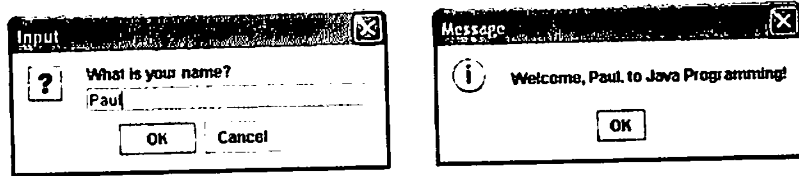

Fig. 3.18 | Obtaining user input from a dialog. (Part 2 of 2.)

Lines 10–11 use method `showInputDialog` of class `JOptionPane` to display a simple input dialog containing a prompt and a field for the user to enter text, known as a text field. The argument to `showInputDialog` is the prompt that indicates what the user should enter. The user types characters in the text field, then clicks the **OK** button or presses the *Enter* key to return the `String` to the program. Method `showInputDialog` returns a `String` containing the characters typed by the user, which we store in variable `name`. [*Note:* If you press the **Cancel** button in the dialog, the method returns `null` and the program displays the word "null" as the name.]

Lines 14–15 use `static String` method `format` to return a `String` containing a greeting with the name entered by the user. Method `format` is similar to method `System.out.printf`, except that `format` returns a formatted `String` rather than displaying it in a command window. Line 18 displays the greeting in a message dialog.

GUI and Graphics Case Study Exercise

3.1 Modify the addition program in Fig. 2.7 to use dialog-based input with `JOptionPane` instead of console-based input using `Scanner`. Since method `showInputDialog` only returns a `String`, you must convert the `String` the user enters to an `int` for use in calculations. Method `Integer.parseInt( String s )` takes a `String` argument representing an integer (e.g., the result of `JOptionPane.showInputDialog`) and returns the value as an `int`. If the `String` does not contain a valid integer, then the program will terminate with an error.

3.10 (Optional) Software Engineering Case Study: Identifying the Classes in a Requirements Document

Now we begin designing the ATM system that we introduced in Chapter 2. In this section, we identify the classes that are needed to build the ATM system by analyzing the nouns and noun phrases that appear in the requirements document. We introduce UML class diagrams to model the relationships between these classes. This is an important first step in defining the structure of our system.

Identifying the Classes in a System

We begin our OOD process by identifying the classes required to build the ATM system. We will eventually describe these classes using UML class diagrams and implement these classes in Java. First, we review the requirements document of Section 2.9 and identify key nouns and noun phrases to help us identify classes that comprise the ATM system. We may decide that some of these nouns and noun phrases are attributes of other classes in the system. We may also conclude that some of the nouns do not correspond to parts of the system and thus should not be modeled at all. Additional classes may become apparent to us as we proceed through the design process.

Figure 3.19 lists the nouns and noun phrases found in the requirements document in Section 2.9. We list them from left to right in the order in which we first encounter them in the requirements document. We list only the singular form of each noun or noun phrase.

We create classes only for the nouns and noun phrases that have significance in the ATM system. We do not need to model "bank" as a class, because the bank is not a part of the ATM system—the bank simply wants us to build the ATM. "Customer" and "user" also represent entities outside of the system—they are important because they interact with our ATM system, but we do not need to model them as classes in the ATM software. Recall that we modeled an ATM user (i.e., a bank customer) as the actor in the use case diagram of Fig. 2.20.

We do not model "$20 bill" or "deposit envelope" as classes. These are physical objects in the real world, but they are not part of what is being automated. We can adequately represent the presence of bills in the system using an attribute of the class that models the cash dispenser. (We assign attributes to classes in Section 4.15.) For example, the cash dispenser maintains a count of the number of bills it contains. The requirements document does not say anything about what the system should do with deposit envelopes after it receives them. We can assume that simply acknowledging the receipt of an envelope—an operation performed by the class that models the deposit slot—is sufficient to represent the presence of an envelope in the system. (We assign operations to classes in Section 6.14.)

