ATM Case Study, Part 1: Object-Oriented Design with the UML

Action speaks louder than words but not nearly as often.
—Mark Twain

Always design a thing by considering it in its next larger context.
—Elies Saarinen

Oh, life is a glorious cycle of song.
—Dorothy Parker

The Wright brothers’ design … allowed them to survive long enough to learn how to fly.
—Michael Potts

Objectives
In this chapter you’ll learn:

- A simple object-oriented design methodology.
- What a requirements document is.
- To identify classes and class attributes from a requirements document.
- To identify objects’ states, activities and operations from a requirements document.
- To determine the collaborations among objects in a system.
- To work with the UML’s use case, class, state, activity, communication and sequence diagrams to graphically model an object-oriented system.
12.1 Case Study Introduction

Now we begin the optional portion of our object-oriented design and implementation case study. In this chapter and Chapter 13, you’ll design and implement an object-oriented automated teller machine (ATM) software system. The case study provides you with a concise, carefully paced, complete design and implementation experience. In Sections 12.2–12.7 and 13.2–13.3, you’ll perform the steps of an object-oriented design (OOD) process using the UML while relating these steps to the object-oriented concepts discussed in Chapters 2–10. In this chapter, you’ll work with six popular types of UML diagrams to graphically represent the design. In Chapter 13, you’ll tune the design with inheritance, then fully implement the ATM in a 673-line Java application (Section 13.4).

This is not an exercise; rather, it’s an end-to-end learning experience that concludes with a detailed walkthrough of the complete Java code that implements our design. It will begin to acquaint you with the kinds of substantial problems encountered in industry.

These chapters can be studied as a continuous unit after you’ve completed the introduction to object-oriented programming in Chapters 8–11. Or, you can pace the sections one at a time after Chapters 2–8 and 10. Each section of the case study begins with a note telling you the chapter after which it can be covered.

12.2 Examining the Requirements Document

[Note: This section can be taught after Chapter 2.]

We begin our design process by presenting a requirements document that specifies the purpose of the ATM system and what it must do. Throughout the case study, we refer often to this requirements document.

Requirements Document

A local bank intends to install a new automated teller machine (ATM) to allow users (i.e., bank customers) to perform basic financial transactions (Fig. 12.1). Each user can have only one account at the bank. ATM users should be able to view their account balance, withdraw cash (i.e., take money out of an account) and deposit funds (i.e., place money into an account). The user interface of the automated teller machine contains:

- a screen that displays messages to the user
- a keypad that receives numeric input from the user
- a cash dispenser that dispenses cash to the user and
- a deposit slot that receives deposit envelopes from the user.
The cash dispenser begins each day loaded with 500 $20 bills. [Note: Owing to the limited scope of this case study, certain elements of the ATM described here do not accurately mimic those of a real ATM. For example, a real ATM typically contains a device that reads a user’s account number from an ATM card, whereas this ATM asks the user to type the account number on the keypad. A real ATM also usually prints a receipt at the end of a session, but all output from this ATM appears on the screen.]

![Automated teller machine user interface.](image)

*Fig. 12.1 | Automated teller machine user interface.*

The bank wants you to develop software to perform the financial transactions initiated by bank customers through the ATM. The bank will integrate the software with the ATM’s hardware at a later time. The software should encapsulate the functionality of the hardware devices (e.g., cash dispenser, deposit slot) within software components, but it need not concern itself with how these devices perform their duties. The ATM hardware has not been developed yet, so instead of writing your software to run on the ATM, you should develop a first version to run on a personal computer. This version should use the computer’s monitor to simulate the ATM’s screen, and the computer’s keyboard to simulate the ATM’s keypad.

An ATM session consists of authenticating a user (i.e., proving the user’s identity) based on an account number and personal identification number (PIN), followed by creating and executing financial transactions. To authenticate a user and perform transactions, the ATM must interact with the bank’s account information database (i.e., an organized collection of data stored on a computer; we study database access in Chapter 28). For each bank account, the database stores an account number, a PIN and a balance indicating the amount of money in the account. [Note: We assume that the bank plans to build only one ATM, so we need not worry about multiple ATMs accessing this database at the same time. Furthermore, we assume that the bank does not make any changes to the information in the database while a user is accessing the ATM. Also, any
business system like an ATM faces complex and challenging security issues that are beyond the scope of a first or second programming course. We make the simplifying assumption, however, that the bank trusts the ATM to access and manipulate the information in the database without significant security measures.]

Upon first approaching the ATM (assuming no one is currently using it), the user should experience the following sequence of events (shown in Fig. 12.1):

1. The screen displays Welcome! and prompts the user to enter an account number.
2. The user enters a five-digit account number using the keypad.
3. The screen prompts the user to enter the PIN (personal identification number) associated with the specified account number.
4. The user enters a five-digit PIN using the keypad.\(^1\)
5. If the user enters a valid account number and the correct PIN for that account, the screen displays the main menu (Fig. 12.2). If the user enters an invalid account number or an incorrect PIN, the screen displays an appropriate message, then the ATM returns to Step 1 to restart the authentication process.

![Fig. 12.2](image) ATM main menu.

After the ATM authenticates the user, the main menu (Fig. 12.2) should contain a numbered option for each of the three types of transactions: balance inquiry (option 1),

---

\(^1\) In this simple, command-line, text-based ATM, as you type the PIN, it appears on the screen. This is an obvious security breach—you would not want someone looking over your shoulder at an ATM and seeing your PIN displayed on the screen. In Chapter 14, we introduce the JPasswordField GUI component, which displays asterisks as the user types—making it more appropriate for entering PIN numbers and passwords. Exercise 14.18 asks you to build a GUI-based version of the ATM and to use a JPasswordField to obtain the user’s PIN.
withdrawal (option 2) and deposit (option 3). It also should contain an option to allow the user to exit the system (option 4). The user then chooses either to perform a transaction (by entering 1, 2 or 3) or to exit the system (by entering 4).

If the user enters 1 to make a balance inquiry, the screen displays the user’s account balance. To do so, the ATM must retrieve the balance from the bank’s database. The following steps describe what occurs when the user enters 2 to make a withdrawal:

1. The screen displays a menu (Fig. 12.3) containing standard withdrawal amounts: $20 (option 1), $40 (option 2), $60 (option 3), $100 (option 4) and $200 (option 5). The menu also contains an option to allow the user to cancel the transaction (option 6).

![Withdrawal menu](image)

**Fig. 12.3** | ATM withdrawal menu.

2. The user enters a menu selection using the keypad.

3. If the withdrawal amount chosen is greater than the user’s account balance, the screen displays a message stating this and telling the user to choose a smaller amount. The ATM then returns to Step 1. If the withdrawal amount chosen is less than or equal to the user’s account balance (i.e., an acceptable amount), the ATM proceeds to Step 4. If the user chooses to cancel the transaction (option 6), the ATM displays the main menu and waits for user input.

4. If the cash dispenser contains enough cash, the ATM proceeds to Step 5. Otherwise, the screen displays a message indicating the problem and telling the user to choose a smaller withdrawal amount. The ATM then returns to Step 1.

5. The ATM debits the withdrawal amount from the user’s account in the bank’s database (i.e., subtracts the withdrawal amount from the user’s account balance).

6. The cash dispenser dispenses the desired amount of money to the user.
7. The screen displays a message reminding the user to take the money.

The following steps describe the actions that occur when the user enters 3 (when viewing the main menu of Fig. 12.2) to make a deposit:

1. The screen prompts the user to enter a deposit amount or type 0 (zero) to cancel.
2. The user enters a deposit amount or 0 using the keypad. [Note: The keypad does not contain a decimal point or a dollar sign, so the user cannot type a real dollar amount (e.g., $27.25). Instead, the user must enter a deposit amount as a number of cents (e.g., 2725). The ATM then divides this number by 100 to obtain a number representing a dollar amount (e.g., 2725 ÷ 100 = 27.25).]
3. If the user specifies a deposit amount, the ATM proceeds to Step 4. If the user chooses to cancel the transaction (by entering 0), the ATM displays the main menu and waits for user input.
4. The screen displays a message telling the user to insert a deposit envelope.
5. If the deposit slot receives a deposit envelope within two minutes, the ATM credits the deposit amount to the user’s account in the bank’s database (i.e., adds the deposit amount to the user’s account balance). [Note: This money is not immediately available for withdrawal. The bank first must physically verify the amount of cash in the deposit envelope, and any checks in the envelope must clear (i.e., money must be transferred from the check writer’s account to the check recipient’s account). When either of these events occurs, the bank appropriately updates the user’s balance stored in its database. This occurs independently of the ATM system.] If the deposit slot does not receive a deposit envelope within this time period, the screen displays a message that the system has canceled the transaction due to inactivity. The ATM then displays the main menu and waits for user input.

After the system successfully executes a transaction, it should return to the main menu so that the user can perform additional transactions. If the user exits the system, the screen should display a thank you message, then display the welcome message for the next user.

Analyzing the ATM System
The preceding statement is a simplified example of a requirements document. Typically, such a document is the result of a detailed process of requirements gathering, which might include interviews with possible users of the system and specialists in fields related to the system. For example, a systems analyst who is hired to prepare a requirements document for banking software (e.g., the ATM system described here) might interview banking experts to gain a better understanding of what the software must do. The analyst would use the information gained to compile a list of system requirements to guide systems designers as they design the system.

The process of requirements gathering is a key task of the first stage of the software life cycle. The software life cycle specifies the stages through which software goes from the time it’s first conceived to the time it’s retired from use. These stages typically include: analysis, design, implementation, testing and debugging, deployment, maintenance and retirement. Several software life-cycle models exist, each with its own preferences and specifications for when and how often software engineers should perform each of these stages.
Waterfall models perform each stage once in succession, whereas iterative models may repeat one or more stages several times throughout a product’s life cycle.

The analysis stage focuses on defining the problem to be solved. When designing any system, one must solve the problem right, but of equal importance, one must solve the right problem. Systems analysts collect the requirements that indicate the specific problem to solve. Our requirements document describes the requirements of our ATM system in sufficient detail that you need not go through an extensive analysis stage—it’s been done for you.

To capture what a proposed system should do, developers often employ a technique known as use case modeling. This process identifies the use cases of the system, each representing a different capability that the system provides to its clients. For example, ATMs typically have several use cases, such as “View Account Balance,” “Withdraw Cash,” “Deposit Funds,” “Transfer Funds Between Accounts” and “Buy Postage Stamps.” The simplified ATM system we build in this case study allows only the first three.

Each use case describes a typical scenario for which the user uses the system. You’ve already read descriptions of the ATM system’s use cases in the requirements document; the lists of steps required to perform each transaction type (i.e., balance inquiry, withdrawal and deposit) actually described the three use cases of our ATM—“View Account Balance,” “Withdraw Cash” and “Deposit Funds,” respectively.

**Use Case Diagrams**

We now introduce the first of several UML diagrams in the case study. We create a use case diagram to model the interactions between a system’s clients (in this case study, bank customers) and its use cases. The goal is to show the kinds of interactions users have with a system without providing the details—these are provided in other UML diagrams (which we present throughout this case study). Use case diagrams are often accompanied by informal text that gives more detail—like the text that appears in the requirements document. Use case diagrams are produced during the analysis stage of the software life cycle. In larger systems, use case diagrams are indispensable tools that help system designers remain focused on satisfying the users’ needs.

Figure 12.4 shows the use case diagram for our ATM system. The stick figure represents an actor, which defines the roles that an external entity—such as a person or another system—plays when interacting with the system. For our automated teller machine, the actor is a User who can view an account balance, withdraw cash and deposit funds from

![Use case diagram for the ATM system from the User’s perspective.](image-url)
the ATM. The User is not an actual person, but instead comprises the roles that a real person—when playing the part of a User—can play while interacting with the ATM. A use case diagram can include multiple actors. For example, the use case diagram for a real bank’s ATM system might also include an actor named Administrator who refills the cash dispenser each day.

Our requirements document supplies the actors—“ATM users should be able to view their account balance, withdraw cash and deposit funds.” Therefore, the actor in each of the three use cases is the user who interacts with the ATM. An external entity—a real person—plays the part of the user to perform financial transactions. Figure 12.4 shows one actor, whose name, User, appears below the actor in the diagram. The UML models each use case as an oval connected to an actor with a solid line.

Software engineers (more precisely, systems designers) must analyze the requirements document or a set of use cases and design the system before programmers implement it in a particular programming language. During the analysis stage, systems designers focus on understanding the requirements document to produce a high-level specification that describes what the system is supposed to do. The output of the design stage—a design specification—should specify clearly how the system should be constructed to satisfy these requirements. In the next several sections, we perform the steps of a simple object-oriented design (OOD) process on the ATM system to produce a design specification containing a collection of UML diagrams and supporting text.

The UML is designed for use with any OOD process. Many such processes exist, the best known of which is the Rational Unified Process™ (RUP) developed by Rational Software Corporation, now part of IBM. RUP is a rich process intended for designing “industrial strength” applications. For this case study, we present our own simplified design process, designed for students in first and second programming courses.

**Designing the ATM System**

We now begin the design stage of our ATM system. A system is a set of components that interact to solve a problem. For example, to perform the ATM system’s designated tasks, our ATM system has a user interface (Fig. 12.1), and contains software that executes financial transactions and interacts with a database of bank account information. System structure describes the system’s objects and their interrelationships. System behavior describes how the system changes as its objects interact with one another.

Every system has both structure and behavior—designers must specify both. There are several types of system structures and behaviors. For example, the interactions among objects in the system differ from those between the user and the system, yet both constitute a portion of the system behavior.

The UML 2 standard specifies 13 diagram types for documenting the system models. Each models a distinct characteristic of a system’s structure or behavior—six diagrams relate to system structure, the remaining seven to system behavior. We list here only the six diagram types used in our case study—one models system structure; the other five model system behavior. We provide an overview of the remaining seven UML diagram types in Appendix P, UML 2: Additional Diagram Types.

1. Use case diagrams, such as the one in Fig. 12.4, model the interactions between a system and its external entities (actors) in terms of use cases (system capabilities, such as “View Account Balance,” “Withdraw Cash” and “Deposit Funds”).
2. **Class diagrams**, which you’ll study in Section 12.3, model the classes, or “building blocks,” used in a system. Each noun or “thing” described in the requirements document is a candidate to be a class in the system (e.g., Account, Keypad). Class diagrams help us specify the *structural relationships* between parts of the system. For example, the ATM system class diagram will specify that the ATM is physically composed of a screen, a keypad, a cash dispenser and a deposit slot.

3. **State machine diagrams**, which you’ll study in Section 12.5, model the ways in which an object changes state. An object’s *state* is indicated by the values of all its attributes at a given time. When an object changes state, it may behave differently in the system. For example, after validating a user’s PIN, the ATM transitions from the “user not authenticated” state to the “user authenticated” state, at which point it allows the user to perform financial transactions (e.g., view account balance, withdraw cash, deposit funds).

4. **Activity diagrams**, which you’ll also study in Section 12.5, model an object’s *activity*—is workflow (sequence of events) during program execution. An activity diagram models the *actions* the object performs and specifies the *order* in which it performs them. For example, an activity diagram shows that the ATM must obtain the balance of the user’s account (from the bank’s account information database) *before* the screen can display the balance to the user.

5. **Communication diagrams** (called *collaboration diagrams* in earlier versions of the UML) model the interactions among objects in a system, with an emphasis on *what* interactions occur. You’ll learn in Section 12.7 that these diagrams show which objects must interact to perform an ATM transaction. For example, the ATM must communicate with the bank’s account information database to retrieve an account balance.

6. **Sequence diagrams** also model the interactions among the objects in a system, but unlike communication diagrams, they emphasize *when* interactions occur. You’ll learn in Section 12.7 that these diagrams help show the order in which interactions occur in executing a financial transaction. For example, the screen prompts the user to enter a withdrawal amount before cash is dispensed.

In Section 12.3, we continue designing our ATM system by identifying the classes from the requirements document. We accomplish this by extracting key *nouns and noun phrases* from the requirements document. Using these classes, we develop our first draft of the class diagram that models the structure of our ATM system.

**Web Resource**
We’ve created an extensive UML Resource Center that contains many links to additional information, including introductions, tutorials, blogs, books, certification, conferences, developer tools, documentation, e-books, FAQs, forums, groups, UML in Java, podcasts, security, tools, downloads, training courses, videos and more. Browse our UML Resource Center at [www.deitel.com/UML/](http://www.deitel.com/UML/).

**Self-Review Exercises for Section 12.2**

12.1 Suppose we enabled a user of our ATM system to transfer money between two bank accounts. Modify the use case diagram of Fig. 12.4 to reflect this change.
12.2 Model the interactions among objects in a system with an emphasis on when these interactions occur.

a) Class diagrams
b) Sequence diagrams
c) Communication diagrams
d) Activity diagrams

12.3 Which of the following choices lists stages of a typical software life cycle in sequential order?

a) design, analysis, implementation, testing
b) design, analysis, testing, implementation
c) analysis, design, testing, implementation
d) analysis, design, implementation, testing

12.3 Identifying the Classes in a Requirements Document

[Note: This section can be taught after Chapter 3.]

