the operations shortly. For some of the operations, we do not yet know the return types, so we also omit them from the diagram. These omissions are perfectly normal at this point. As our design and implementation proceed, we will add the remaining return types.

Figure 6.20 lists the phrase "authenticates a user" next to class `BankDatabase`—the database is the object that contains the account information necessary to determine whether the account number and PIN entered by a user match those of an account held at the bank. Therefore, class `BankDatabase` needs an operation that provides an authentication service to the ATM. We place the operation `authenticateUser` in the third compartment of class `BankDatabase` (Fig. 6.21). However, an object of class `Account`, not class `BankDatabase`, stores the account number and PIN that must be accessed to authenticate a user, so class `Account` must provide a service to validate a PIN obtained through user input against a PIN stored in an `Account` object. Therefore, we add a `validatePIN` operation to class `Account`. Note that we specify a return type of `Boolean` for the `authenticateUser` and `validatePIN` operations. Each operation returns a value indi-

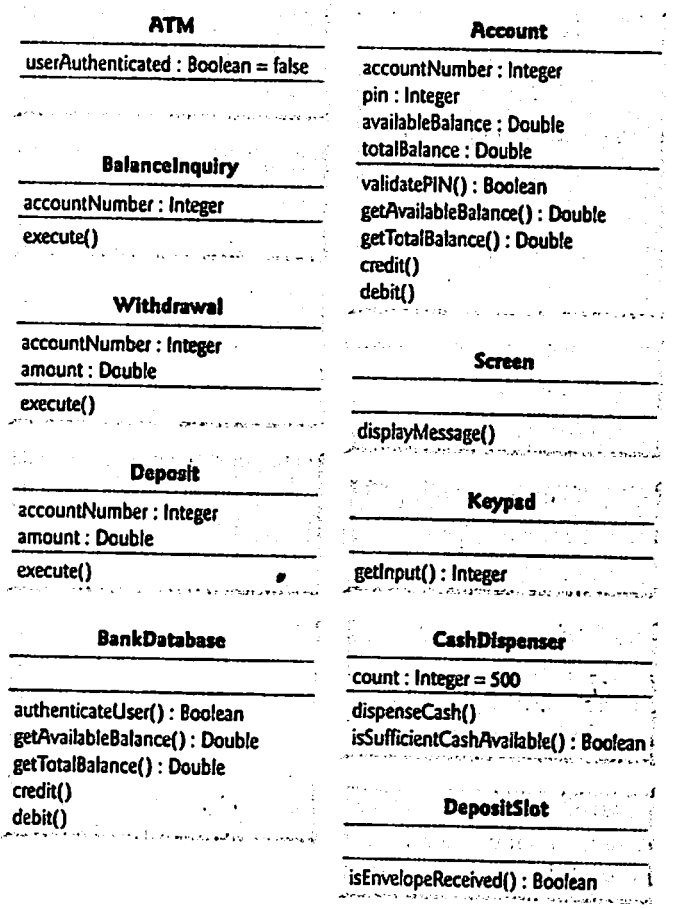

Fig. 6.21 | Classes in the ATM system with attributes and operations.

cating either that the operation was successful in performing its task (i.e., a return value of `true`) or that it was not (i.e., a return value of `false`).

Figure 6.20 lists several additional verb phrases for class `BankDatabase`: "retrieves an account balance," "credits a deposit amount to an account" and "debits a withdrawal amount from an account." Like "authenticates a user," these remaining phrases refer to services that the database must provide to the ATM, because the database holds all the account data used to authenticate a user and perform ATM transactions. However, objects of class `Account` actually perform the operations to which these phrases refer. Thus, we assign an operation to both class `BankDatabase` and class `Account` to correspond to each of these phrases. Recall from Section 3.10 that, because a bank account contains sensitive information, we do not allow the ATM to access accounts directly. The database acts as an intermediary between the ATM and the account data, thus preventing unauthorized access. As we will see in Section 7.14, class `ATM` invokes the operations of class `BankDatabase`, each of which in turn invokes the operation with the same name in class `Account`.

The phrase "retrieves an account balance" suggests that classes BankDatabase and Account each need a getBalance operation. However, recall that we created two attributes in class Account to represent a balance—availableBalance and totalBalance. A balance inquiry requires access to both balance attributes so that it can display them to the user, but a withdrawal needs to check only the value of availableBalance. To allow objects in the system to obtain each balance attribute individually, we add operations getAvailableBalance and getTotalBalance to the third compartment of classes BankDatabase and Account (Fig. 6.21). We specify a return type of Double for these operations because the balance attributes which they retrieve are of type Double.