In our simplified ATM system, representing various amounts of "money," including the "balance" of an account, as attributes of other classes seems most appropriate. Likewise, the nouns "account number" and "PIN" represent significant pieces of information in the ATM system. They are important attributes of a bank account. They do not, however, exhibit behaviors. Thus, we can most appropriately model them as attributes of an account class.

Though the requirements document frequently describes a "transaction" in a general sense, we do not model the broad notion of a financial transaction at this time. Instead, we model the three types of transactions (i.e., "balance inquiry," "withdrawal" and "deposit") as individual classes. These classes possess specific attributes needed for executing the transactions they represent. For example, a withdrawal needs to know the amount of money the user wants to withdraw. A balance inquiry, however, does not

bank	money / funds	account number
ATM	screen	PIN
user	keypad	bank database
customer	cash dispenser	balance inquiry
transaction	$20 bill / cash	withdrawal
account	deposit slot	deposit
balance	deposit envelope	

Fig. 3.19 | Nouns and noun phrases in the requirements document.

require any additional data. Furthermore, the three transaction classes exhibit unique behaviors. A withdrawal includes dispensing cash to the user, whereas a deposit involves receiving deposit envelopes from the user. [*Note:* In Section 10.9, we "factor out" common features of all transactions into a general "transaction" class using the object-oriented concept of inheritance.]

We determine the classes for our system based on the remaining nouns and noun phrases from Fig. 3.19. Each of these refers to one or more of the following:

- ATM
- screen
- keypad
- cash dispenser
- deposit slot
- account
- bank database
- balance inquiry
- withdrawal
- deposit

The elements of this list are likely to be classes we will need to implement our system.

We can now model the classes in our system based on the list we have created. We capitalize class names in the design process—a UML convention—as we will do when we write the actual Java code that implements our design. If the name of a class contains more than one word, we run the words together and capitalize each word (e.g., `MultipleWordName`). Using this convention, we create classes `ATM`, `Screen`, `Keypad`, `CashDispenser`, `DepositSlot`, `Account`, `BankDatabase`, `BalanceInquiry`, `Withdrawal` and `Deposit`. We construct our system using all of these classes as building blocks. Before we begin building the system, however, we must gain a better understanding of how the classes relate to one another.

Modeling Classes

The UML enables us to model, via class diagrams, the classes in the ATM system and their interrelationships. Figure 3.20 represents class `ATM`. In the UML, each class is modeled as a rectangle with three compartments. The top compartment contains the name of the class centered horizontally in boldface. The middle compartment contains the class's attributes. (We discuss attributes in Section 4.15 and Section 5.11.) The bottom compartment contains the class's operations (discussed in Section 6.14). In Fig. 3.20, the middle and bot-

Fig. 3.20 | Representing a class in the UML using a class diagram.

tom compartments are empty because we have not yet determined this class's attributes and operations.

Class diagrams also show the relationships between the classes of the system. Figure 3.21 shows how our classes `ATM` and `Withdrawal` relate to one another. For the moment, we choose to model only this subset of classes for simplicity. We present a more complete class diagram later in this section. Notice that the rectangles representing classes in this diagram are not subdivided into compartments. The UML allows the suppression of class attributes and operations in this manner to create more readable diagrams, when appropriate. Such a diagram is said to be an elided diagram—one in which some information, such as the contents of the second and third compartments, is not modeled. We will place information in these compartments in Section 4.15 and Section 6.14.

In Fig. 3.21, the solid line that connects the two classes represents an association—a relationship between classes. The numbers near each end of the line are multiplicity values, which indicate how many objects of each class participate in the association. In this case, following the line from one end to the other reveals that, at any given moment, one `ATM` object participates in an association with either zero or one `Withdrawal` objects—zero if the current user is not currently performing a transaction or has requested a different type of transaction, and one if the user has requested a withdrawal. The UML can model many types of multiplicity. Figure 3.22 lists and explains the multiplicity types.