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

Identifying the Classes in a System

We begin our OOD process by identifying the classes required to build the ATM system. We’ll eventually describe these classes using UML class diagrams and implement these classes in Java. First, we review the requirements document of Section 12.2 and identify key nouns and noun phrases to help us identify classes that comprise the ATM system. We may decide that some of these are actually 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 12.5 lists the nouns and noun phrases found in the requirements document. We list them from left to right in the order in which we first encounter them. We list only the singular form of each.

<table>
<thead>
<tr>
<th>Nouns and noun phrases in the ATM requirements document</th>
</tr>
</thead>
<tbody>
<tr>
<td>bank</td>
</tr>
<tr>
<td>screen</td>
</tr>
<tr>
<td>bank database</td>
</tr>
<tr>
<td>transaction</td>
</tr>
<tr>
<td>deposit slot</td>
</tr>
</tbody>
</table>

Fig. 12.5 | Nouns and noun phrases in the ATM requirements document.

We create classes only for the nouns and noun phrases that have significance in the ATM system. We don’t 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 repre-
sent outside entities—they’re 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. 12.4.

We do not model “$20 bill” or “deposit envelope” as classes. These are physical objects in the real world, but they’re 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 the ATM system’s classes in Section 12.4.) 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 the ATM system’s classes in Section 12.6.

In our simplified ATM system, representing various amounts of “money,” including an account’s “balance,” as attributes of classes seems most appropriate. Likewise, the nouns “account number” and “PIN” represent significant pieces of information in the ATM system. They’re 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 the withdrawal. A balance inquiry, however, does not require any additional data other than the account number. 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. In Section 13.3, 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. 12.5. 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 that we’ll need to implement our system.

We can now model the classes in our system based on the list we’ve created. We capitalize class names in the design process—a UML convention—as we’ll 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 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 12.6 represents class `ATM`. Each class is modeled as a rectangle with three compartments. The top one contains the name of the class centered horizontally in boldface. The middle compartment contains the class’s attributes. (We discuss attributes in Sections 12.4–12.5.) The bottom compartment contains the class’s operations (discussed in Section 12.6). In Fig. 12.6, the middle and bottom compartments are empty because we’ve not yet determined this class’s attributes and operations.

**Fig. 12.6** | Representing a class in the UML using a class diagram.

Class diagrams also show the relationships between the classes of the system. Figure 12.7 shows how our classes `ATM` and `Withdrawal` relate to one another. For the moment, for simplicity, we choose to model only this subset of classes. 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’ll place information in these compartments in Sections 12.4–12.6.

**Fig. 12.7** | Class diagram showing an association among classes.

In Fig. 12.7, 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 left to right 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 12.8 lists and explains the multiplicity types.
12.3 Identifying the Classes in a Requirements Document

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>1</td>
<td>One</td>
</tr>
<tr>
<td>(m)</td>
<td>An integer value</td>
</tr>
<tr>
<td>(0..1)</td>
<td>Zero or one</td>
</tr>
<tr>
<td>(m, n)</td>
<td>(m) or (n)</td>
</tr>
<tr>
<td>(m..n)</td>
<td>At least (m), but not more than (n)</td>
</tr>
<tr>
<td>(*)</td>
<td>Any nonnegative integer (zero or more)</td>
</tr>
<tr>
<td>(0.*)</td>
<td>Zero or more (identical to (*))</td>
</tr>
<tr>
<td>(1.*)</td>
<td>One or more</td>
</tr>
</tbody>
</table>

**Fig. 12.8** | Multiplicity types.

An association can be named. For example, the word **Executes** above the line connecting classes **ATM** and **Withdrawal** in Fig. 12.7 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**." 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**."

The word **currentTransaction** at the **Withdrawal** end of the association line in Fig. 12.7 is a **role name**, identifying 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 an association with another professor, and "coach" when coaching student athletes. In Fig. 12.7, 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. 12.9, the **solid diamonds** attached to the **ATM** class's association lines indicate that **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, **Screen**, **Keypad**, **CashDispenser** and **DepositSlot**. The compositions in Fig. 12.9 indicate that an object of class **ATM** is formed from one object of class **Screen**, one object
of class \textit{CashDispenser}, one object of class \textit{Keypad} and one object of class \textit{DepositSlot}. The ATM \textit{has a} screen, a keypad, a cash dispenser and a deposit slot. (As we saw in Chapter 9, the \textit{is-a} relationship defines inheritance. We'll see in Section 13.3 that there's a nice opportunity to use inheritance in the ATM system design.)

![Class diagram showing composition relationships.](image)

**Fig. 12.9** Class diagram showing composition relationships.

According to the UML specification (\url{www.omg.org/technology/documents/formal/uml.htm}), composition relationships have the following properties:

1. Only one class in the relationship can represent the \textit{whole} (i.e., the diamond can be placed on only \textit{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 \textit{parts} in the composition relationship exist only as long as the whole does, 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. Also, if the ATM is destroyed, its screen, keypad, cash dispenser and deposit slot are also destroyed.

3. A \textit{part} may belong to only \textit{one whole} at a time, although it 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 properties. If a \textit{has-a} relationship does not satisfy one or more of these criteria, the UML specifies that \textbf{hollow diamonds} be attached to the ends of association lines to indicate \textit{aggregation}—a weaker form of composition. For example, a personal computer and a computer monitor participate in an aggregation relationship—the computer \textit{has a} monitor, but the two parts can exist independently, and the same monitor can be attached to multiple computers at once, thus violating composition’s second and third properties.

Figure 12.10 shows a class diagram for the ATM system. This diagram models most of the classes that we’ve identified, as well as the associations between them that we can infer from the requirements document. Classes \textit{BalanceInquiry} and \textit{Deposit} participate in associations similar to those of class \textit{Withdrawal}, so we’ve chosen to omit them from this diagram to keep it simple. In Section 13.3, we expand our class diagram to include all the classes in the ATM system.
Figure 12.10 presents a graphical model of ATM system’s structure. It includes classes BankDatabase and Account, and several associations that were not present in either Fig. 12.7 or Fig. 12.9. It shows that class ATM has a one-to-one relationship with class BankDatabase—one ATM object authenticates users against one BankDatabase object. In Fig. 12.10, we also model the fact that the bank’s database contains information about many accounts—one BankDatabase object participates in a composition relationship with zero or more Account objects. 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 can contain many Accounts. Similarly, class Account has a many-to-one relationship with class BankDatabase—there can be many Accounts stored in the BankDatabase. Recall from Fig. 12.8 that the multiplicity value * is identical to 0..*. We include 0..* in our class diagrams for clarity.

Figure 12.10 also indicates that at any given time 0 or 1 Withdrawal objects can exist. 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 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 in-
formation (e.g., an account balance).

The class diagram in Fig. 12.10 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. 12.10, 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 BalanceInquiry 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’ve now identified the initial classes in our ATM system—we may discover others as we proceed with the design and implementation. In Section 12.4 we determine the attributes for each of these classes, and in Section 12.5 we use these attributes to examine how the system changes over time.

Self-Review Exercises for Section 12.3

12.4 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. 12.9) that models some of the composition relationships of class Car.

12.5 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.

12.6 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.

12.7 Modify the class diagram of Fig. 12.10 to include class Deposit instead of class Withdrawal.

12.4 Identifying Class Attributes

[Note: This section can be taught after Chapter 4.]

Classes have attributes (data) and operations (behaviors). Class attributes are implemented as fields, and class operations are implemented as methods. In this section, we determine many of the attributes needed in the ATM system. In Section 12.5 we examine how these attributes represent an object’s state. In Section 12.6 we determine class operations.

Identifying Attributes

Consider the attributes of some real-world objects: A person’s attributes include height, weight and whether the person is left-handed, right-handed or ambidextrous. A radio’s at-
tributes include its station, volume and AM or FM settings. A car’s attributes include its speedometer and odometer readings, the amount of gas in its tank and what gear it’s in. A personal computer’s attributes include its manufacturer (e.g., Dell, Sun, Apple or IBM), type of screen (e.g., LCD or CRT), main memory size and hard disk size.

We can identify many attributes of the classes in our system by looking for descriptive words and phrases in the requirements document. For each such word and phrase we find that plays a significant role in the ATM system, we create an attribute and assign it to one or more of the classes identified in Section 12.3. We also create attributes to represent any additional data that a class may need, as such needs become clear throughout the design process.

Figure 12.11 lists the words or phrases from the requirements document that describe each class. We formed this list by reading the requirements document and identifying any words or phrases that refer to characteristics of the classes in the system. For example, the requirements document describes the steps taken to obtain a “withdrawal amount,” so we list “amount” next to class Withdrawal.

<table>
<thead>
<tr>
<th>Class</th>
<th>Descriptive words and phrases</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATM</td>
<td>user is authenticated</td>
</tr>
<tr>
<td>BalanceInquiry</td>
<td>account number</td>
</tr>
<tr>
<td>Withdrawal</td>
<td>account number amount</td>
</tr>
<tr>
<td>Deposit</td>
<td>account number amount</td>
</tr>
<tr>
<td>BankDatabase</td>
<td>[no descriptive words or phrases]</td>
</tr>
<tr>
<td>Account</td>
<td>account number PIN</td>
</tr>
<tr>
<td>Screen</td>
<td>balance</td>
</tr>
<tr>
<td>Keypad</td>
<td>[no descriptive words or phrases]</td>
</tr>
<tr>
<td>CashDispenser</td>
<td>begins each day loaded with 500 $20 bills</td>
</tr>
<tr>
<td>DepositSlot</td>
<td>[no descriptive words or phrases]</td>
</tr>
</tbody>
</table>

**Fig. 12.11** Descriptive words and phrases from the ATM requirements document.

Figure 12.11 leads us to create one attribute of class ATM. Class ATM maintains information about the state of the ATM. The phrase “user is authenticated” describes a state of the ATM (we introduce states in Section 12.5), so we include userAuthenticated as a **Boolean attribute** (i.e., an attribute that has a value of either true or false) in class ATM. The Boolean attribute type in the UML is equivalent to the boolean type in Java. This attribute indicates whether the ATM has successfully authenticated the current user—userAuthenticated must be true for the system to allow the user to perform transactions and access account information. This attribute helps ensure the security of the data in the system.

Classes BalanceInquiry, Withdrawal and Deposit share one attribute. Each transaction involves an “account number” that corresponds to the account of the user making the
transaction. We assign an integer attribute `accountNumber` to each transaction class to identify the account to which an object of the class applies.

Descriptive words and phrases in the requirements document also suggest some differences in the attributes required by each transaction class. The requirements document indicates that to withdraw cash or deposit funds, users must input a specific “amount” of money to be withdrawn or deposited, respectively. Thus, we assign to classes `Withdrawal` and `Deposit` an attribute amount to store the value supplied by the user. The amounts of money related to a withdrawal and a deposit are defining characteristics of these transactions that the system requires for these transactions to take place. Class `BalanceInquiry`, however, needs no additional data to perform its task—it requires only an account number to indicate the account whose balance should be retrieved.

Class `Account` has several attributes. The requirements document states that each bank account has an “account number” and “PIN,” which the system uses for identifying accounts and authenticating users. We assign to class `Account` two integer attributes: `accountNumber` and `pin`. The requirements document also specifies that an account maintains a “balance” of the amount of money in the account and that money the user deposits does not become available for a withdrawal until the bank verifies the amount of cash in the deposit envelope, and any checks in the envelope clear. An account must still record the amount of money that a user deposits, however. Therefore, we decide that an account should represent a balance using two attributes: `availableBalance` and `totalBalance`. Attribute `availableBalance` tracks the amount of money that a user can withdraw from the account. Attribute `totalBalance` refers to the total amount of money that the user has “on deposit” (i.e., the amount of money available, plus the amount waiting to be verified or cleared). For example, suppose an ATM user deposits $50.00 into an empty account. The `totalBalance` attribute would increase to $50.00 to record the deposit, but the `availableBalance` would remain at $0. [Note: We assume that the bank updates the `availableBalance` attribute of an `Account` some length of time after the ATM transaction occurs, in response to confirming that $50 worth of cash or checks was found in the deposit envelope. We assume that this update occurs through a transaction that a bank employee performs using some piece of bank software other than the ATM. Thus, we do not discuss this transaction in our case study.]

Class `CashDispenser` has one attribute. The requirements document states that the cash dispenser “begins each day loaded with 500 $20 bills.” The cash dispenser must keep track of the number of bills it contains to determine whether enough cash is on hand to satisfy withdrawal requests. We assign to class `CashDispenser` an integer attribute `count`, which is initially set to 500.

For real problems in industry, there’s no guarantee that requirements documents will be precise enough for the object-oriented systems designer to determine all the attributes or even all the classes. The need for additional classes, attributes and behaviors may become clear as the design process proceeds. As we progress through this case study, we will continue to add, modify and delete information about the classes in our system.

**Modeling Attributes**

The class diagram in Fig. 12.12 lists some of the attributes for the classes in our system—the descriptive words and phrases in Fig. 12.11 lead us to identify these attributes. For simplicity, Fig. 12.12 does not show the associations among classes—we showed these in Fig. 12.10. This is a common practice of systems designers when designs are being devel-
oped. Recall from Section 12.3 that in the UML, a class’s attributes are placed in the middle compartment of the class’s rectangle. We list each attribute’s name and type separated by a colon (:), followed in some cases by an equal sign (=) and an initial value.

Consider the `userAuthenticated` attribute of class `ATM`:

```plaintext
userAuthenticated : Boolean = false
```

This attribute declaration contains three pieces of information about the attribute. The attribute name is `userAuthenticated`. The attribute type is `Boolean`. In Java, an attribute can be represented by a primitive type, such as `boolean`, `int` or `double`, or a reference type like a class. We’ve chosen to model only primitive-type attributes in Fig. 12.12—we discuss the reasoning behind this decision shortly. The attribute types in Fig. 12.12 are in UML notation. We’ll associate the types `Boolean`, `Integer` and `Double` in the UML diagram with the primitive types `boolean`, `int` and `double` in Java, respectively.

![Diagram of ATM and related classes](image)

**Fig. 12.12** | Classes with attributes.

We can also indicate an initial value for an attribute. The `userAuthenticated` attribute in class `ATM` has an initial value of `false`. This indicates that the system initially does not consider the user to be authenticated. If an attribute has no initial value specified, only its name and type (separated by a colon) are shown. For example, the `accountNumber` attribute of class `BalanceInquiry` is an integer. Here we show no initial value, because the
value of this attribute is a number that we do not yet know. This number will be determined at execution time based on the account number entered by the current ATM user.

Figure 12.12 does not include attributes for classes Screen, Keypad and DepositSlot. These are important components of our system, for which our design process has not yet revealed any attributes. We may discover some, however, in the remaining phases of design or when we implement these classes in Java. This is perfectly normal.

Software Engineering Observation 12.1

At early stages in the design process, classes often lack attributes (and operations). Such classes should not be eliminated, however, because attributes (and operations) may become evident in the later phases of design and implementation.

Figure 12.12 also does not include attributes for class BankDatabase. Recall that attributes in Java can be represented by either primitive types or reference types. We’ve chosen to include only primitive-type attributes in the class diagram in Fig. 12.12 (and in similar class diagrams throughout the case study). A reference-type attribute is modeled more clearly as an association between the class holding the reference and the class of the object to which the reference points. For example, the class diagram in Fig. 12.10 indicates that class BankDatabase participates in a composition relationship with zero or more Account objects. From this composition, we can determine that when we implement the ATM system in Java, we’ll be required to create an attribute of class BankDatabase to hold references to zero or more Account objects. Similarly, we can determine reference-type attributes of class ATM that correspond to its composition relationships with classes Screen, Keypad, CashDispenser and DepositSlot. These composition-based attributes would be redundant if modeled in Fig. 12.12, because the compositions modeled in Fig. 12.10 already convey the fact that the database contains information about zero or more accounts and that an ATM is composed of a screen, keypad, cash dispenser and deposit slot. Software developers typically model these whole/part relationships as compositions rather than as attributes required to implement the relationships.

The class diagram in Fig. 12.12 provides a solid basis for the structure of our model, but the diagram is not complete. In Section 12.5 we identify the states and activities of the objects in the model, and in Section 12.6 we identify the operations that the objects perform. As we present more of the UML and object-oriented design, we’ll continue to strengthen the structure of our model.

Self-Review Exercises for Section 12.4

12.8 We typically identify the attributes of the classes in our system by analyzing the _______ in the requirements document.
   a) nouns and noun phrases
   b) descriptive words and phrases
   c) verbs and verb phrases
   d) All of the above.

12.9 Which of the following is not an attribute of an airplane?
   a) length
   b) wingspan
   c) fly
   d) number of seats
12.10 Describe the meaning of the following attribute declaration of class CashDispenser in the class diagram in Fig. 12.12:

    count : Integer = 500

12.5 Identifying Objects’ States and Activities

[Note: This section can be taught after Chapter 5.]