The phrases "credits a deposit amount to an account" and "debits a withdrawal amount from an account" indicate that classes BankDatabase and Account must perform operations to update an account during a deposit and withdrawal, respectively. We therefore assign credit and debit operations to classes BankDatabase and Account. You may recall that crediting an account (as in a deposit) adds an amount only to the totalBalance attribute. Debiting an account (as in a withdrawal), on the other hand, subtracts the amount from both balance attributes. We hide these implementation details inside class Account. This is a good example of encapsulation and information hiding.

If this were a real ATM system, classes BankDatabase and Account would also provide a set of operations to allow another banking system to update a user's account balance after either confirming or rejecting all or part of a deposit. Operation confirmDepositAmount, for example, would add an amount to the availableBalance attribute, thus making deposited funds available for withdrawal. Operation rejectDepositAmount would subtract an amount from the totalBalance attribute to indicate that a specified amount, which had recently been deposited through the ATM and added to the totalBalance, was not found in the deposit envelope. The bank would invoke this operation after determining either that the user failed to include the correct amount of cash or that any checks did not clear (i.e, they "bounced"). While adding these operations would make our system more complete, we do not include them in our class diagrams or our implementation because they are beyond the scope of the case study.

Class Screen "displays a message to the user" at various times in an ATM session. All visual output occurs through the screen of the ATM. The requirements document describes many types of messages (e.g., a welcome message, an error message, a thank you message) that the screen displays to the user. The requirements document also indicates that the screen displays prompts and menus to the user. However, a prompt is really just a message describing what the user should input next, and a menu is essentially a type of prompt consisting of a series of messages (i.e., menu options) displayed consecutively. Therefore, rather than assign class Screen an individual operation to display each type of message, prompt and menu, we simply create one operation that can display any message specified by a parameter. We place this operation (displayMessage) in the third compartment of class Screen in our class diagram (Fig. 6.21). Note that we do not worry about the parameter of this operation at this time—we model the parameter later in this section.

From the phrase "receives numeric input from the user" listed by class Keypad in Fig. 6.20, we conclude that class Keypad should perform a getInput operation. Because the ATM's keypad, unlike a computer keyboard, contains only the numbers 0–9, we specify that this operation returns an integer value. Recall from the requirements document that in different situations the user may be required to enter a different type of

number (e.g., an account number, a PIN, the number of a menu option, a deposit amount as a number of cents). Class `Keypad` simply obtains a numeric value for a client of the class—it does not determine whether the value meets any specific criteria. Any class that uses this operation must verify that the user entered an appropriate number in a given situation, then respond accordingly (i.e., display an error message via class `Screen`). [*Note:* When we implement the system, we simulate the ATM's keypad with a computer keyboard, and for simplicity we assume that the user does not enter non-numeric input using keys on the computer keyboard that do not appear on the ATM's keypad.]

Figure 6.20 lists "dispenses cash" for class `CashDispenser`. Therefore, we create operation `dispenseCash` and list it under class `CashDispenser` in Fig. 6.21. Class `CashDispenser` also "indicates whether it contains enough cash to satisfy a withdrawal request." Thus, we include `isSufficientCashAvailable`, an operation that returns a value of UML type `Boolean`, in class `CashDispenser`. Figure 6.20 also lists "receives a deposit envelope" for class `DepositSlot`. The deposit slot must indicate whether it received an envelope, so we place an operation `isEnvelopeReceived`, which returns a `Boolean` value, in the third compartment of class `DepositSlot`. [*Note:* A real hardware deposit slot would most likely send the ATM a signal to indicate that an envelope was received. We simulate this behavior, however, with an operation in class `DepositSlot` that class `ATM` can invoke to find out whether the deposit slot received an envelope.]

We do not list any operations for class `ATM` at this time. We are not yet aware of any services that class `ATM` provides to other classes in the system. When we implement the system with Java code, however, operations of this class, and additional operations of the other classes in the system, may emerge.