An association can be named. For example, the word `Executes` above the line connecting classes `ATM` and `Withdrawal` in Fig. 3.21 indicates the name of that association. This part of the diagram reads "one object of class `ATM` executes zero or one objects of class `Withdrawal`." Note that association names are directional, as indicated by the filled arrowhead—so it would be improper, for example, to read the preceding association from right to left as "zero or one objects of class `Withdrawal` execute one object of class `ATM`."

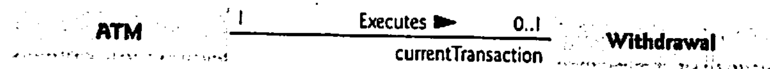

Fig. 3.21 | Class diagram showing an association among classes.

[illegible]	[illegible]
0	None
1	One
m	An integer value
0..1	Zero or one
m, *n*	*m* or *n*
m..*n*	At least *m*, but not more than *n*
*	Any non-negative integer (zero or more)
0..*	Zero or more (identical to *)
1..*	One or more

Fig. 3.22 | Multiplicity types.

The word currentTransaction at the Withdrawal end of the association line in Fig. 3.21 is a role name, which identifies the role the Withdrawal object plays in its relationship with the ATM. A role name adds meaning to an association between classes by identifying the role a class plays in the context of an association. A class can play several roles in the same system. For example, in a school personnel system, a person may play the role of "professor" when relating to students. The same person may take on the role of "colleague" when participating in a relationship with another professor, and "coach" when coaching student athletes. In Fig. 3.21, the role name currentTransaction indicates that the Withdrawal object participating in the Executes association with an object of class ATM represents the transaction currently being processed by the ATM. In other contexts, a Withdrawal object may take on other roles (e.g., the previous transaction). Notice that we do not specify a role name for the ATM end of the Executes association. Role names in class diagrams are often omitted when the meaning of an association is clear without them.

In addition to indicating simple relationships, associations can specify more complex relationships, such as objects of one class being composed of objects of other classes. Consider a real-world automated teller machine. What "pieces" does a manufacturer put together to build a working ATM? Our requirements document tells us that the ATM is composed of a screen, a keypad, a cash dispenser and a deposit slot.

In Fig. 3.23, the solid diamonds attached to the association lines of class ATM indicate that class ATM has a composition relationship with classes Screen, Keypad, CashDispenser and DepositSlot. Composition implies a whole/part relationship. The class that has the composition symbol (the solid diamond) on its end of the association line is the whole (in this case, ATM), and the classes on the other end of the association lines are the parts—in this case, classes Screen, Keypad, CashDispenser and DepositSlot. The compositions in Fig. 3.23 indicate that an object of class ATM is formed from one object of class Screen, one object of class CashDispenser, one object of class Keypad and one object of class DepositSlot. The ATM "has a" screen, a keypad, a cash dispenser and a deposit slot. The "has-a" relationship defines composition. (We will see in the "Software Engineering Case Study" section in Chapter 10 that the "is-a" relationship defines inheritance.)

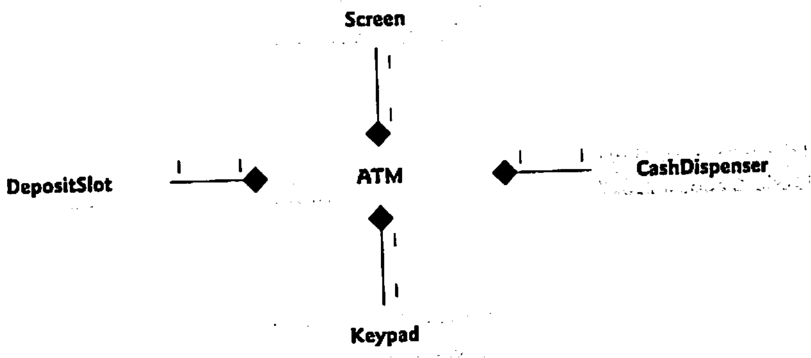

Fig. 3.23 | Class diagram showing composition relationships.