In Section 12.4, we identified many of the class attributes needed to implement the ATM system and added them to the class diagram in Fig. 12.12. We now show how these attributes represent an object's state. We identify some key states that our objects may occupy and discuss how objects change state in response to various events occurring in the system. We also discuss the workflow, or activities, that objects perform in the ATM system, and we present the activities of BalanceInquiry and Withdrawal transaction objects.

State Machine Diagrams

Each object in a system goes through a series of states. An object’s state is indicated by the values of its attributes at a given time. State machine diagrams (commonly called state diagrams) model several states of an object and show under what circumstances the object changes state. Unlike the class diagrams presented in earlier case study sections, which focused primarily on the system’s structure, state diagrams model some of the system’s behavior.

Figure 12.13 is a simple state diagram that models some of the states of an object of class ATM. The UML represents each state in a state diagram as a rounded rectangle with the name of the state placed inside it. A solid circle with an attached stick (→) arrowhead designates the initial state. Recall that we modeled this state information as the Boolean attribute userAuthenticated in the class diagram of Fig. 12.12. This attribute is initialized to false, or the “User not authenticated” state, according to the state diagram.

![State Diagram](image)

**Fig. 12.13** | State diagram for the ATM object.

The arrows with stick (→) arrowhead indicate transitions between states. An object can transition from one state to another in response to various events that occur in the system. The name or description of the event that causes a transition is written near the line that corresponds to the transition. For example, the ATM object changes from the “User not authenticated” to the “User authenticated” state after the database authenticates the user. Recall from the requirements document that the database authenticates a user by comparing the account number and PIN entered by the user with those of an account in the database. If the user has entered a valid account number and the correct PIN, the ATM object transitions to the “User authenticated” state and changes its userAuthenticated attribute to a value of true. When the user exits the system by choosing the “exit” option from the main menu, the ATM object returns to the “User not authenticated” state.
Software Engineering Observation 12.2
Software designers do not generally create state diagrams showing every possible state and state transition for all attributes—there are simply too many of them. State diagrams typically show only key states and state transitions.

Activity Diagrams
Like a state diagram, an activity diagram models aspects of system behavior. Unlike a state diagram, an activity diagram models an object’s workflow (sequence of events) during program execution. An activity diagram models the actions the object will perform and in what order. The activity diagram in Fig. 12.14 models the actions involved in executing a balance-inquiry transaction. We assume that a BalanceInquiry object has already been initialized and assigned a valid account number (that of the current user), so the object knows which balance to retrieve. The diagram includes the actions that occur after the user selects a balance inquiry from the main menu and before the ATM returns the user to the main menu—a BalanceInquiry object does not perform or initiate these actions, so we do not model them here. The diagram begins with retrieving the balance of the account from the database. Next, the BalanceInquiry displays the balance on the screen. This action completes the execution of the transaction. Recall that we’ve chosen to represent an account balance as both the availableBalance and totalBalance attributes of class Account, so the actions modeled in Fig. 12.14 refer to the retrieval and display of both balance attributes.

Fig. 12.14 | Activity diagram for a BalanceInquiry object.

The UML represents an action in an activity diagram as an action state modeled by a rectangle with its left and right sides replaced by arcs curving outward. Each action state contains an action expression—for example, “get balance of account from database”—that specifies an action to be performed. An arrow with a stick (⇒) arrowhead connects two action states, indicating the order in which the actions represented by the action states occur. The solid circle (at the top of Fig. 12.14) represents the activity’s initial state—the beginning of the workflow before the object performs the modeled actions. In this case, the transaction first executes the “get balance of account from database” action expression. The transaction then displays both balances on the screen. The solid circle enclosed in an open circle (at the bottom of Fig. 12.14) represents the final state—the end of the work-
12.5 Identifying Objects’ States and Activities

flow after the object performs the modeled actions. We used UML activity diagrams to illustrate the flow of control for the control statements presented in Chapters 4–5.

Figure 12.15 shows an activity diagram for a withdrawal transaction. We assume that a Withdrawal object has been assigned a valid account number. We do not model the user
selecting a withdrawal from the main menu or the ATM returning the user to the main menu because these are not actions performed by a Withdrawal object. The transaction first displays a menu of standard withdrawal amounts (shown in Fig. 12.3) and an option to cancel the transaction. The transaction then receives a menu selection from the user. The activity flow now arrives at a decision (a fork indicated by the small diamond symbol). This point determines the next action based on the associated guard condition (in square brackets next to the transition), which states that the transition occurs if this guard condition is met. If the user cancels the transaction by choosing the “cancel” option from the menu, the activity flow immediately skips to the final state. Note the merge (indicated by the small diamond symbol) where the cancellation flow of activity joins the main flow of activity before reaching the activity’s final state. If the user selects a withdrawal amount from the menu, Withdrawal sets amount (an attribute originally modeled in Fig. 12.12) to the value chosen by the user.

After setting the withdrawal amount, the transaction retrieves the available balance of the user’s account (i.e., the availableBalance attribute of the user’s Account object) from the database. The activity flow then arrives at another decision. If the requested withdrawal amount exceeds the user’s available balance, the system displays an appropriate error message informing the user of the problem, then returns to the beginning of the activity diagram and prompts the user to input a new amount. If the requested withdrawal amount is less than or equal to the user’s available balance, the transaction proceeds. The transaction next tests whether the cash dispenser has enough cash remaining to satisfy the withdrawal request. If it does not, the transaction displays an appropriate error message, then returns to the beginning of the activity diagram and prompts the user to choose a new amount. If sufficient cash is available, the transaction interacts with the database to debit the withdrawal amount from the user’s account (i.e., subtract the amount from both the availableBalance and totalBalance attributes of the user’s Account object). The transaction then dispenses the desired amount of cash and instructs the user to take it. Finally, the main flow of activity merges with the cancellation flow of activity before reaching the final state.

We’ve taken the first steps in modeling the ATM software system’s behavior and have shown how an object’s attributes participate in performing the object’s activities. In Section 12.6, we investigate the behaviors for all classes to give a more accurate interpretation of the system behavior by filling in the third compartments of the classes in our class diagram.

Self-Review Exercises for Section 12.5

12.11 State whether the following statement is true or false, and if false, explain why: State diagrams model structural aspects of a system.

12.12 An activity diagram models the _________ that an object performs and the order in which it performs them.
   a) actions
   b) attributes
   c) states
   d) state transitions

12.13 Based on the requirements document, create an activity diagram for a deposit transaction.
12.6 Identifying Class Operations

[Note: This section can be taught after Chapter 6.]

In this section, we determine some of the class operations (or behaviors) needed to implement the ATM system. 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’s adjusting the radio’s controls). A car’s operations include accelerating (invoked by the driver’s pressing the accelerator pedal), decelerating (invoked by the driver’s 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 class operations by examining the key verbs and verb phrases in the requirements document. We then relate these verbs and verb phrases to classes in our system (Fig. 12.16). The verb phrases in Fig. 12.16 help us determine the operations of each class.

<table>
<thead>
<tr>
<th>Class</th>
<th>Verbs and verb phrases</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATM</td>
<td>executes financial transactions</td>
</tr>
<tr>
<td>BalanceInquiry</td>
<td>[none in the requirements document]</td>
</tr>
<tr>
<td>Withdrawal</td>
<td>[none in the requirements document]</td>
</tr>
<tr>
<td>Deposit</td>
<td>[none in the requirements document]</td>
</tr>
<tr>
<td>BankDatabase</td>
<td>authenticates a user, retrieves an account balance, credits a deposit amount to an account, debits a withdrawal amount from an account</td>
</tr>
<tr>
<td>Account</td>
<td>retrieves an account balance, credits a deposit amount to an account, debits a withdrawal amount from an account</td>
</tr>
<tr>
<td>Screen</td>
<td>displays a message to the user</td>
</tr>
<tr>
<td>Keypad</td>
<td>receives numeric input from the user</td>
</tr>
<tr>
<td>CashDispenser</td>
<td>dispenses cash, indicates whether it contains enough cash to satisfy a withdrawal request</td>
</tr>
<tr>
<td>DepositSlot</td>
<td>receives a deposit envelope</td>
</tr>
</tbody>
</table>

Fig. 12.16 | Verbs and verb phrases for each class in the ATM system.

Modeling Operations

To identify operations, we examine the verb phrases listed for each class in Fig. 12.16. 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’ve named execute) in the third compartment of the three transaction classes in the updated class diagram of Fig. 12.17. During an ATM session, the ATM object will invoke these transaction operations as necessary.
The UML represents operations (that is, methods) by listing the operation name, followed by a comma-separated list of parameters in parentheses, a colon and the return type:

\[
\text{operationName( parameter1, parameter2, \ldots, parameterN ) : return type}
\]

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

\[
\text{parameterName } : \text{parameterType}
\]

For the moment, we do not list the parameters of our operations—we'll identify and model some of them 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'll add the remaining return types.

**Authenticating a User**

Figure 12.16 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. 12.17). 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. We specify a return type of Boolean for the authenticateUser and validatePIN operations. Each operation returns a value indicating 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).

Other BankDatabase and Account Operations
Figure 12.16 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 12.3 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’ll see in Section 12.7, class ATM invokes the operations of class BankDatabase, each of which in turn invokes the operation with the same name in class Account.

Getting the Balances
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. 12.17). We specify a return type of Double for these operations because the balance attributes they retrieve are of type Double.

Crediting and Debiting an Account
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.
Deposit Confirmations Performed by Another Banking System
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’re beyond the scope of the case study.

Displaying Messages
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. 12.17). We do not worry about the parameter of this operation at this time—we model it later in this section.

Keyboard Input
From the phrase “receives numeric input from the user” listed by class Keypad in Fig. 12.16, 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.]

Dispensing Cash
Figure 12.16 lists “dispenses cash” for class CashDispenser. Therefore, we create operation dispenseCash and list it under class CashDispenser in Fig. 12.17. 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 12.16 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.]

**Class ATM**

We do not list any operations for class ATM at this time. We’re 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 for Class BankDatabase**

So far, we’ve not been concerned with the parameters of our operations—we’ve 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 BankDatabase’s authenticateUser operation. To authenticate a user, this operation must know the account number and PIN supplied by the user. So we specify that authenticateUser takes integer parameters userAccountNumber and userPIN, which the operation must compare to an Account object’s account number and PIN in the database. We prefix these parameter names with “user” to avoid confusion between the operation’s parameter names and class Account’s attribute names. We list these parameters in the class diagram in Fig. 12.18 that models only class BankDatabase. [Note: It’s perfectly normal to model only one class. In this case, we’re examining the parameters of this one class, 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.]

![BankDatabase class diagram](image)

**Fig. 12.18** | Class BankDatabase with operation 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 12.18 thus specifies that operation authenticateUser takes two parameters—userAccountNumber and userPIN, both of type Integer. When we implement the system in Java, we’ll represent these parameters with int values.

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. 12.18. In addition, operations credit and debit each require a Double parameter amount to specify the amount of money to be credited or debited, respectively.

**Identifying and Modeling Operation Parameters for Class Account**

Figure 12.19 models class Account’s operation parameters. Operation validatePIN requires only a userPIN parameter, which contains the user-specified PIN to be compared with the account’s PIN. Like their BankDatabase counterparts, 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. 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.

![Class Account with operation parameters.](image)

**Fig. 12.19** | Class Account with operation parameters.

**Identifying and Modeling Operation Parameters for Class Screen**

Figure 12.20 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. 12.20 refers to the UML type. When we implement the system in Java, we’ll use the Java class String to represent this parameter.

![Class Screen with operation parameters.](image)

**Fig. 12.20** | Class Screen with operation parameters.
Identifying and Modeling Operation Parameters for Class CashDispenser

Figure 12.21 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.

<table>
<thead>
<tr>
<th>CashDispenser</th>
</tr>
</thead>
<tbody>
<tr>
<td>count : Integer = 500</td>
</tr>
<tr>
<td>dispenseCash( amount : Double )</td>
</tr>
<tr>
<td>isSufficientCashAvailable( amount : Double ) : Boolean</td>
</tr>
</tbody>
</table>

Fig. 12.21 | Class CashDispenser with operation parameters.

Identifying and Modeling Operation Parameters for Other Classes

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, so we leave their parameter lists empty. Later, we may decide to add parameters.

In this section, we’ve determined many of the operations performed by the classes in the ATM system. We’ve 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. We also might determine that some of our class operations need additional parameters and different return types, or that some parameters are unnecessary or require different types.

Self-Review Exercises for Section 12.6

12.14 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

12.15 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. 12.17?

12.16 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

12.7 Indicating Collaboration Among Objects

[Note: This section can be taught after Chapter 7.]

In this section, we concentrate on the collaborations (interactions) among objects. When two objects communicate with each other to accomplish a task, they’re said to collaborate—objects do this by invoking one another’s operations. A collaboration consists of an
object of one class sending a **message** to an object of another class. Messages are sent in Java via method calls.

In Section 12.6, we determined many of the operations of the system’s classes. Now, we concentrate on the messages that invoke these operations. To identify the collaborations in the system, we return to the requirements document in Section 12.2. Recall that this document specifies the range of activities that occur during an ATM session (e.g., authenticating a user, performing transactions). The steps used to describe how the system must perform each of these tasks are our first indication of the collaborations in our system. As we proceed through this section and Chapter 13, we may discover additional collaborations.

**Identifying the Collaborations in a System**

We identify the collaborations in the system by carefully reading the sections of the requirements document that specify what the ATM should do to authenticate a user and to perform each transaction type. For each action or step described, we decide which objects in our system must interact to achieve the desired result. We identify one object as the sending object and another as the receiving object. We then select one of the receiving object’s operations (identified in Section 12.6) that must be invoked by the sending object to produce the proper behavior. For example, the ATM displays a welcome message when idle. We know that an object of class **Screen** displays a message to the user via its displayMessage operation. Thus, we decide that the system can display a welcome message by employing a collaboration between the **ATM** and the **Screen** in which the **ATM** sends a displayMessage message to the **Screen** by invoking the displayMessage operation of class **Screen**. [Note: To avoid repeating the phrase “an object of class...,” we refer to an object by using its class name preceded by an article (e.g., “a,” “an” or “the”)—for example, “the **ATM**” refers to an object of class **ATM**.]

Figure 12.22 lists the collaborations that can be derived from the requirements document. For each sending object, we list the collaborations in the order in which they first occur during an ATM session (i.e., the order in which they’re discussed in the requirements document). We list each collaboration involving a unique sender, message and recipient only once, even though the collaborations may occur at several different times throughout an ATM session. For example, the first row in Fig. 12.22 indicates that the **ATM** collaborates with the **Screen** whenever the **ATM** needs to display a message to the user.

Let’s consider the collaborations in Fig. 12.22. Before allowing a user to perform any transactions, the ATM must prompt the user to enter an account number, then to enter a PIN. It accomplishes these tasks by sending a displayMessage message to the **Screen**. Both actions refer to the same collaboration between the **ATM** and the **Screen**, which is already listed in Fig. 12.22. The **ATM** obtains input in response to a prompt by sending a getInput message to the **Keypad**. Next, the **ATM** must determine whether the user-specified account number and PIN match those of an account in the database. It does so by sending an authenticateUser message to the **BankDatabase**. Recall that the **BankDatabase** cannot authenticate a user directly—only the user’s **Account** (i.e., the **Account** that contains the account number specified by the user) can access the user’s PIN on record to authenticate the user. Figure 12.22 therefore lists a collaboration in which the **BankDatabase** sends a validatePIN message to an **Account**.

After the user is authenticated, the **ATM** displays the main menu by sending a series of displayMessage messages to the **Screen** and obtains input containing a menu selection by sending a getInput message to the **Keypad**. We’ve already accounted for these collab-
orations, so we do not add anything to Fig. 12.22. After the user chooses a type of transaction to perform, the ATM executes the transaction by sending an execute message to an object of the appropriate transaction class (i.e., a BalanceInquiry, a Withdrawal or a Deposit). For example, if the user chooses to perform a balance inquiry, the ATM sends an execute message to a BalanceInquiry.