Identifying and Modeling Operation Parameters

So far, we have not been concerned with the parameters of our operations—we have attempted to gain only a basic understanding of the operations of each class. Let's now take a closer look at some operation parameters. We identify an operation's parameters by examining what data the operation requires to perform its assigned task.

Consider the `authenticateUser` operation of class `BankDatabase`. To authenticate a user, this operation must know the account number and PIN supplied by the user. Thus we specify that operation `authenticateUser` takes integer parameters `userAccountNumber` and `userPIN`, which the operation must compare to the account number and PIN of an `Account` object in the database. We prefix these parameter names with "user" to avoid confusion between the operation's parameter names and the attribute names that belong to class `Account`. We list these parameters in the class diagram in Fig. 6.22 that models only class `BankDatabase`. [*Note:* It is perfectly normal to model only one class in a class diagram. In this case, we are most concerned with examining the parameters of this one class in particular, so we omit the other classes. In class diagrams later in the case study, in which parameters are no longer the focus of our attention, we omit these parameters to save space. Remember, however, that the operations listed in these diagrams still have parameters.]

Recall that the UML models each parameter in an operation's comma-separated parameter list by listing the parameter name, followed by a colon and the parameter type (in UML notation). Figure 6.22 thus specifies that operation `authenticateUser` takes two parameters—`userAccountNumber` and `userPIN`, both of type `Integer`. When we implement the system in Java, we will represent these parameters with `int` values.

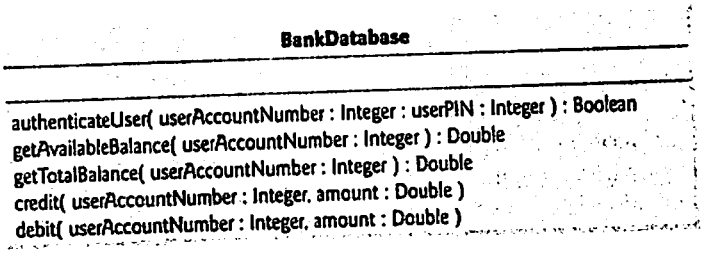

Fig. 6.22 | Class `BankDatabase` with operation parameters.

Class `BankDatabase` operations `getAvailableBalance`, `getTotalBalance`, `credit` and `debit` also each require a `userAccountNumber` parameter to identify the account to which the database must apply the operations, so we include these parameters in the class diagram of Fig. 6.22. In addition, operations `credit` and `debit` each require a `Double` parameter `amount` to specify the amount of money to be credited or debited, respectively.

The class diagram in Fig. 6.23 models the parameters of class `Account`'s operations. Operation `validatePIN` requires only a `userPIN` parameter, which contains the user-specified PIN to be compared with the PIN associated with the account. Like their counterparts in class `BankDatabase`, operations `credit` and `debit` in class `Account` each require a `Double` parameter `amount` that indicates the amount of money involved in the operation. Operations `getAvailableBalance` and `getTotalBalance` in class `Account` require no additional data to perform their tasks. Note that class `Account`'s operations do not require an account number parameter to distinguish between `Accounts`, because these operations can be invoked only on a specific `Account` object.

Figure 6.24 models class `Screen` with a parameter specified for operation `displayMessage`. This operation requires only a `String` parameter `message` that indicates the text to be displayed. Recall that the parameter types listed in our class diagrams are in UML notation, so the `String` type listed in Fig. 6.24 refers to the UML type. When we implement the system in Java, we will in fact use the Java class `String` to represent this parameter.

The class diagram in Fig. 6.25 specifies that operation `dispenseCash` of class `CashDispenser` takes a `Double` parameter `amount` to indicate the amount of cash (in dollars) to be dispensed. Operation `isSufficientCashAvailable` also takes a `Double` parameter `amount` to indicate the amount of cash in question.

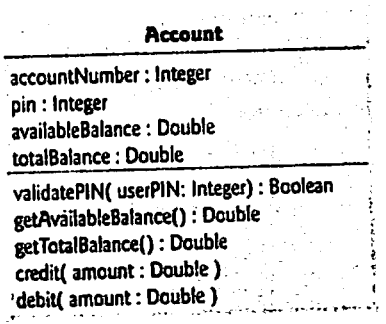

Fig. 6.23 | Class `Account` with operation parameters.

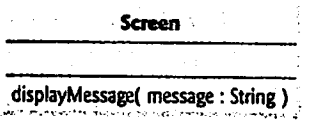

Fig. 6.24 | Class `Screen` with operation parameters.