According to the UML specification (www.uml.org), composition relationships have the following properties:

1. Only one class in the relationship can represent the whole (i.e., the diamond can be placed on only one end of the association line). For example, either the screen is part of the ATM or the ATM is part of the screen, but the screen and the ATM cannot both represent the whole in the relationship.
2. The parts in the composition relationship exist only as long as the whole, and the whole is responsible for the creation and destruction of its parts. For example, the act of constructing an ATM includes manufacturing its parts. Furthermore, if the ATM is destroyed, its screen, keypad, cash dispenser and deposit slot are also destroyed.
3. A part may belong to only one whole at a time, although the part may be removed and attached to another whole, which then assumes responsibility for the part.

The solid diamonds in our class diagrams indicate composition relationships that fulfill these three properties. If a "has-a" relationship does not satisfy one or more of these criteria, the UML specifies that hollow diamonds be attached to the ends of association lines to indicate aggregation—a weaker form of composition. For example, a personal computer and a computer monitor participate in an aggregation relationship—the computer "has a" monitor, but the two parts can exist independently, and the same monitor can be attached to multiple computers at once, thus violating the second and third properties of composition.

Figure 3.24 shows a class diagram for the ATM system. This diagram models most of the classes that we identified earlier in this section, as well as the associations between them that we can infer from the requirements document. [*Note:* Classes `BalanceInquiry` and `Deposit` participate in associations similar to those of class `Withdrawal`, so we have chosen to omit them from this diagram to keep the diagram simple. In Chapter 10, we expand our class diagram to include all the classes in the ATM system.]

Figure 3.24 presents a graphical model of the structure of the ATM system. This class diagram includes classes `BankDatabase` and `Account`, and several associations that were not present in either Fig. 3.21 or Fig. 3.23. The class diagram shows that class `ATM` has a one-to-one relationship with class `BankDatabase`—one `ATM` object authenticates users against one `BankDatabase` object. In Fig. 3.24, we also model the fact that the bank's database contains information about many accounts—one object of class `BankDatabase` participates in a composition relationship with zero or more objects of class `Account`. Recall from Fig. 3.22 that the multiplicity value 0..* at the `Account` end of the association between class `BankDatabase` and class `Account` indicates that zero or more objects of class `Account` take part in the association. Class `BankDatabase` has a one-to-many relationship with class `Account`—the `BankDatabase` stores many `Accounts`. Similarly, class `Account` has a many-to-one relationship with class `BankDatabase`—there can be many `Accounts` stored in the `BankDatabase`. [*Note:* Recall from Fig. 3.22 that the multiplicity value * is identical to 0..*. We include 0..* in our class diagrams for clarity.]

Figure 3.24 also indicates that if the user is performing a withdrawal, "one object of class `Withdrawal` accesses/modifies an account balance through one object of class `BankDatabase`." We could have created an association directly between class `Withdrawal` and class `Account`. The requirements document, however, states that the "ATM must interact