<table>
<thead>
<tr>
<th>An object of class...</th>
<th>sends the message...</th>
<th>to an object of class...</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATM</td>
<td>displayMessage</td>
<td>Screen</td>
</tr>
<tr>
<td></td>
<td>getInput</td>
<td>Keypad</td>
</tr>
<tr>
<td></td>
<td>authenticateUser</td>
<td>BankDatabase</td>
</tr>
<tr>
<td></td>
<td>execute</td>
<td>BalanceInquiry</td>
</tr>
<tr>
<td></td>
<td>execute</td>
<td>Withdrawal</td>
</tr>
<tr>
<td></td>
<td>execute</td>
<td>Deposit</td>
</tr>
<tr>
<td>BalanceInquiry</td>
<td>getAvailableBalance</td>
<td>BankDatabase</td>
</tr>
<tr>
<td></td>
<td>getTotalBalance</td>
<td>BankDatabase</td>
</tr>
<tr>
<td></td>
<td>displayMessage</td>
<td>Screen</td>
</tr>
<tr>
<td>Withdrawal</td>
<td>displayMessage</td>
<td>Screen</td>
</tr>
<tr>
<td></td>
<td>getInput</td>
<td>Keypad</td>
</tr>
<tr>
<td></td>
<td>getAvailableBalance</td>
<td>BankDatabase</td>
</tr>
<tr>
<td></td>
<td>isSufficientCashAvailable</td>
<td>CashDispenser</td>
</tr>
<tr>
<td></td>
<td>debit</td>
<td>BankDatabase</td>
</tr>
<tr>
<td></td>
<td>dispenseCash</td>
<td>CashDispenser</td>
</tr>
<tr>
<td>Deposit</td>
<td>displayMessage</td>
<td>Screen</td>
</tr>
<tr>
<td></td>
<td>getInput</td>
<td>Keypad</td>
</tr>
<tr>
<td></td>
<td>isEnvelopeReceived</td>
<td>DepositSlot</td>
</tr>
<tr>
<td></td>
<td>credit</td>
<td>BankDatabase</td>
</tr>
<tr>
<td>BankDatabase</td>
<td>validatePIN</td>
<td>Account</td>
</tr>
<tr>
<td></td>
<td>getAvailableBalance</td>
<td>Account</td>
</tr>
<tr>
<td></td>
<td>getTotalBalance</td>
<td>Account</td>
</tr>
<tr>
<td></td>
<td>debit</td>
<td>Account</td>
</tr>
<tr>
<td></td>
<td>credit</td>
<td>Account</td>
</tr>
</tbody>
</table>

**Fig. 12.22** | Collaborations in the ATM system.

Further examination of the requirements document reveals the collaborations involved in executing each transaction type. A BalanceInquiry retrieves the amount of money available in the user’s account by sending a getAvailableBalance message to the BankDatabase, which responds by sending a getAvailableBalance message to the user’s Account. Similarly, the BalanceInquiry retrieves the amount on deposit by sending a getTotalBalance message to the BankDatabase, which sends the same message to the user’s Account. To display both parts of the user’s account balance at the same time, the BalanceInquiry sends a displayMessage message to the Screen.

A Withdrawal responds to an execute message by sending displayMessage messages to the Screen to display a menu of standard withdrawal amounts (i.e., $20, $40, $60, $100, $200). The Withdrawal sends a getInput message to the Keypad to obtain the user’s selection. Next, the Withdrawal determines whether the requested amount is less than or equal
to the user’s account balance. The Withdrawal can obtain the amount of money available by sending a getAvailableBalance message to the BankDatabase. The Withdrawal then tests whether the cash dispenser contains enough cash by sending an isSufficientCashAvailable message to the CashDispenser. A Withdrawal sends a debit message to the BankDatabase to decrease the user’s account balance. The BankDatabase in turn sends the same message to the appropriate Account, which decreases both the totalBalance and the availableBalance. To dispense the requested amount of cash, the Withdrawal sends a dispenseCash message to the CashDispenser. Finally, the Withdrawal sends a displayMessage message to the Screen, instructing the user to take the cash.

A Deposit responds to an execute message first by sending a displayMessage message to the Screen to prompt the user for a deposit amount. The Deposit sends a getInput message to the Keypad to obtain the user’s input. The Deposit then sends a displayMessage message to the Screen to tell the user to insert a deposit envelope. To determine whether the deposit slot received an incoming deposit envelope, the Deposit sends an isEnvelopeReceived message to the DepositSlot. The Deposit updates the user’s account by sending a credit message to the BankDatabase, which subsequently sends a credit message to the user’s Account. Recall that crediting funds to an Account increases the totalBalance but not the availableBalance.

Interaction Diagrams
Now that we’ve identified possible collaborations between our ATM system’s objects, let’s graphically model these interactions using the UML. The UML provides several types of interaction diagrams that model the behavior of a system by modeling how objects interact. The communication diagram emphasizes which objects participate in collaborations. Like the communication diagram, the sequence diagram shows collaborations among objects, but it emphasizes when messages are sent between objects over time.

Communication Diagrams
Figure 12.23 shows a communication diagram that models the ATM executing a Balance Inquiry. Objects are modeled in the UML as rectangles containing names in the form objectName : ClassName. In this example, which involves only one object of each type, we disregard the object name and list only a colon followed by the class name. [Note: Specifying each object’s name in a communication diagram is recommended when modeling multiple objects of the same type.] Communicating objects are connected with solid lines, and messages are passed between objects along these lines in the direction shown by arrows. The name of the message, which appears next to the arrow, is the name of an operation (i.e., a method in Java) belonging to the receiving object—think of the name as a “service” that the receiving object provides to sending objects (its clients).

![Communication Diagram](image)

Fig. 12.23 | Communication diagram of the ATM executing a balance inquiry.

The solid filled arrow represents a message—or synchronous call—in the UML and a method call in Java. This arrow indicates that the flow of control is from the sending object
(the ATM) to the receiving object (a BalanceInquiry). Since this is a synchronous call, the sending object can’t send another message, or do anything at all, until the receiving object processes the message and returns control to the sending object. The sender just waits. In Fig. 12.23, the ATM calls BalanceInquiry method execute and can’t send another message until execute has finished and returns control to the ATM. [Note: If this were an asynchronous call, represented by a stick (⇒) arrowhead, the sending object would not have to wait for the receiving object to return control—it would continue sending additional messages immediately following the asynchronous call. Asynchronous calls are implemented in Java using a technique called multithreading, which is discussed in Chapter 26.]

**Sequence of Messages in a Communication Diagram**

Figure 12.24 shows a communication diagram that models the interactions among system objects when an object of class BalanceInquiry executes. We assume that the object’s accountNumber attribute contains the account number of the current user. The collaborations in Fig. 12.24 begin after the ATM sends an execute message to a BalanceInquiry (i.e., the interaction modeled in Fig. 12.23). The number to the left of a message name indicates the order in which the message is passed. The sequence of messages in a communication diagram progresses in numerical order from least to greatest. In this diagram, the numbering starts with message 1 and ends with message 3. The BalanceInquiry first sends a getAvailableBalance message to the BankDatabase (message 1), then sends a getTotalBalance message to the BankDatabase (message 2). Within the parentheses following a message name, we can specify a comma-separated list of the names of the parameters sent with the message (i.e., arguments in a Java method call)—the BalanceInquiry passes attribute accountNumber with its messages to the BankDatabase to indicate which Account’s balance information to retrieve. Recall from Fig. 12.18 that operations getAvailableBalance and getTotalBalance of class BankDatabase each require a parameter to identify an account. The BalanceInquiry next displays the availableBalance and the totalBalance to the user by passing a displayMessage message to the Screen (message 3) that includes a parameter indicating the message to be displayed.

---

**Fig. 12.24** Communication diagram for executing a balance inquiry.
Figure 12.24 models two additional messages passing from the BankDatabase to an Account (message 1.1 and message 2.1). To provide the ATM with the two balances of the user’s Account (as requested by messages 1 and 2), the BankDatabase must pass a getAvailableBalance and a getTotalBalance message to the user’s Account. Such messages passed within the handling of another message are called nested messages. The UML recommends using a decimal numbering scheme to indicate nested messages. For example, message 1.1 is the first message nested in message 1—the BankDatabase passes a getAvailableBalance message during BankDatabase’s processing of a message by the same name. [Note: If the BankDatabase needed to pass a second nested message while processing message 1, the second message would be numbered 1.2.] A message may be passed only when all the nested messages from the previous message have been passed. For example, the BalanceInquiry passes message 3 only after messages 2 and 2.1 have been passed, in that order.

The nested numbering scheme used in communication diagrams helps clarify precisely when and in what context each message is passed. For example, if we numbered the messages in Fig. 12.24 using a flat numbering scheme (i.e., 1, 2, 3, 4, 5), someone looking at the diagram might not be able to determine that BankDatabase passes the getAvailableBalance message (message 1.1) to an Account during the BankDatabase’s processing of message 1, as opposed to after completing the processing of message 1. The nested decimal numbers make it clear that the second getAvailableBalance message (message 1.1) is passed to an Account within the handling of the first getAvailableBalance message (message 1) by the BankDatabase.

Sequence Diagrams
Communication diagrams emphasize the participants in collaborations, but model their timing a bit awkwardly. A sequence diagram helps model the timing of collaborations more clearly. Figure 12.25 shows a sequence diagram modeling the sequence of interactions that occur when a Withdrawal executes. The dotted line extending down from an object’s rectangle is that object’s lifeline, which represents the progression of time. Actions occur along an object’s lifeline in chronological order from top to bottom—an action near the top happens before one near the bottom.

Message passing in sequence diagrams is similar to message passing in communication diagrams. A solid arrow with a filled arrowhead extending from the sending object to the receiving object represents a message between two objects. The arrowhead points to an activation on the receiving object’s lifeline. An activation, shown as a thin vertical rectangle, indicates that an object is executing. When an object returns control, a return message, represented as a dashed line with a stick (\(\rightarrow\)) arrowhead, extends from the activation of the object returning control to the activation of the object that initially sent the message. To eliminate clutter, we omit the return-message arrows—the UML allows this practice to make diagrams more readable. Like communication diagrams, sequence diagrams can indicate message parameters between the parentheses following a message name.

The sequence of messages in Fig. 12.25 begins when a Withdrawal prompts the user to choose a withdrawal amount by sending a DisplayMessage message to the Screen. The Withdrawal then sends a getInput message to the Keypad, which obtains input from the user. We’ve already modeled the control logic involved in a Withdrawal in the activity diagram of Fig. 12.15, so we do not show this logic in the sequence diagram of Fig. 12.25. Instead, we model the best-case scenario in which the balance of the user’s account is
greater than or equal to the chosen withdrawal amount, and the cash dispenser contains a sufficient amount of cash to satisfy the request. You can model control logic in a sequence diagram with UML frames (which are not covered in this case study). For a quick overview of UML frames, visit www.agilemodeling.com/style/frame.htm.

After obtaining a withdrawal amount, the Withdrawal sends a getAvailableBalance message to the BankDatabase, which in turn sends a getAvailableBalance message to the user’s Account. Assuming that the user’s account has enough money available to permit the transaction, the Withdrawal next sends an isSufficientCashAvailable message to the CashDispenser. Assuming that there’s enough cash available, the Withdrawal decreases the balance of the user’s account (i.e., both the totalBalance and the availableBalance) by sending a debit message to the BankDatabase. The BankDatabase responds by sending a debit message to the user’s Account. Finally, the Withdrawal sends
a dispenseCash message to the CashDispenser and a displayMessage message to the Screen, telling the user to remove the cash from the machine.

We’ve identified the collaborations among objects in the ATM system and modeled some of them using UML interaction diagrams—both communication diagrams and sequence diagrams. In Section 13.2, we enhance the structure of our model to complete a preliminary object-oriented design, then we begin implementing the ATM system in Java.

**Self-Review Exercises for Section 12.7**

12.17 A(n) _____ consists of an object of one class sending a message to an object of another class.
   a) association  
   b) aggregation  
   c) collaboration  
   d) composition

12.18 Which form of interaction diagram emphasizes what collaborations occur? Which form emphasizes when collaborations occur?

12.19 Create a sequence diagram that models the interactions among objects in the ATM system that occur when a deposit executes successfully, and explain the sequence of messages modeled by the diagram.

**12.8 Wrap-Up**

In this chapter, you learned how to work from a detailed requirements document to develop an object-oriented design. You worked with six popular types of UML diagrams to graphically model an object-oriented automated teller machine software system. In Chapter 13, we tune the design using inheritance, then completely implement the design in a 673-line Java application.

---

**Answers to Self-Review Exercises**

12.1 Figure 12.26 contains a use case diagram for a modified version of our ATM system that also allows users to transfer money between accounts.

12.2 b.

![Use case diagram](image-url)

**Fig. 12.26** Use case diagram for a modified version of our ATM system that also allows users to transfer money between accounts.
12.3  d.
12.4  [Note: Student answers may vary.] Figure 12.27 presents a class diagram that shows some of the composition relationships of a class Car.

12.5  c. [Note: In a computer network, this relationship could be many-to-many.]
12.6  True.
12.7  Figure 12.28 presents a class diagram for the ATM including class Deposit instead of class Withdrawal (as in Fig. 12.10). Deposit does not access CashDispenser, but does access DepositSlot.

---

**Fig. 12.27**  | Class diagram showing composition relationships of a class Car.

**Fig. 12.28**  | Class diagram for the ATM system model including class Deposit.
12.8  b.
12.9  c. Fly is an operation or behavior of an airplane, not an attribute.
12.10 This indicates that count is an integer with an initial value of 500. This attribute keeps
track of the number of bills available in the CashDispenser at any given time.
12.11 False. State diagrams model some of the behavior of a system.
12.12 a.
12.13 Figure 12.29 models the actions that occur after the user chooses the deposit option from
the main menu and before the ATM returns the user to the main menu. Recall that part of receiving
a deposit amount from the user involves converting an integer number of cents to a dollar amount.
Also recall that crediting a deposit amount to an account increases only the totalAvailable
attribute of the user’s Account object. The bank updates the availableBalance attribute of the user’s Account
object only after confirming the amount of cash in the deposit envelope and after the enclosed
checks clear—this occurs independently of the ATM system.

Fig. 12.29  |  Activity diagram for a deposit transaction.
12.14 c.

12.15 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`:

```java
getAmount() : Double
```

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

12.17 c.

12.18 Communication diagrams emphasize *what* collaborations occur. Sequence diagrams emphasize *when* collaborations occur.

12.19 Figure 12.30 presents a sequence diagram that models the interactions between objects in the ATM system that occur when a `Deposit` executes successfully. A `Deposit` first sends a `displayMessage` message to the `Screen` to ask the user to enter a deposit amount. Next the `Deposit` sends a `getInput` message to the `Keypad` to receive input from the user. The `Deposit` then instructs the user to enter a deposit envelope by sending a `displayMessage` message to the `Screen`. The `Deposit` next sends an `isEnvelopeReceived` message to the `DepositSlot` to confirm that the deposit envelope has been received by the ATM. Finally, the `Deposit` increases the `totalBalance` attribute (but not the `availableBalance` attribute) of the user’s `Account` by sending a `credit` message to the `BankDatabase`. The `BankDatabase` responds by sending the same message to the user’s `Account`.

![Sequence diagram](image)
ATM Case Study Part 2:
Implementing an Object-Oriented Design

You can’t work in the abstract.
—L. M. Pei

To generalize means to think.
—Georg Wilhelm Friedrich Hegel

We are all gifted. That is our inheritance.
—Ethel Waters

Let me walk through the fields
of paper
touching with my wand
dry stems and stunted
butterflies…
—Denise Levertov

Objectives
In this chapter you’ll:

■ Incorporate inheritance into
  the design of the ATM.

■ Incorporate polymorphism
  into the design of the ATM.

■ Fully implement in Java the
  UML-based object-oriented
  design of the ATM software.

■ Study a detailed code
  walkthrough of the ATM
  software system that explains
  the implementation issues.
13.1 Introduction

In Chapter 12, we developed an object-oriented design for our ATM system. We now implement our object-oriented design in Java. In Section 13.2, we show how to convert class diagrams to Java code. In Section 13.3, we tune the design with inheritance and polymorphism. Then we present a full Java code implementation of the ATM software in Section 13.4. The code is carefully commented and the discussions of the implementation are thorough and precise. Studying this application provides the opportunity for you to see a more substantial application of the kind you’re likely to encounter in industry.

13.2 Starting to Program the Classes of the ATM System

[Note: This section can be taught after Chapter 8.]

Visibility

We now apply access modifiers to the members of our classes. We’ve introduced access modifiers public and private. Access modifiers determine the visibility or accessibility of an object’s attributes and methods to other objects. Before we can begin implementing our design, we must consider which attributes and methods of our classes should be public and which should be private.

We’ve observed that attributes normally should be private and that methods invoked by clients of a given class should be public. Methods that are called as “utility methods” only by other methods of the same class normally should be private. The UML employs visibility markers for modeling the visibility of attributes and operations. Public visibility is indicated by placing a plus sign (+) before an operation or an attribute, whereas a minus sign (−) indicates private visibility. Figure 13.1 shows our updated class diagram with visibility markers included. [Note: We do not include any operation parameters in Fig. 13.1—this is perfectly normal. Adding visibility markers does not affect the parameters already modeled in the class diagrams of Figs. 12.17–12.21.]

Navigability

Before we begin implementing our design in Java, we introduce an additional UML notation. The class diagram in Fig. 13.2 further refines the relationships among classes in the ATM system by adding navigability arrows to the association lines. Navigability arrows (represented as arrows with stick (⇒) arrowheads in the class diagram) indicate in the
direction which an association can be traversed. When implementing a system designed using the UML, you use navigability arrows to determine which objects need references to other objects. For example, the navigability arrow pointing from class ATM to class BankDatabase indicates that we can navigate from the former to the latter, thereby enabling the ATM to invoke the BankDatabase’s operations. However, since Fig. 13.2 does not contain a navigability arrow pointing from class BankDatabase to class ATM, the BankDatabase cannot access the ATM’s operations. Associations in a class diagram that have navigability arrows at both ends or have none at all indicate bidirectional navigability—navigation can proceed in either direction across the association.