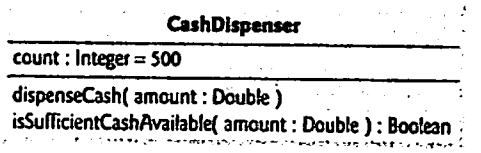

Fig. 6.25 | Class `CashDispenser` with operation parameters.

Note that we do not discuss parameters for operation `execute` of classes `BalanceInquiry`, `Withdrawal` and `Deposit`, operation `getInput` of class `Keypad` and operation `isEnvelopeReceived` of class `DepositSlot`. At this point in our design process, we cannot determine whether these operations require additional data to perform their tasks, so we leave their parameter lists empty. As we progress through the case study, we may decide to add parameters to these operations.

In this section, we have determined many of the operations performed by the classes in the ATM system. We have identified the parameters and return types of some of the operations. As we continue our design process, the number of operations belonging to each class may vary—we might find that new operations are needed or that some current operations are unnecessary—and we might determine that some of our class operations need additional parameters and different return types.

Software Engineering Case Study Self-Review Exercises

6.1 Which of the following is not a behavior?
a) reading data from a file
b) printing output
c) text output
d) obtaining input from the user

6.2 If you were to add to the ATM system an operation that returns the `amount` attribute of class `Withdrawal`, how and where would you specify this operation in the class diagram of Fig. 6.21?

6.3 Describe the meaning of the following operation listing that might appear in a class diagram for an object-oriented design of a calculator:

```
add( x : Integer, y : Integer ) : Integer
```

Answers to Software Engineering Case Study Self-Review Exercises

6.1 c.

6.2 To specify an operation that retrieves the `amount` attribute of class `Withdrawal`, the following operation listing would be placed in the operation (i.e., third) compartment of class `Withdrawal`:

```
getAmount( ) : Double
```

6.3 This operation listing indicates an operation named add that takes integers x and y as parameters and returns an integer value.

6.15 Wrap-Up

In this chapter, you learned more about the details of method declarations. You also learned the difference between non-`static` and `static` methods and how to call `static` methods by preceding the method name with the name of the class in which it appears and a dot (.). You learned how to use operator + to perform string concatenations. You learned how to declare named constants using both `enum` types and `public final static` variables. You saw how to use class `Random` to generate sets of random numbers that can be used for simulations. You also learned about the scope of fields and local variables in a class. Finally, you learned that multiple methods in one class can be overloaded by providing methods with the same name and different signatures. Such methods can be used to perform the same or similar tasks using different types or different numbers of parameters.

In Chapter 7, you will learn how to maintain lists and tables of data in arrays. You will see a more elegant implementation of the application that rolls a die 6000 times and two enhanced versions of our `GradeBook` case study that you studied in Chapters 3–5. You will also learn how to access an application's command-line arguments that are passed to method `main` when an application begins execution.

Summary

- Experience has shown that the best way to develop and maintain a large program is to construct it from small, simple pieces, or modules. This technique is called divide and conquer.
- There are three kinds of modules in Java—methods, classes and packages. Methods are declared within classes. Classes are typically grouped into packages so that they can be imported into programs and reused.
- Methods allow the programmer to modularize a program by separating its tasks into self-contained units. The statements in a method are written only once and hidden from other methods.
- Using existing methods as building blocks to create new programs is a form of software reusability that allows programmers to avoid repeating code within a program.
- A method call specifies the name of the method to call and provides the arguments that the called method requires to perform its task. When the method call completes, the method returns either a result or simply control to its caller.
- A class may contain `static` methods to perform common tasks that do not require an object of the class. Any data a `static` method might require to perform its tasks can be sent to the method as arguments in a method call. A `static` method is called by specifying the name of the class in which the method is declared followed by a dot (.) and the method name, as in

  ClassName.*methodName*(*arguments*)

- Method arguments may be constants, variables or expressions.
- Class `Math` provides `static` methods for performing common mathematical calculations. Class `Math` declares two fields that represent commonly used mathematical constants: `Math.PI` and `Math.E`. The constant `Math.PI` (3.14159265358979323846) is the ratio of a circle's circumference to its diameter. The constant `Math.E` (2.7182818284590452354) is the base value for natural logarithms (calculated with `static` `Math` method `log`).