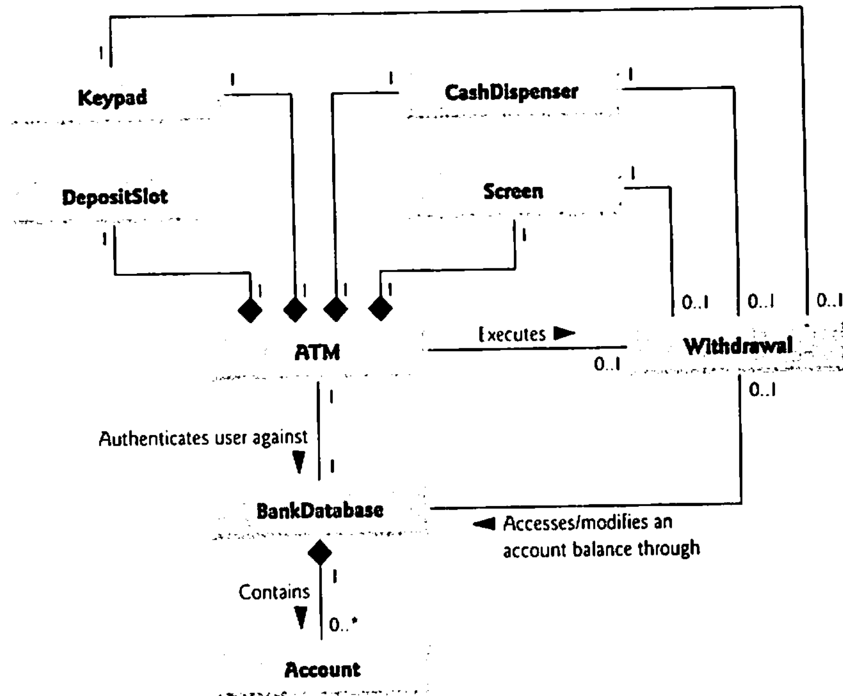

Fig. 3.24 | Class diagram for the ATM system model.

with the bank's account information database" to perform transactions. A bank account contains sensitive information, and systems engineers must always consider the security of personal data when designing a system. Thus, only the `BankDatabase` can access and manipulate an account directly. All other parts of the system must interact with the database to retrieve or update account information (e.g., an account balance).

The class diagram in Fig. 3.24 also models associations between class `Withdrawal` and classes `Screen`, `CashDispenser` and `Keypad`. A withdrawal transaction includes prompting the user to choose a withdrawal amount and receiving numeric input. These actions require the use of the screen and the keypad, respectively. Furthermore, dispensing cash to the user requires access to the cash dispenser.

Classes `BalanceInquiry` and `Deposit`, though not shown in Fig. 3.24, take part in several associations with the other classes of the ATM system. Like class `Withdrawal`, each of these classes associates with classes `ATM` and `BankDatabase`. An object of class `Balance-Inquiry` also associates with an object of class `Screen` to display the balance of an account to the user. Class `Deposit` associates with classes `Screen`, `Keypad` and `DepositSlot`. Like withdrawals, deposit transactions require use of the screen and the keypad to display prompts and receive input, respectively. To receive deposit envelopes, an object of class `Deposit` accesses the deposit slot.

We have now identified the classes in our ATM system (although we may discover others as we proceed with the design and implementation). In Section 4.15, we determine the attributes for each of these classes, and in Section 5.11, we use these attributes to examine how the system changes over time.

Software Engineering Case Study Self-Review Exercises

3.1 Suppose we have a class `Car` that represents a car. Think of some of the different pieces that a manufacturer would put together to produce a whole car. Create a class diagram (similar to Fig. 3.23) that models some of the composition relationships of class `Car`.

3.2 Suppose we have a class `File` that represents an electronic document in a standalone, non-networked computer represented by class `Computer`. What sort of association exists between class `Computer` and class `File`?

a) Class `Computer` has a one-to-one relationship with class `File`.
b) Class `Computer` has a many-to-one relationship with class `File`.
c) Class `Computer` has a one-to-many relationship with class `File`.
d) Class `Computer` has a many-to-many relationship with class `File`.

3.3 State whether the following statement is *true* or *false*, and if *false*, explain why: A UML diagram in which a class's second and third compartments are not modeled is said to be an elided diagram.

3.4 Modify the class diagram of Fig. 3.24 to include class `Deposit` instead of class `Withdrawal`.

Answers to Software Engineering Case Study Self-Review Exercises

3.1 [*Note:* Student answers may vary.] Figure 3.25 presents a class diagram that shows some of the composition relationships of a class `Car`.

3.2 c. [*Note:* In a computer network, this relationship could be many-to-many.]

3.3 True.

3.4 Figure 3.26 presents a class diagram for the ATM including class `Deposit` instead of class `Withdrawal` (as in Fig. 3.24). Note that `Deposit` does not access `CashDispenser`, but does access `DepositSlot`.