Like the class diagram of Fig. 12.10, that of Fig. 13.2 omits classes BalanceInquiry and Deposit for simplicity. The navigability of the associations in which these classes participate closely parallels that of class Withdrawal. Recall from Section 12.3 that BalanceInquiry has an association with class Screen. We can navigate from class BalanceInquiry to class Screen along this association, but we cannot navigate from class Screen to class BalanceInquiry. Thus, if we were to model class BalanceInquiry in Fig. 13.2, we would place a navigability
arrow at class Screen’s end of this association. Also recall that class Deposit associates with classes Screen, Keypad and DepositSlot. We can navigate from class Deposit to each of these classes, but not vice versa. We therefore would place navigability arrows at the Screen, Keypad and DepositSlot ends of these associations. [Note: We model these additional classes and associations in our final class diagram in Section 13.3, after we’ve simplified the structure of our system by incorporating the object-oriented concept of inheritance.]

**Implementing the ATM System from Its UML Design**

We’re now ready to begin implementing the ATM system. We first convert the classes in the diagrams of Fig. 13.1 and Fig. 13.2 into Java code. The code will represent the “skeleton” of the system. In Section 13.3, we modify the code to incorporate inheritance. In Section 13.4, we present the complete working Java code for our model.

As an example, we develop the code from our design of class Withdrawal in Fig. 13.1. We use this figure to determine the attributes and operations of the class. We use the UML model in Fig. 13.2 to determine the associations among classes. We follow the following four guidelines for each class:

1. Use the name located in the first compartment to declare the class as a `public` class with an empty no-argument constructor. We include this constructor simply as a placeholder to remind us that `most classes will indeed need custom constructors`. In Section 13.4, when we complete a working version of this class, we’ll add arguments and code the body of the constructor as needed. For example, class
Chapter 13  ATM Case Study Part 2: Implementing an Object-Oriented Design

Withdrawal yields the code in Fig. 13.3. If we find that the class's instance variables require only default initialization, then we'll remove the empty no-argument constructor because it's unnecessary.

```
1 // Class Withdrawal represents an ATM withdrawal transaction
2 public class Withdrawal
3 {
4     // no-argument constructor
5     public Withdrawal()
6     {
7     } // end no-argument Withdrawal constructor
8 } // end class Withdrawal
```

Fig. 13.3 | Java code for class Withdrawal based on Figs. 13.1–13.2.

2. Use the attributes located in the second compartment to declare the instance variables. For example, the private attributes accountNumber and amount of class Withdrawal yield the code in Fig. 13.4. [Note: The constructor of the complete working version of this class will assign values to these attributes.]

```
1 // Class Withdrawal represents an ATM withdrawal transaction
2 public class Withdrawal
3 {
4     // attributes
5     private int accountNumber; // account to withdraw funds from
6     private double amount; // amount to withdraw
7     // no-argument constructor
8     public Withdrawal()
9     {
10    } // end no-argument Withdrawal constructor
11 } // end class Withdrawal
```

Fig. 13.4 | Java code for class Withdrawal based on Figs. 13.1–13.2.

3. Use the associations described in the class diagram to declare the references to other objects. For example, according to Fig. 13.2, Withdrawal can access one object of class Screen, one object of class Keypad, one object of class CashDispenser and one object of class BankDatabase. This yields the code in Fig. 13.5. [Note: The constructor of the complete working version of this class will initialize these instance variables with references to actual objects.]

4. Use the operations located in the third compartment of Fig. 13.1 to declare the shells of the methods. If we have not yet specified a return type for an operation, we declare the method with return type void. Refer to the class diagrams of Figs. 12.17–12.21 to declare any necessary parameters. For example, adding the public operation execute in class Withdrawal, which has an empty parameter list, yields the code in Fig. 13.6. [Note: We code the bodies of methods when we implement the complete system in Section 13.4.]

This concludes our discussion of the basics of generating classes from UML diagrams.
Self-Review Exercises for Section 13.2

1. Class Withdrawal represents an ATM withdrawal transaction
   public class Withdrawal
   {
   // attributes
   private int accountNumber; // account to withdraw funds from
   private double amount; // amount to withdraw
   
   // references to associated objects
   private Screen screen; // ATM's screen
   private Keypad keypad; // ATM's keypad
   private CashDispenser cashDispenser; // ATM's cash dispenser
   private BankDatabase bankDatabase; // account info database
   
   // no-argument constructor
   public Withdrawal()
   {
   } // end no-argument Withdrawal constructor
   
   } // end class Withdrawal

Fig. 13.5 | Java code for class Withdrawal based on Figs. 13.1–13.2.

2. Class Withdrawal represents an ATM withdrawal transaction
   public class Withdrawal
   {
   // attributes
   private int accountNumber; // account to withdraw funds from
   private double amount; // amount to withdraw
   
   // references to associated objects
   private Screen screen; // ATM's screen
   private Keypad keypad; // ATM's keypad
   private CashDispenser cashDispenser; // ATM's cash dispenser
   private BankDatabase bankDatabase; // account info database
   
   // no-argument constructor
   public Withdrawal()
   {
   } // end no-argument Withdrawal constructor
   
   // operations
   public void execute()
   {
   } // end method execute
   
   } // end class Withdrawal

Fig. 13.6 | Java code for class Withdrawal based on Figs. 13.1–13.2.

Self-Review Exercises for Section 13.2

13.1 State whether the following statement is true or false, and if false, explain why: If an attribute of a class is marked with a minus sign (-) in a class diagram, the attribute is not directly accessible outside the class.
13.2 In Fig. 13.2, the association between the ATM and the Screen indicates that:
   a) we can navigate from the Screen to the ATM
   b) we can navigate from the ATM to the Screen
   c) Both (a) and (b); the association is bidirectional
   d) None of the above

13.3 Write Java code to begin implementing the design for class Keypad.

13.3 Incorporating Inheritance and Polymorphism into the ATM System

[Note: This section can be taught after Chapter 10.]

We now revisit our ATM system design to see how it might benefit from inheritance. To apply inheritance, we first look for commonality among classes in the system. We create an inheritance hierarchy to model similar (yet not identical) classes in a more elegant and efficient manner. We then modify our class diagram to incorporate the new inheritance relationships. Finally, we demonstrate how our updated design is translated into Java code.

In Section 12.3, we encountered the problem of representing a financial transaction in the system. Rather than create one class to represent all transaction types, we decided to create three individual transaction classes—BalanceInquiry, Withdrawal and Deposit—to represent the transactions that the ATM system can perform. Figure 13.7 shows the attributes and operations of classes BalanceInquiry, Withdrawal and Deposit. These classes have one attribute (accountNumber) and one operation (execute) in common. Each class requires attribute accountNumber to specify the account to which the transaction applies. Each class contains operation execute, which the ATM invokes to perform the transaction. Clearly, BalanceInquiry, Withdrawal and Deposit represent types of transactions. Figure 13.7 reveals commonality among the transaction classes, so using inheritance to factor out the common features seems appropriate for designing classes BalanceInquiry, Withdrawal and Deposit. We place the common functionality in a superclass, Transaction, that classes BalanceInquiry, Withdrawal and Deposit extend.

Fig. 13.7 | Attributes and operations of BalanceInquiry, Withdrawal and Deposit.

Generalization

The UML specifies a relationship called a generalization to model inheritance. Figure 13.8 is the class diagram that models the generalization of superclass Transaction and subclasses BalanceInquiry, Withdrawal and Deposit. The arrows with triangular
hollow arrowheads indicate that classes BalanceInquiry, Withdrawal and Deposit extend class Transaction. Class Transaction is said to be a generalization of classes BalanceInquiry, Withdrawal and Deposit. Class BalanceInquiry, Withdrawal and Deposit are said to be specializations of class Transaction.

**Fig. 13.8** | Class diagram modeling generalization of superclass Transaction and subclasses BalanceInquiry, Withdrawal and Deposit. Abstract class names (e.g., Transaction) and method names (e.g., execute in class Transaction) appear in italics.

Classes BalanceInquiry, Withdrawal and Deposit share integer attribute accountNumber, so we factor out this common attribute and place it in superclass Transaction. We no longer list accountNumber in the second compartment of each subclass, because the three subclasses inherit this attribute from Transaction. Recall, however, that subclasses cannot directly access private attributes of a superclass. We therefore include public method getAccountNumber in class Transaction. Each subclass will inherit this method, enabling the subclass to access its accountNumber as needed to execute a transaction.

According to Fig. 13.7, classes BalanceInquiry, Withdrawal and Deposit also share operation execute, so we placed public method execute in superclass Transaction. However, it does not make sense to implement execute in class Transaction, because the functionality that this method provides depends on the type of the actual transaction. We therefore declare method execute as abstract in superclass Transaction. Any class that contains at least one abstract method must also be declared abstract. This forces any subclass of Transaction that must be a concrete class (i.e., BalanceInquiry, Withdrawal and Deposit) to implement method execute. The UML requires that we place abstract class names (and abstract methods) in italics, so Transaction and its method execute appear in italics in Fig. 13.8. Method execute is not italicized in subclasses BalanceInquiry, Withdrawal and Deposit. Each subclass overrides superclass Transaction’s execute method with a concrete implementation that performs the steps appropriate for completing that type of transaction. Figure 13.8 includes operation execute in the third compartment of classes BalanceInquiry, Withdrawal and Deposit, because each class has a different concrete implementation of the overridden method.

**Processing Transactions Polymorphically**

Polymorphism provides the ATM with an elegant way to execute all transactions “in the general.” For example, suppose a user chooses to perform a balance inquiry. The ATM sets a
Transaction reference to a new BalanceInquiry object. When the ATM uses its Transaction reference to invoke method execute, BalanceInquiry’s version of execute is called.

This polymorphic approach also makes the system easily extensible. Should we wish to create a new transaction type (e.g., funds transfer or bill payment), we would just create an additional Transaction subclass that overrides the execute method with a version of the method appropriate for executing the new transaction type. We would need to make only minimal changes to the system code to allow users to choose the new transaction type from the main menu and for the ATM to instantiate and execute objects of the new subclass. The ATM could execute transactions of the new type using the current code, because it executes all transactions polymorphically using a general Transaction reference.

Recall that an abstract class like Transaction is one for which you never intend to instantiate objects. An abstract class simply declares common attributes and behaviors of its subclasses in an inheritance hierarchy. Class Transaction defines the concept of what it means to be a transaction that has an account number and executes. You may wonder why we bother to include abstract method execute in class Transaction if it lacks a concrete implementation. Conceptually, we include it because it corresponds to the defining behavior of all transactions—executing. Technically, we must include method execute in superclass Transaction so that the ATM (or any other class) can polymorphically invoke each subclass’s overridden version of this method through a Transaction reference. Also, from a software engineering perspective, including an abstract method in a superclass forces the implementor of the subclasses to override that method with concrete implementations in the subclasses, or else the subclasses, too, will be abstract, preventing objects of those subclasses from being instantiated.

Additional Attribute of Classes Withdrawal and Deposit
Subclasses BalanceInquiry, Withdrawal and Deposit inherit attribute accountNumber from superclass Transaction, but classes Withdrawal and Deposit contain the additional attribute amount that distinguishes them from class BalanceInquiry. Classes Withdrawal and Deposit require this additional attribute to store the amount of money that the user wishes to withdraw or deposit. Class BalanceInquiry has no need for such an attribute and requires only an account number to execute. Even though two of the three Transaction subclasses share this attribute, we do not place it in superclass Transaction—we place only features common to all the subclasses in the superclass, otherwise subclasses could inherit attributes (and methods) that they do not need and should not have.

Class Diagram with Transaction Hierarchy Incorporated
Figure 13.9 presents an updated class diagram of our model that incorporates inheritance and introduces class Transaction. We model an association between class ATM and class Transaction to show that the ATM, at any given moment, either is executing a transaction or is not (i.e., zero or one objects of type Transaction exist in the system at a time). Because a Withdrawal is a type of Transaction, we no longer draw an association line directly between class ATM and class Withdrawal. Subclass Withdrawal inherits superclass Transaction’s association with class ATM. Subclasses BalanceInquiry and Deposit inherit this association, too, so the previously omitted associations between ATM and classes BalanceInquiry and Deposit no longer exist either.
Incorporating Inheritance and Polymorphism into the ATM System

We also add an association between class Transaction and the BankDatabase (Fig. 13.9). All Transactions require a reference to the BankDatabase so they can access and modify account information. Because each Transaction subclass inherits this reference, we no longer model the association between class Withdrawal and the BankDatabase. Similarly, the previously omitted associations between the BankDatabase and classes BalanceInquiry and Deposit no longer exist.

We show an association between class Transaction and the Screen. All Transactions display output to the user via the Screen. Thus, we no longer include the association previously modeled between Withdrawal and the Screen, although Withdrawal still participates in associations with the CashDispenser and the Keypad. Our class diagram incorporating inheritance also models Deposit and BalanceInquiry. We show associations between Deposit and both the DepositSlot and the Keypad. Class BalanceInquiry takes part in no associations other than those inherited from class Transaction—a BalanceInquiry needs to interact only with the BankDatabase and with the Screen.

Figure 13.1 showed attributes and operations with visibility markers. Now in Fig. 13.10 we present a modified class diagram that incorporates inheritance. This abbreviated diagram does not show inheritance relationships, but instead shows the attributes and methods after we’ve employed inheritance in our system. To save space, as we did in
Fig. 12.12, we do not include those attributes shown by associations in Fig. 13.9—we do, however, include them in the Java implementation in Section 13.4. We also omit all operation parameters, as we did in Fig. 13.1—incorporating inheritance does not affect the parameters already modeled in Figs. 12.17–12.21.

**Software Engineering Observation 13.1**

A complete class diagram shows all the associations among classes and all the attributes and operations for each class. When the number of class attributes, methods and associations is substantial (as in Figs. 13.9 and 13.10), a good practice that promotes readability is to divide this information between two class diagrams—one focusing on associations and the other on attributes and methods.

---

**Fig. 13.10** | Class diagram with attributes and operations (incorporating inheritance). The abstract class name `Transaction` and the abstract method name `execute` in class `Transaction` appear in italics.
13.3 Incorporating Inheritance and Polymorphism into the ATM System

Implementing the ATM System Design (Incorporating Inheritance)

In Section 13.2, we began implementing the ATM system design in Java code. We now modify our implementation to incorporate inheritance, using class Withdrawal as an example.

1. If a class A is a generalization of class B, then class B extends class A in the class declaration. For example, abstract superclass Transaction is a generalization of class Withdrawal. Figure 13.11 shows the declaration of class Withdrawal.

```java
1 // Class Withdrawal represents an ATM withdrawal transaction
2 public class Withdrawal extends Transaction
3 } // end class Withdrawal
```

**Fig. 13.11** | Java code for shell of class Withdrawal.

2. If class A is an abstract class and class B is a subclass of class A, then class B must implement the abstract methods of class A if class B is to be a concrete class. For example, class Transaction contains abstract method execute, so class Withdrawal must implement this method if we want to instantiate a Withdrawal object. Figure 13.12 is the Java code for class Withdrawal from Fig. 13.9 and Fig. 13.10. Class Withdrawal inherits field accountNumber from superclass Transaction, so Withdrawal does not need to declare this field. Class Withdrawal also inherits references to the Screen and the BankDatabase from its superclass Transaction, so we do not include these references in our code. Figure 13.10 specifies attribute amount and operation execute for class Withdrawal. Line 6 of Fig. 13.12 declares a field for attribute amount. Lines 16–19 declare the shell of a method for operation execute. Recall that subclass Withdrawal must provide a concrete implementation of the abstract method execute in superclass Transaction. The keypad and cashDispenser references (lines 7–8) are fields derived from Withdrawal’s associations in Fig. 13.9. The constructor in the complete working version of this class will initialize these references to actual objects.

```java
1 // Withdrawal.java
2 // Generated using the class diagrams in Fig. 13.9 and Fig. 13.10
3 public class Withdrawal extends Transaction
4 {
5     // attributes
6     private double amount; // amount to withdraw
7     private Keypad keypad; // reference to keypad
8     private CashDispenser cashDispenser; // reference to cash dispenser
9
10     // no-argument constructor
11     public Withdrawal()
12     { // end no-argument Withdrawal constructor
13     }
```

**Fig. 13.12** | Java code for class Withdrawal based on Figs. 13.9 and 13.10. (Part 1 of 2.)
Congratulations on completing the case study’s design portion! We implement the ATM system in Java code in Section 13.4. We recommend that you carefully read the code and its description. The code is abundantly commented and precisely follows the design with which you’re now familiar. The accompanying description is carefully written to guide your understanding of the implementation based on the UML design. Mastering this code is a wonderful culminating accomplishment after studying Sections 12.2–12.7 and 13.2–13.3.

**Self-Review Exercises for Section 13.3**

13.4 The UML uses an arrow with a __________ to indicate a generalization relationship.
   a) solid filled arrowhead
   b) triangular hollow arrowhead
   c) diamond-shaped hollow arrowhead
   d) stick arrowhead

13.5 State whether the following statement is true or false, and if false, explain why: The UML requires that we underline abstract class names and method names.

13.6 Write Java code to begin implementing the design for class Transaction specified in Figs. 13.9 and 13.10. Be sure to include private reference-type attributes based on class Transaction’s associations. Also be sure to include public get methods that provide access to any of these private attributes that the subclasses require to perform their tasks.

**13.4 ATM Case Study Implementation**

This section contains the complete working 673-line implementation of the ATM system. We consider the classes in the order in which we identified them in Section 12.3—ATM, Screen, Keypad, CashDispenser, DepositSlot, Account, BankDatabase, Transaction, BalanceInquiry, Withdrawal and Deposit.

We apply the guidelines discussed in Sections 13.2–13.3 to code these classes based on how we modeled them in the UML class diagrams of Figs. 13.9 and 13.10. To develop the bodies of class methods, we refer to the activity diagrams in Section 12.5 and the communication and sequence diagrams presented in Section 12.7. Our ATM design does not specify all the program logic and may not specify all the attributes and operations required to complete the ATM implementation. This is a normal part of the object-oriented design.
process. As we implement the system, we complete the program logic and add attributes and behaviors as necessary to construct the ATM system specified by the requirements document in Section 12.2.

We conclude the discussion by presenting a Java application (ATMCaseStudy) that starts the ATM and puts the other classes in the system in use. Recall that we’re developing a first version of the ATM system that runs on a personal computer and uses the computer’s keyboard and monitor to approximate the ATM’s keypad and screen. We also simulate only the actions of the ATM’s cash dispenser and deposit slot. We attempt to implement the system, however, so that real hardware versions of these devices could be integrated without significant changes in the code.

13.4.1 Class ATM

Class ATM (Fig. 13.13) represents the ATM as a whole. Lines 6–12 implement the class’s attributes. We determine all but one of these attributes from the UML class diagrams of Figs. 13.9 and 13.10. We implement the UML Boolean attribute userAuthenticated in Fig. 13.10 as a boolean in Java (line 6). Line 7 declares an attribute not found in our UML design—an int attribute currentAccountNumber that keeps track of the account number of the current authenticated user. We’ll soon see how the class uses this attribute. Lines 8–12 declare reference-type attributes corresponding to the ATM class’s associations modeled in the class diagram of Fig. 13.9. These attributes allow the ATM to access its parts (i.e., its Screen, Keypad, CashDispenser and DepositSlot) and interact with the bank’s account-information database (i.e., a BankDatabase object).

```java
1 // ATM.java
2 // Represents an automated teller machine
3 public class ATM
4 {
5     private boolean userAuthenticated; // whether user is authenticated
6     private int currentAccountNumber; // current user’s account number
7     private Screen screen; // ATM’s screen
8     private Keypad keypad; // ATM’s keypad
9     private CashDispenser cashDispenser; // ATM’s cash dispenser
10    private DepositSlot depositSlot; // ATM’s deposit slot
11    private BankDatabase bankDatabase; // account information database
12
13    // constants corresponding to main menu options
14    private static final int BALANCE_INQUIRY = 1;
15    private static final int WITHDRAWAL = 2;
16    private static final int DEPOSIT = 3;
17    private static final int EXIT = 4;
18
19    // no-argument ATM constructor initializes instance variables
20    public ATM()
21    {
22        userAuthenticated = false; // user is not authenticated to start
23        currentAccountNumber = 0; // no current account number to start
24        screen = new Screen(); // create screen
```

*Fig. 13.13* | Class ATM represents the ATM. (Part 1 of 4.)
keypad = new Keypad(); // create keypad
cashDispenser = new CashDispenser(); // create cash dispenser
depositSlot = new DepositSlot(); // create deposit slot
bankDatabase = new BankDatabase(); // create acct info database

// start ATM
public void run()
{
    // welcome and authenticate user; perform transactions
    while (true)
    {
        // loop while user is not yet authenticated
        while (!userAuthenticated)
        {
            screen.displayMessageLine("Welcome!");
            authenticateUser(); // authenticate user
        } // end while
        performTransactions(); // user is now authenticated
        userAuthenticated = false; // reset before next ATM session
        currentAccountNumber = 0; // reset before next ATM session
        screen.displayMessageLine("Thank you! Goodbye!");
    } // end while
} // end method run

// attempt to authenticate user against database
private void authenticateUser()
{
    screen.displayMessage("Please enter your account number: ");
    int accountNumber = keypad.getInput(); // input account number
    screen.displayMessage("Enter your PIN: "); // prompt for PIN
    int pin = keypad.getInput(); // input PIN

    // set userAuthenticated to boolean value returned by database
    userAuthenticated = bankDatabase.authenticateUser(accountNumber, pin);

    // check whether authentication succeeded
    if (userAuthenticated)
    {
        currentAccountNumber = accountNumber; // save user's account #
    } // end if
    else
        screen.displayMessageLine("Invalid account number or PIN. Please try again.");
} // end method authenticateUser

// display the main menu and perform transactions
private void performTransactions()
{
    // local variable to store transaction currently being processed
    Transaction currentTransaction = null;

Fig. 13.13 | Class ATM represents the ATM. (Part 2 of 4.)
boolean userExited = false; // user has not chosen to exit

// loop while user has not chosen option to exit system
while (!userExited)
{
    // show main menu and get user selection
    int mainMenuSelection = displayMainMenu();

    // decide how to proceed based on user's menu selection
    switch (mainMenuSelection)
    {
        case BALANCE_INQUIRY:
        case WITHDRAWAL:
        case DEPOSIT:
        {
            currentTransaction = createTransaction(mainMenuSelection);
            currentTransaction.execute(); // execute transaction
            break;
        }
        case EXIT:
        {
            screen.displayMessageLine("\nExiting the system...\n");
            userExited = true; // this ATM session should end
            break;
        }
        default: // user did not enter an integer from 1-4
        {
            screen.displayMessageLine("\nYou did not enter a valid selection. Try again.\n");
            break;
        }
    } // end switch

} // end while

} // end method performTransactions

private int displayMainMenu()
{
    screen.displayMessageLine("\nMain menu:\n");
    screen.displayMessageLine("1 - View my balance\n");
    screen.displayMessageLine("2 - Withdraw cash\n");
    screen.displayMessageLine("3 - Deposit funds\n");
    screen.displayMessageLine("4 - Exit\n");
    screen.displayMessage(screen.displayMessage("Enter a choice: ");
    return keypad.getInput(); // return user's selection
} // end method displayMainMenu

// return object of specified Transaction subclass
private Transaction createTransaction(int type)
{
    Transaction temp = null; // temporary Transaction variable

Fig. 13.13 | Class ATM represents the ATM. (Part 3 of 4.)
// determine which type of Transaction to create
switch ( type )
{
    case BALANCE_INQUIRY: // create new BalanceInquiry transaction
        temp = new BalanceInquiry(
            currentAccountNumber, screen, bankDatabase );
        break;
    case WITHDRAWAL: // create new Withdrawal transaction
        temp = new Withdrawal( currentAccountNumber, screen,
            bankDatabase, keypad, cashDispenser );
        break;
    case DEPOSIT: // create new Deposit transaction
        temp = new Deposit( currentAccountNumber, screen,
            bankDatabase, keypad, depositSlot );
        break;
    } // end switch
    return temp; // return the newly created object
} // end method createTransaction

Fig. 13.13 | Class ATM represents the ATM. (Part 4 of 4.)

Lines 15–18 declare integer constants that correspond to the four options in the
ATM’s main menu (i.e., balance inquiry, withdrawal, deposit and exit). Lines 21–30
declare the constructor, which initializes the class’s attributes. When an ATM object is first
created, no user is authenticated, so line 23 initializes userAuthenticated to false. Like-
wise, line 24 initializes currentAccountNumber to 0 because there’s no current user yet.
Lines 25–28 instantiate new objects to represent the ATM’s parts. Recall that class ATM has
composition relationships with classes Screen, Keypad, CashDispenser and DepositSlot, so
class ATM is responsible for their creation. Line 29 creates a new BankDatabase. [Note:
If this were a real ATM system, the ATM class would receive a reference to an existing data-
base object created by the bank. However, in this implementation we’re only simulating
the bank’s database, so class ATM creates the BankDatabase object with which it interacts.]

ATM Method run
The class diagram of Fig. 13.10 does not list any operations for class ATM. We now imple-
ment one operation (i.e., public method) in class ATM that allows an external client of the
class (i.e., class ATMCaseStudy) to tell the ATM to run. ATM method run (lines 33–50) uses
an infinite loop (lines 36–49) to repeatedly welcome a user, attempt to authenticate the
user and, if authentication succeeds, allow the user to perform transactions. After an au-
thenticated user performs the desired transactions and chooses to exit, the ATM resets it-
self, displays a goodbye message to the user and restarts the process. We use an infinite loop
here to simulate the fact that an ATM appears to run continuously until the bank turns it
off (an action beyond the user’s control). An ATM user has the option to exit the system
but not the ability to turn off the ATM completely.

Authenticating a User
In method run’s infinite loop, lines 39–43 cause the ATM to repeatedly welcome and at-
tempt to authenticate the user as long as the user has not been authenticated (i.e., !user-
Authenticated is true). Line 41 invokes method `displayMessage` of the ATM's screen to display a welcome message. Like Screen method `displayMessage` designed in the case study, method `displayMessage` (declared in lines 13–16 of Fig. 13.14) displays a message to the user, but this method also outputs a newline after the message. We've added this method during implementation to give class Screen's clients more control over the placement of displayed messages. Line 42 invokes class ATM's private utility method `authenticateUser` (declared in lines 53–72) to attempt to authenticate the user.

We refer to the requirements document to determine the steps necessary to authenticate the user before allowing transactions to occur. Line 55 of method `authenticateUser` invokes method `displayMessage` of the screen to prompt the user to enter an account number. Line 56 invokes method `getInput` of the keypad to obtain the user's input, then stores the integer value entered by the user in a local variable `accountNumber`. Method `authenticateUser` next prompts the user to enter a PIN (line 57), and stores the PIN input by the user in a local variable `pin` (line 58). Next, lines 61–62 attempt to authenticate the user by passing the `accountNumber` and `pin` entered by the user to the `bankDatabase`'s `authenticateUser` method. Class ATM sets its `userAuthenticated` attribute to the `boolean` value returned by this method—`userAuthenticated` becomes true if authentication succeeds (i.e., `accountNumber` and `pin` match those of an existing Account in `bankDatabase`) and remains false otherwise. If `userAuthenticated` is true, line 67 saves the account number entered by the user (i.e., `accountNumber`) in the ATM attribute `currentAccountNumber`. The other ATM methods use this variable whenever an ATM session requires access to the user's account number. If `userAuthenticated` is false, lines 70–71 use the screen's `displayMessage` method to indicate that an invalid account number and/or PIN was entered and the user must try again. We set `currentAccountNumber` only after authenticating the user's account number and the associated PIN—if the database could not authenticate the user, `currentAccountNumber` remains 0.

After method `run` attempts to authenticate the user (line 42), if `userAuthenticated` is still false, the `while` loop in lines 39–43 executes again. If `userAuthenticated` is now true, the loop terminates and control continues with line 45, which calls class ATM's utility method `performTransactions`.

**Performing Transactions**

Method `performTransactions` (lines 75–112) carries out an ATM session for an authenticated user. Line 78 declares a local `Transaction` variable to which we'll assign a `BalanceInquiry`, `Withdrawal` or `Deposit` object representing the ATM transaction the user selected. We use a `Transaction` variable here to allow us to take advantage of polymorphism. Also, we name this variable after the *role name* included in the class diagram of Fig. 12.7—`currentTransaction`. Line 80 declares another local variable—a boolean called `userExited` that keeps track of whether the user has chosen to exit. This variable controls a `while` loop (lines 83–111) that allows the user to execute an unlimited number of transactions before choosing to exit. Within this loop, line 86 displays the main menu and obtains the user's menu selection by calling an ATM utility method `displayMainMenu` (declared in lines 115–124). This method displays the main menu by invoking methods of the ATM's screen and returns a menu selection obtained from the user through the ATM's keypad. Line 86 stores the user's selection returned by `displayMainMenu` in local variable `mainMenuSelection`.

After obtaining a main menu selection, method `performTransactions` uses a switch statement (lines 89–110) to respond to the selection appropriately. If `mainMenuSelection`
is equal to any of the three integer constants representing transaction types (i.e., if the user chose to perform a transaction), lines 97–98 call utility method createTransaction (declared in lines 127–149) to return a newly instantiated object of the type that corresponds to the selected transaction. Variable currentTransaction is assigned the reference returned by createTransaction, then line 100 invokes method execute of this transaction to execute it. We'll discuss Transaction method execute and the three Transaction subclasses shortly. We assign the Transaction variable currentTransaction an object of one of the three Transaction subclasses so that we can execute transactions polymorphically. For example, if the user chooses to perform a balance inquiry, mainMenuSelection equals BALANCE_INQUIRY, leading createTransaction to return a BalanceInquiry object. Thus, currentTransaction refers to a BalanceInquiry, and invoking currentTransaction.execute() results in BalanceInquiry's version of execute being called.

Creating a Transaction
Method createTransaction (lines 127–149) uses a switch statement (lines 132–146) to instantiate a new Transaction subclass object of the type indicated by the parameter type. Recall that method performTransactions passes mainMenuSelection to this method only when mainMenuSelection contains a value corresponding to one of the three transaction types. Therefore type is BALANCE_INQUIRY, WITHDRAWAL or DEPOSIT. Each case in the switch statement instantiates a new object by calling the appropriate Transaction subclass constructor. Each constructor has a unique parameter list, based on the specific data required to initialize the subclass object. A BalanceInquiry requires only the account number of the current user and references to the ATM's screen and the bankDatabase. In addition to these parameters, a Withdrawal requires references to the ATM's keypad and cashDispenser, and a Deposit requires references to the ATM's keypad and depositSlot. We discuss the transaction classes in more detail in Sections 13.4.8–13.4.11.

Exiting the Main Menu and Processing Invalid Selections
After executing a transaction (line 100 in performTransactions), userExited remains false and lines 83–111 repeat, returning the user to the main menu. However, if a user does not perform a transaction and instead selects the main menu option to exit, line 104 sets userExited to true, causing the condition of the while loop (userExited) to become false. This while is the final statement of method performTransactions, so control returns to the calling method run. If the user enters an invalid main menu selection (i.e., not an integer from 1–4), lines 107–108 display an appropriate error message, userExited remains false and the user returns to the main menu to try again.

Awaiting the Next ATM User
When performTransactions returns control to method run, the user has chosen to exit the system, so lines 46–47 reset the ATM's attributes userAuthenticated and currentAccountNumber to prepare for the next ATM user. Line 48 displays a goodbye message before the ATM starts over and welcomes the next user.

13.4.2 Class Screen
Class Screen (Fig. 13.14) represents the screen of the ATM and encapsulates all aspects of displaying output to the user. Class Screen approximates a real ATM's screen with a computer monitor and outputs text messages using standard console output methods
13.4 ATM Case Study Implementation

System.out.print, System.out.println and System.out.printf. In this case study, we

designed class Screen to have one operation—displayMessage. For greater flexibility in
displaying messages to the Screen, we now declare three Screen methods—displayMes-
sage, displayMessageLine and displayDollarAmount.

```java
public class Screen {

    // display a message without a carriage return
    public void displayMessage(String message) {
        System.out.print( message );
    }

    // display a message with a carriage return
    public void displayMessageLine(String message) {
        System.out.println( message );
    }

    // displays a dollar amount
    public void displayDollarAmount(double amount) {
        System.out.printf("$%.2f", amount);
    }
}
```

**Fig. 13.14** | Class Screen represents the screen of the ATM.

Method `displayMessage` (lines 7–10) takes a `String` argument and prints it to the
console. The cursor stays on the same line, making this method appropriate for displaying
prompts to the user. Method `displayMessageLine` (lines 13–16) does the same using
`System.out.println`, which outputs a newline to move the cursor to the next line.
Finally, method `displayDollarAmount` (lines 19–22) outputs a properly formatted dollar
amount (e.g., $1,234.56). Line 21 uses `System.out.printf` to output a `double` value for-
matted with commas to increase readability and two decimal places.

### 13.4.3 Class Keypad

Class `Keypad` (Fig. 13.15) represents the keypad of the ATM and is responsible for receiv-
ing all user input. Recall that we’re simulating this hardware, so we use the computer’s key-
board to approximate the keypad. We use class `Scanner` to obtain console input from the
user. A computer keyboard contains many keys not found on the ATM’s keypad. Howev-
er, we assume that the user presses only the keys on the computer keyboard that also appear
on the keypad—the keys numbered 0–9 and the `Enter` key.