3.11 Wrap-Up

In this chapter, you learned the basic concepts of classes, objects, methods and instance variables—these will be used in most Java applications you create. In particular, you learned how to declare instance variables of a class to maintain data for each object of the class, and how to declare methods that operate on that data. You learned how to call a

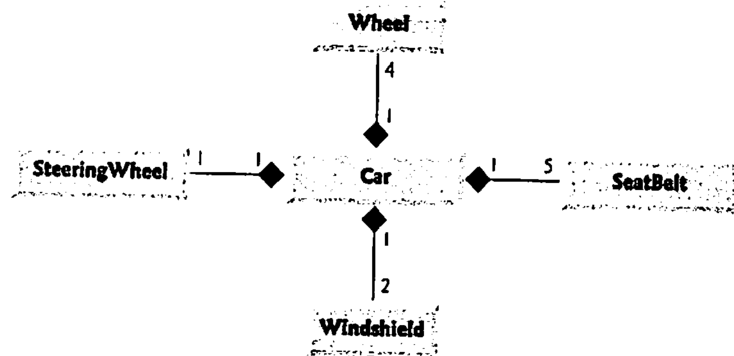

Fig. 3.25 | Class diagram showing composition relationships of a class `Car`.

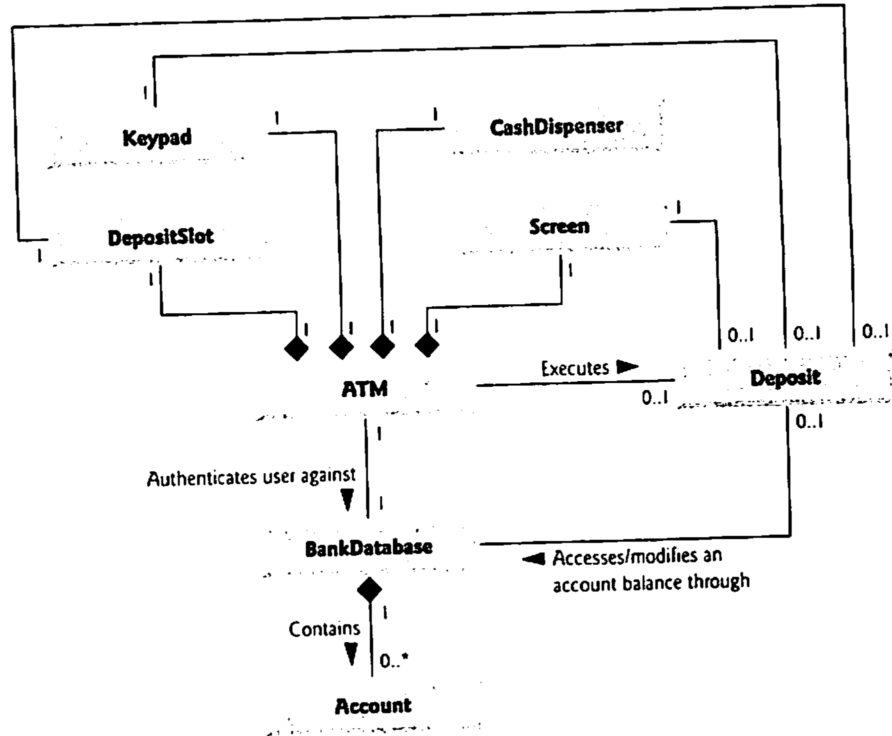

Fig. 3.26 | Class diagram for the ATM system model including class `Deposit`.

method to tell it to perform its task and how to pass information to methods as arguments. You learned the difference between a local variable of a method and an instance variable of a class and that only instance variables are initialized automatically. You also learned how to use a class's constructor to specify the initial values for an object's instance variables. Throughout the chapter, you saw how the UML can be used to create class diagrams that model the constructors, methods and attributes of classes. Finally, you learned about floating-point numbers—how to store them with variables of primitive type `double`, how to input them with a `Scanner` object and how to format them with `printf` and format specifier `%f` for display purposes. In the next chapter we begin our introduction to control statements, which specify the order in which a program's actions are performed. You will use these in your methods to specify how they should perform their tasks.

Summary

- Performing a task in a program requires a method. Inside the method you put the mechanisms that make the method do its tasks—that is, the method hides the implementation details of the tasks that it performs.
- The program unit that houses a method is called a class. A class may contain one or more methods that are designed to perform the class's tasks.
- A method can perform a task and return a result.

Copyright © 1992-2005 by Deitel & Associates, Inc. All Rights Reserved