Line 3 of class `Keypad` imports class `Scanner` for use in class `Keypad`. Line 7 declares
`Scanner` variable `input` as an instance variable. Line 12 in the constructor creates a new
`Scanner` object that reads input from the standard input stream (`System.in`) and assigns
the object’s reference to variable `input`. Method `getInput` (lines 16–19) invokes `Scanner`
public class Keypad
{
    private Scanner input; // reads data from the command line
    // no-argument constructor initializes the Scanner
    public Keypad()
    {
        input = new Scanner( System.in );
    } // end no-argument Keypad constructor
    // return an integer value entered by user
    public int getInput()
    {
        return input.nextInt(); // we assume that user enters an integer
    } // end method getInput
} // end class Keypad

Fig. 13.15 | Class Keypad represents the ATM’s keypad.

method `nextInt` (line 18) to return the next integer input by the user. [Note: Method
nextInt can throw an InputMismatchException if the user enters non-integer input.
Because the real ATM’s keypad permits only integer input, we assume that no exception
will occur and do not attempt to fix this problem. See Chapter 11, Exception Handling: A
Deeper Look, for information on catching exceptions.] Recall that `nextInt` obtains all the
input used by the ATM. Keypad’s `getInput` method simply returns the integer input by the
user. If a client of class Keypad requires input that satisfies some criteria (i.e., a number cor-
responding to a valid menu option), the client must perform the error checking.

13.4.4 Class CashDispenser

Class CashDispenser (Fig. 13.16) represents the cash dispenser of the ATM. Line 7 de-
clares constant INITIAL_COUNT, which indicates the initial count of bills in the cash dis-
enser when the ATM starts (i.e., 500). Line 8 implements attribute `count` (modeled in
Fig. 13.10), which keeps track of the number of bills remaining in the CashDispenser at
any time. The constructor (lines 11–14) sets count to the initial count. CashDispenser
has two public methods—`dispenseCash` (lines 17–21) and `isSufficientCashAvailable`
(lines 24–32). The class trusts that a client (i.e., Withdrawal) calls `dispenseCash` only af-
ter establishing that sufficient cash is available by calling `isSufficientCashAvailable`. Thus, `dispenseCash` simply simulates dispensing the requested amount without checking whether sufficient cash is available.

// CashDispenser.java
// Represents the cash dispenser of the ATM

Fig. 13.16 | Class CashDispenser represents the ATM’s cash dispenser. (Part 1 of 2.)
public class CashDispenser
{
    // the default initial number of bills in the cash dispenser
    private final static int INITIAL_COUNT = 500;
    private int count; // number of $20 bills remaining

    // no-argument CashDispenser constructor initializes count to default
    public CashDispenser()
    {
        count = INITIAL_COUNT; // set count attribute to default
    } // end CashDispenser constructor

    // simulates dispensing of specified amount of cash
    public void dispenseCash(int amount)
    {
        int billsRequired = amount / 20; // number of $20 bills required
        count -= billsRequired; // update the count of bills
    } // end method dispenseCash

    // indicates whether cash dispenser can dispense desired amount
    public boolean isSufficientCashAvailable(int amount)
    {
        int billsRequired = amount / 20; // number of $20 bills required
        if (count >= billsRequired)
            return true; // enough bills available
        else
            return false; // not enough bills available
    } // end method isSufficientCashAvailable

    } // end class CashDispenser

Fig. 13.16 | Class CashDispenser represents the ATM’s cash dispenser. (Part 2 of 2.)

Method isSufficientCashAvailable (lines 24–32) has a parameter amount that specifies the amount of cash in question. Line 26 calculates the number of $20 bills required to dispense the specified amount. The ATM allows the user to choose only withdrawal amounts that are multiples of $20, so we divide amount by 20 to obtain the number of billsRequired. Lines 28–31 return true if the CashDispenser’s count is greater than or equal to billsRequired (i.e., enough bills are available) and false otherwise (i.e., not enough bills). For example, if a user wishes to withdraw $80 (i.e., billsRequired is 4), but only three bills remain (i.e., count is 3), the method returns false.

Method dispenseCash (lines 17–21) simulates cash dispensing. If our system were hooked up to a real hardware cash dispenser, this method would interact with the device to physically dispense cash. Our version of the method simply decreases the count of bills remaining by the number required to dispense the specified amount (line 20). It’s the responsibility of the client of the class (i.e., withdrawal) to inform the user that cash has been dispensed—CashDispenser cannot interact directly with Screen.

13.4.5 Class DepositSlot

Class DepositSlot (Fig. 13.17) represents the ATM’s deposit slot. Like class CashDispenser, class DepositSlot merely simulates the functionality of a real hardware deposit
slot. DepositSlot has no attributes and only one method—isEnvelopeReceived (lines 8–11)—which indicates whether a deposit envelope was received.

```
1 // DepositSlot.java
2 // Represents the deposit slot of the ATM
3 4 public class DepositSlot
5 {
6     // indicates whether envelope was received (always returns true,
7     // because this is only a software simulation of a real deposit slot)
8     public boolean isEnvelopeReceived()
9     {
10        return true; // deposit envelope was received
11     } // end method isEnvelopeReceived
12 } // end class DepositSlot
```

Fig. 13.17 | Class DepositSlot represents the ATM’s deposit slot.

Recall from the requirements document that the ATM allows the user up to two minutes to insert an envelope. The current version of method isEnvelopeReceived simply returns true immediately (line 10), because this is only a software simulation, and we assume that the user has inserted an envelope within the required time frame. If an actual hardware deposit slot were connected to our system, method isEnvelopeReceived might be implemented to wait for a maximum of two minutes to receive a signal from the hardware deposit slot indicating that the user has indeed inserted a deposit envelope. If isEnvelopeReceived were to receive such a signal within two minutes, the method would return true. If two minutes elapsed and the method still had not received a signal, then the method would return false.

### 13.4.6 Class Account

Class Account (Fig. 13.18) represents a bank account. Each Account has four attributes (modeled in Fig. 13.10)—accountNumber, pin, availableBalance and totalBalance. Lines 6–9 implement these attributes as private fields. Variable availableBalance represents the amount of funds available for withdrawal. Variable totalBalance represents the amount of funds available, plus the amount of deposited funds still pending confirmation or clearance.

```
1 // Account.java
2 // Represents a bank account
3 4 public class Account
5 {
6     private int accountNumber; // account number
7     private int pin; // PIN for authentication
8     private double availableBalance; // funds available for withdrawal
9     private double totalBalance; // funds available + pending deposits
```

Fig. 13.18 | Class Account represents a bank account. (Part 1 of 2.)
10 // Account constructor initializes attributes
11 public Account( int theAccountNumber, int thePIN,
12 double theAvailableBalance, double theTotalBalance )
13 {
14     accountNumber = theAccountNumber;
15     pin = thePIN;
16     availableBalance = theAvailableBalance;
17     totalBalance = theTotalBalance;
18 } // end Account constructor
19
20 // determines whether a user-specified PIN matches PIN in Account
21 public boolean validatePIN( int userPIN )
22 {
23     if ( userPIN == pin )
24         return true;
25     else
26         return false;
27 } // end method validatePIN
28
29 // returns available balance
30 public double getAvailableBalance() {
31     return availableBalance;
32 } // end getAvailableBalance
33
34 // returns the total balance
35 public double getTotalBalance() {
36     return totalBalance;
37 } // end method getTotalBalance
38
39 // credits an amount to the account
40 public void credit( double amount ) {
41     totalBalance += amount; // add to total balance
42 } // end method credit
43
44 // debits an amount from the account
45 public void debit( double amount ) {
46     availableBalance -= amount; // subtract from available balance
47     totalBalance -= amount; // subtract from total balance
48 } // end method debit
49
50 // returns account number
51 public int getAccountNumber() {
52     return accountNumber;
53 } // end method getAccountNumber
54 } // end class Account

Fig. 13.18 | Class Account represents a bank account. (Part 2 of 2.)
The Account class has a constructor (lines 12–19) that takes an account number, the PIN established for the account, the account’s initial available balance and the account’s initial total balance as arguments. Lines 15–18 assign these values to the class’s attributes (i.e., fields).

Method validatePIN (lines 22–28) determines whether a user-specified PIN (i.e., parameter userPIN) matches the PIN associated with the account (i.e., attribute pin). Recall that we modeled this method’s parameter userPIN in Fig. 12.19. If the two PINs match, the method returns true (line 25); otherwise, it returns false (line 27).

Methods getAvailableBalance (lines 31–34) and getTotalBalance (lines 37–40) return the values of double attributes availableBalance and totalBalance, respectively.

Method credit (lines 43–46) adds an amount of money (i.e., parameter amount) to an Account as part of a deposit transaction. This method adds the amount only to attribute totalBalance (line 45). The money credited to an account during a deposit does not become available immediately, so we modify only the total balance. We assume that the bank updates the available balance appropriately at a later time. Our implementation of class Account includes only methods required for carrying out ATM transactions. Therefore, we omit the methods that some other bank system would invoke to add to attribute availableBalance (to confirm a deposit) or subtract from attribute totalBalance (to reject a deposit).

Method debit (lines 49–53) subtracts an amount of money (i.e., parameter amount) from an Account as part of a withdrawal transaction. This method subtracts the amount from both attribute availableBalance (line 51) and attribute totalBalance (line 52), because a withdrawal affects both measures of an account balance.

Method getAccountNumber (lines 56–59) provides access to an Account’s account-Number. We include this method in our implementation so that a client of the class (i.e., BankDatabase) can identify a particular Account. For example, BankDatabase contains many Account objects, and it can invoke this method on each of its Account objects to locate the one with a specific account number.

### 13.4.7 Class BankDatabase

Class BankDatabase (Fig. 13.19) models the bank’s database with which the ATM interacts to access and modify a user’s account information. We study database access in Chapter 28. For now we model the database as an array. An exercise in Chapter 28 asks you to reimplement this portion of the ATM using an actual database.

```java
1 // BankDatabase.java
2 // Represents the bank account information database
3
4 public class BankDatabase
5 {
6     private Account[] accounts; // array of Accounts
7     // no-argument BankDatabase constructor initializes accounts
8     public BankDatabase()
9     {
```

*Fig. 13.19* | Class BankDatabase represents the bank’s account information database. (Part 1 of 3.)
11 accounts = new Account[2]; // just 2 accounts for testing
12 accounts[0] = new Account(12345, 54321, 1000.0, 1200.0);
13 accounts[1] = new Account(98765, 65789, 200.0, 200.0);
14 } // end no-argument BankDatabase constructor
15
16 // retrieve Account object containing specified account number
17 private Account getAccount( int accountNumber )
18 {
19     // loop through accounts searching for matching account number
20     for ( Account currentAccount : accounts )
21     {
22         // return current account if match found
23         if ( currentAccount.get AccountNumber() == accountNumber )
24             return currentAccount;
25     } // end for
26     return null; // if no matching account was found, return null
27 } // end method getAccount
28
29 // determine whether user-specified account number and PIN match
30 // those of an account in the database
31 public boolean authenticateUser( int userAccountNumber, int userPIN )
32 {
33     Account userAccount = getAccount( userAccountNumber );
34     // if account exists, return result of Account method validatePIN
35     if ( userAccount != null )
36         return userAccount.validatePIN( userPIN );
37     else
38         return false; // account number not found, so return false
39 } // end method authenticateUser
40
41 // return available balance of Account with specified account number
42 public double getAvailableBalance( int userAccountNumber )
43 {
44     return getAccount( userAccountNumber ).getAvailableBalance();
45 } // end method getAvailableBalance
46
47 // return total balance of Account with specified account number
48 public double getTotalBalance( int userAccountNumber )
49 {
50     return getAccount( userAccountNumber ).getTotalBalance();
51 } // end method getTotalBalance
52
53 // credit an amount to Account with specified account number
54 public void credit( int userAccountNumber, double amount )
55 {
56     getAccount( userAccountNumber ).credit( amount );
57 } // end method credit
58

Fig. 13.19 | Class BankDatabase represents the bank's account information database. (Part 2 of 3.)
// debit an amount from Account with specified account number
public void debit( int userAccountNumber, double amount )
{
    getAccount( userAccountNumber ).debit( amount );
} // end method debit
} // end class BankDatabase

Fig. 13.19  |  Class BankDatabase represents the bank’s account information database. (Part 3 of 3.)
debit (lines 63–66) therefore simply retrieve the user's Account object with utility method getAccount, then invoke the appropriate Account method on that object. We know that the calls to getAccount from these methods will never return null, because userAccountNumber must refer to an existing Account. Methods getAvailableBalance and getTotalBalance return the values returned by the corresponding Account methods. Also, credit and debit simply redirect parameter amount to the Account methods they invoke.

### 13.4.8 Class Transaction

Class Transaction (Fig. 13.20) is an abstract superclass that represents the notion of an ATM transaction. It contains the common features of subclasses BalanceInquiry, Withdrawal, and Deposit. This class expands upon the "skeleton" code first developed in Section 13.3. Line 4 declares this class to be abstract. Lines 6–8 declare the class's private attributes. Recall from the class diagram of Fig. 13.10 that class Transaction contains an attribute accountNumber (line 6) that indicates the account involved in the Transaction. We derive attributes screen (line 7) and bankDatabase (line 8) from class Transaction's associations modeled in Fig. 13.9—all transactions require access to the ATM's screen and the bank's database.

```java
1 // Transaction.java
2 // Abstract superclass Transaction represents an ATM transaction
3
4 public abstract class Transaction
5 {
6     private int accountNumber; // indicates account involved
7     private Screen screen; // ATM's screen
8     private BankDatabase bankDatabase; // account info database
9
10 // Transaction constructor invoked by subclasses using super()
11 public Transaction( int userAccountNumber, Screen atmScreen,
12     BankDatabase atmBankDatabase )
13 {
14     accountNumber = userAccountNumber;
15     screen = atmScreen;
16     bankDatabase = atmBankDatabase;
17 } // end Transaction constructor
18
19 // return account number
20 public int getAccountNumber()
21 {
22     return accountNumber;
23 } // end method getAccountNumber
24
25 // return reference to screen
26 public Screen getScreen()
27 {
28     return screen;
29 } // end method getScreen
30
```

**Fig. 13.20** | Abstract superclass Transaction represents an ATM transaction. (Part 1 of 2.)
Chapter 13  ATM Case Study Part 2: Implementing an Object-Oriented Design

```java
31 // return reference to bank database
32 public BankDatabase getBankDatabase()
33 {
34    return bankDatabase;
35 } // end method getBankDatabase
36 // perform the transaction (overridden by each subclass)
37 abstract public void execute();
38 } // end class Transaction
```

**Fig. 13.20**  Abstract superclass Transaction represents an ATM transaction. (Part 2 of 2.)

Class Transaction has a constructor (lines 11–17) that takes as arguments the current user’s account number and references to the ATM’s screen and the bank’s database. Because Transaction is an abstract class, this constructor will be called only by the constructors of the Transaction subclasses.

The class has three public get methods—getAccountNumber (lines 20–23), getScreen (lines 26–29) and getBankDatabase (lines 32–35). These are inherited by Transaction subclasses and used to gain access to class Transaction’s private attributes.

Class Transaction also declares abstract method execute (line 38). It does not make sense to provide this method’s implementation, because a generic transaction cannot be executed. So, we declare this method abstract and force each Transaction subclass to provide a concrete implementation that executes that particular type of transaction.

### 13.4.9 Class BalanceInquiry

Class BalanceInquiry (Fig. 13.21) extends Transaction and represents a balance-inquiry ATM transaction. BalanceInquiry does not have any attributes of its own, but it inherits Transaction attributes accountNumber, screen and bankDatabase, which are accessible through Transaction’s public get methods. The BalanceInquiry constructor takes arguments corresponding to these attributes and simply forwards them to Transaction’s constructor using super (line 10).

```java
1 // BalanceInquiry.java
2 // Represents a balance inquiry ATM transaction
3
4 public class BalanceInquiry extends Transaction
5 {
6   // BalanceInquiry constructor
7   public BalanceInquiry( int userAccountNumber, Screen atmScreen,
8                        BankDatabase atmBankDatabase )
9   {
10     super( userAccountNumber, atmScreen, atmBankDatabase );
11   } // end BalanceInquiry constructor
12 // performs the transaction
13 @Override
14 public void execute()
15 {

```

**Fig. 13.21**  Class BalanceInquiry represents a balance-inquiry ATM transaction. (Part 1 of 2.)
13.4 ATM Case Study Implementation

```java
// get references to bank database and screen
BankDatabase bankDatabase = getBankDatabase();
Screen screen = getScreen();

// get the available balance for the account involved
double availableBalance =
    bankDatabase.getAvailableBalance(getAccountNumber());

// get the total balance for the account involved
double totalBalance =
    bankDatabase.getTotalBalance(getAccountNumber());

// display the balance information on the screen
screen.displayMessageLine("Balance Information: ");
screen.displayMessage("Account balance: ");
screen.displayDollarAmount(availableBalance);
screen.displayMessage("Total balance: ");
screen.displayDollarAmount(totalBalance);
```

Fig. 13.21 | Class BalanceInquiry represents a balance-inquiry ATM transaction. (Part 2 of 2.)

Class BalanceInquiry overrides Transaction's abstract method execute to provide a concrete implementation (lines 14–36) that performs the steps involved in a balance inquiry. Lines 18–19 get references to the bank database and the ATM's screen by invoking methods inherited from superclass Transaction. Lines 22–23 retrieve the available balance of the account involved by invoking method getAvailableBalance of bankDatabase. Line 23 uses inherited method getAccountNumber to get the account number of the current user, which it then passes to getAvailableBalance. Lines 26–27 retrieve the total balance of the current user's account. Lines 30–35 display the balance information on the ATM's screen. Recall that displayDollarAmount takes a double argument and outputs it to the screen formatted as a dollar amount. For example, if a user's availableBalance is 1000.5, line 32 outputs $1,000.50. Line 35 inserts a blank line of output to separate the balance information from subsequent output (i.e., the main menu repeated by class ATM after executing the BalanceInquiry).

13.4.10 Class Withdrawal

Class Withdrawal (Fig. 13.22) extends Transaction and represents a withdrawal ATM transaction. This class expands upon the “skeleton” code for this class developed in Fig. 13.12. Recall from the class diagram of Fig. 13.10 that class Withdrawal has one attribute, amount, which line 6 implements as an int field. Figure 13.9 models associations between class Withdrawal and classes Keypad and CashDispenser, for which lines 7–8 implement reference-type attributes keypad and cashDispenser, respectively. Line 11 declares a constant corresponding to the cancel menu option. We’ll soon discuss how the class uses this constant.
public class Withdrawal extends Transaction {
    private int amount; // amount to withdraw
    private Keypad keypad; // reference to keypad
    private CashDispenser cashDispenser; // reference to cash dispenser

    // constant corresponding to menu option to cancel
    private final static int CANCELED = 6;

    // Withdrawal constructor
    public Withdrawal(int userAccountNumber, Screen atmScreen, 
        BankDatabase atmBankDatabase, Keypad atmKeypad, 
        CashDispenser atmCashDispenser)
    {
        // initialize superclass variables
        super( userAccountNumber, atmScreen, atmBankDatabase );

        // initialize references to keypad and cash dispenser
        keypad = atmKeypad;
        cashDispenser = atmCashDispenser;
    } // end Withdrawal constructor

    // perform transaction
    @Override
    public void execute()
    {
        boolean cashDispensed = false; // cash was not dispensed yet
        double availableBalance; // amount available for withdrawal

        // get references to bank database and screen
        BankDatabase bankDatabase = getBankDatabase();
        Screen screen = getScreen();

        // loop until cash is dispensed or the user cancels
        do
        {
            // obtain a chosen withdrawal amount from the user
            amount = displayMenuOfAmounts();

            // check whether user chose a withdrawal amount or canceled
            if ( amount != CANCELED )
            {
                // get available balance of account involved
                availableBalance = bankDatabase.getAvailableBalance( getAccountNumber() );

                // check whether the user has enough money in the account
                if ( amount <= availableBalance )
                {

                } // end if
        } // end do
} // end execute

Fig. 13.22 | Class Withdrawal represents a withdrawal ATM transaction. (Part 1 of 3.)
// check whether the cash dispenser has enough money
if ( cashDispenser.isSufficientCashAvailable( amount ) )
{
    // update the account involved to reflect the withdrawal
    bankDatabase.debit( getAccountNumber(), amount );
    cashDispenser.dispenseCash( amount ); // dispense cash
    cashDispensed = true; // cash was dispensed
    // instruct user to take cash
    screen.displayMessageLine( "Your cash has been dispensed. Please take your cash now." );
} // end if
else // cash dispenser does not have enough cash
    screen.displayMessageLine( "Insufficient cash available in the ATM." +
                             "Please choose a smaller amount." );
} // end if
else // not enough money available in user's account
{
    screen.displayMessageLine( "Insufficient funds in your account." +
                             "Please choose a smaller amount." );
} // end else
} // end if
else // user chose cancel menu option
{
    screen.displayMessageLine( "Canceling transaction..." );
    return; // return to main menu because user canceled
} // end else
} // end method execute

// display a menu of withdrawal amounts and the option to cancel;
// return the chosen amount or 0 if the user chooses to cancel
private int displayMenuOfAmounts()
{
    int userChoice = 0; // local variable to store return value
    Screen screen = getScreen(); // get screen reference
    int[] amounts = { 0, 20, 40, 60, 100, 200 }; // array of amounts to correspond to menu numbers
    while ( userChoice == 0 )
    {
        // display the withdrawal menu
        screen.displayMessageLine( "Withdrawal Menu:" );
        screen.displayMessageLine( "1 - $20" );
        screen.displayMessageLine( "2 - $40" );
        screen.displayMessageLine( "3 - $60" );
    // loop while no valid choice has been made
    }

Fig. 13.22  Class Withdrawal represents a withdrawal ATM transaction. (Part 2 of 3.)
screen.displayMessageLine("4 - $100");
screen.displayMessageLine("5 - $200");
screen.displayMessageLine("6 - Cancel transaction");
screen.displayMessageLine("\nChoose a withdrawal amount: ");

int input = keypad.getInput(); // get user input through keypad

// determine how to proceed based on the input value
switch (input)
{
    case 1: // if the user chose a withdrawal amount
        userChoice = amounts[ input ]; // save user's choice
    break;
    case CANCELED: // the user chose to cancel
        userChoice = CANCELED; // save user's choice
    break;
    default: // the user did not enter a value from 1-6
        screen.displayMessageLine("\nInvalid selection. Try again. ");
}

return userChoice; // return withdrawal amount or CANCELED

} // end method displayMenuOfAmounts

} // end class Withdrawal

Fig. 13.22 | Class Withdrawal represents a withdrawal ATM transaction. (Part 3 of 3.)

Class Withdrawal's constructor (lines 14–24) has five parameters. It uses super to pass parameters userAccountNumber, atmScreen and atmBankDatabase to superclass Transaction's constructor to set the attributes that Withdrawal inherits from Transaction. The constructor also takes references atmKeypad and atmCashDispenser as parameters and assigns them to reference-type attributes keypad and cashDispenser.

Class Withdrawal overrides Transaction method execute with a concrete implementation (lines 27–85) that performs the steps of a withdrawal. Line 30 declares and initializes a local boolean variable cashDispensed, which indicates whether cash has been dispensed (i.e., whether the transaction has completed successfully) and is initially false. Line 31 declares local double variable availableBalance, which will store the user's available balance during a withdrawal transaction. Lines 34–35 get references to the bank database and the ATM's screen by invoking methods inherited from superclass Transaction.

Lines 38–83 contain a do...while that executes its body until cash is dispensed (i.e., until cashDispensed becomes true) or until the user chooses to cancel (in which case, the loop terminates). We use this loop to continuously return the user to the start of the transaction if an error occurs (i.e., the requested withdrawal amount is greater than the user's available balance or greater than the amount of cash in the cash dispenser). Line 41 displays a menu of withdrawal amounts and obtains a user selection by calling private utility method displayMenuOfAmounts (declared in lines 89–133). This method displays the
menu of amounts and returns either an int withdrawal amount or an int constant CANCELED to indicate that the user has chosen to cancel the transaction.

Method `displayMenuOfAmounts` (lines 89–133) first declares local variable `userChoice` (initially 0) to store the value that the method will return (line 91). Line 93 gets a reference to the screen by calling method `getScreen` inherited from superclass `Transaction`. Line 96 declares an integer array of withdrawal amounts that correspond to the amounts displayed in the withdrawal menu. We ignore the first element in the array (index 0) because the menu has no option 0. The while statement at lines 99–130 repeats until `userChoice` takes on a value other than 0. We’ll see shortly that this occurs when the user makes a valid selection from the menu. Lines 102–109 display the withdrawal menu on the screen and prompt the user to enter a choice. Line 111 obtains integer input through the keypad. The switch statement at lines 114–129 determines how to proceed based on the user’s input. If the user selects a number between 1 and 5, line 121 sets `userChoice` to the value of the element in amounts at index input. For example, if the user enters 3 to withdraw $60, line 121 sets `userChoice` to the value of amounts[3] (i.e., 60). Line 122 terminates the switch. Variable `userChoice` no longer equals 0, so the while at lines 99–130 terminates and line 132 returns `userChoice`. If the user selects the cancel menu option, lines 124–125 execute, setting `userChoice` to CANCELED and causing the method to return this value. If the user does not enter a valid menu selection, lines 127–128 display an error message and the user is returned to the withdrawal menu.

Line 44 in method `executeDetermines` whether the user has selected a withdrawal amount or chosen to cancel. If the user cancels, lines 80–81 execute and display an appropriate message to the user before returning control to the calling method (i.e., ATM method `performTransactions`). If the user has chosen a withdrawal amount, lines 47–48 retrieve the available balance of the current user’s Account and store it in variable `availableBalance`. Next, line 51 determines whether the selected amount is less than or equal to the user’s available balance. If it’s not, lines 73–75 display an appropriate error message. Control then continues to the end of the do while, and the loop repeats because `cashDispensed` is still false. If the user’s balance is high enough, the if statement at line 54 determines whether the cash dispenser has enough money to satisfy the withdrawal request by invoking the cashDispenser’s `isSufficientCashAvailable` method. If this method returns false, lines 67–69 display an appropriate error message and the do while repeats. If sufficient cash is available, then the requirements for the withdrawal are satisfied, and line 57 debits amount from the user’s account in the database. Lines 59–60 then instruct the cash dispenser to dispense the cash to the user and set `cashDispensed` to true. Finally, lines 63–64 display a message to the user that cash has been dispensed. Because `cashDispensed` is now true, control continues after the do while. No additional statements appear below the loop, so the method returns.

### 13.4.11 Class Deposit

Class `Deposit` (Fig. 13.23) extends `Transaction` and represents a deposit transaction. Recall from Fig. 13.10 that class `Deposit` has one attribute `amount`, which line 6 implements as an int field. Lines 7–8 create reference attributes `keypad` and `depositSlot` that implement the associations between class `Deposit` and classes `Keypad` and `DepositSlot` modeled in Fig. 13.9. Line 9 declares a constant CANCELED that corresponds to the value a user enters to cancel. We’ll soon discuss how the class uses this constant.
```java
public class Deposit extends Transaction {
    private double amount; // amount to deposit
    private Keypad keypad; // reference to keypad
    private DepositSlot depositSlot; // reference to deposit slot
    private final static int CANCELED = 0; // constant for cancel option

    // Deposit constructor
    public Deposit( int userAccountNumber, Screen atmScreen,
            BankDatabase atmBankDatabase, Keypad atmKeypad,
            DepositSlot atmDepositSlot )
    {
        // initialize superclass variables
        super( userAccountNumber, atmScreen, atmBankDatabase );
        // initialize references to keypad and deposit slot
        keypad = atmKeypad;
        depositSlot = atmDepositSlot;
    } // end Deposit constructor

    // perform transaction
    @Override
    public void execute()
    {
        BankDatabase bankDatabase = getBankDatabase(); // get reference
        Screen screen = getScreen(); // get reference
        amount = promptForDepositAmount(); // get deposit amount from user
        // check whether user entered a deposit amount or canceled
        if ( amount != CANCELED )
        {
            // request deposit envelope containing specified amount
            screen.displayMessage("Please insert a deposit envelope containing ");
            screen.displayDollarAmount( amount );
            screen.displayMessageLine(".");

            // receive deposit envelope
            boolean envelopeReceived = depositSlot.isEnvelopeReceived();
            if ( envelopeReceived )
            {
                screen.displayMessageLine("Your envelope has been received.
                NOTE: The money just deposited will not be available until we verify the amount of any enclosed cash and your checks clear.");
            }
        }
    }
}
```

**Fig. 13.23** Class Deposit represents a deposit ATM transaction. (Part 1 of 2.)
13.4 ATM Case Study Implementation  545

// credit account to reflect the deposit
bankDatabase.credit( getAccountNumber(), amount );
} // end if
else // deposit envelope not received
{
    screen.displayMessageLine( "You did not insert an " +
        "envelope, so the ATM has canceled your transaction." );
} // end else
} // end if
else // user canceled instead of entering amount
{
    screen.displayMessageLine( "Canceling transaction..." );
} // end else
} // end method execute

// prompt user to enter a deposit amount in cents
private double promptForDepositAmount()
{
    Screen screen = getScreen(); // get reference to screen
    // display the prompt
    screen.displayMessage( "Please enter a deposit amount in " +
        "Cents (or 0 to cancel): " );
    int input = keypad.getInput(); // receive input of deposit amount
    // check whether the user canceled or entered a valid amount
    if ( input == CANCELED )
        return CANCELED;
    else
    {
        return ( double ) input / 100; // return dollar amount
    } // end else
} // end method promptForDepositAmount
} // end class Deposit

Fig. 13.23 | Class Deposit represents a deposit ATM transaction. (Part 2 of 2.)

Like Withdrawal, class Deposit contains a constructor (lines 12–22) that passes three
parameters to superclass Transaction’s constructor. The constructor also has parameters
atmKeypad and atmDepositSlot, which it assigns to corresponding attributes (lines 20–21).

Method execute (lines 25–66) overrides the abstract version in superclass Transaction
with a concrete implementation that performs the steps required in a deposit trans-
action. Lines 28–29 get references to the database and the screen. Line 31 prompts the user
to enter a deposit amount by invoking private utility method promptForDepositAmount
(declared in lines 69–85) and sets attribute amount to the value returned. Method prompt-
ForDepositAmount asks the user to enter a deposit amount as an integer number of cents
(because the ATM’s keypad does not contain a decimal point; this is consistent with many
real ATMs) and returns the double value representing the dollar amount to be deposited.

Line 71 in method promptForDepositAmount gets a reference to the ATM’s screen.
Lines 74–75 display a message asking the user to input a deposit amount as a number
of cents or “0” to cancel the transaction. Line 76 receives the user’s input from the keypad.
Lines 79–84 determine whether the user has entered a real deposit amount or chosen to
cancel. If the latter, line 80 returns the constant CANCELED. Otherwise, line 83 returns the deposit amount after converting from the number of cents to a dollar amount by casting input to a double, then dividing by 100. For example, if the user enters 125 as the number of cents, line 83 returns 125.0 divided by 100, or 1.25—125 cents is $1.25.

Lines 34–65 in method execute determine whether the user has chosen to cancel the transaction instead of entering a deposit amount. If the user cancels, line 64 displays an appropriate message, and the method returns. If the user enters a deposit amount, lines 37–40 instruct the user to insert a deposit envelope with the correct amount. Recall that Screen method displayDollarAmount outputs a double formatted as a dollar amount.

Line 43 sets a local boolean variable to the value returned by depositSlot’s isEnvelopeReceived method, indicating whether a deposit envelope has been received. Recall that we coded method isEnvelopeReceived (lines 8–11 of Fig. 13.17) to always return true, because we’re simulating the functionality of the deposit slot and assume that the user always inserts an envelope. However, we code method execute of class Deposit to test for the possibility that the user does not insert an envelope—good software engineering demands that programs account for all possible return values. Thus, class Deposit is prepared for future versions of isEnvelopeReceived that could return false. Lines 48–54 execute if the deposit slot receives an envelope. Lines 48–51 display an appropriate message to the user. Line 54 then credits the deposit amount to the user’s account in the database. Lines 58–59 will execute if the deposit slot does not receive a deposit envelope. In this case, we display a message to the user stating that the ATM has canceled the transaction. The method then returns without modifying the user’s account.

13.4.12 Class ATMCaseStudy

Class ATMCaseStudy (Fig. 13.24) is a simple class that allows us to start, or “turn on,” the ATM and test the implementation of our ATM system model. Class ATMCaseStudy’s main method (lines 7–11) does nothing more than instantiate a new ATM object named theATM (line 9) and invoke its run method (line 10) to start the ATM.

```java
1 // ATMCaseStudy.java
2 // Driver program for the ATM case study
3
4 public class ATMCaseStudy
5 {
6     // main method creates and runs the ATM
7     public static void main( String[] args )
8     {
9         ATM theATM = new ATM();
10         theATM.run();
11     } // end main
12 } // end class ATMCaseStudy
```

Fig. 13.24 ATMCaseStudy.java starts the ATM.

13.5 Wrap-Up

In this chapter, you used inheritance to tune the design of the ATM software system, and you fully implemented the ATM in Java. Congratulations on completing the entire ATM
Answers to Self-Review Exercises

13.1 True. The minus sign (-) indicates private visibility.

13.2 b.

13.3 The design for class Keypad yields the code in Fig. 13.25. Recall that class Keypad has no attributes for the moment, but attributes may become apparent as we continue the implementation. Also, if we were designing a real ATM, method getInput would need to interact with the ATM’s keypad hardware. We’ll actually read input from the keyboard of a personal computer when we write the complete Java code in Section 13.4.

```java
1 // Class Keypad represents an ATM’s keypad
2 public class Keypad
3 {
4     // no attributes have been specified yet
5     // no-argument constructor
6     public Keypad()
7     {
8     } // end no-argument Keypad constructor
9     // operations
10    public int getInput()
11    {
12    } // end method getInput
13 } // end class Keypad
```

Fig. 13.25 | Java code for class Keypad based on Figs. 13.1–13.2.

13.4 b.

13.5 False. The UML requires that we italicize abstract class names and method names.

13.6 The design for class Transaction yields the code in Fig. 13.26. The bodies of the class constructor and methods are completed in Section 13.4. When fully implemented, methods getScreen and getBankDatabase will return superclass Transaction’s private reference attributes screen and bankDatabase, respectively. These methods allow the Transaction subclasses to access the ATM’s screen and interact with the bank’s database.

```java
1 // Abstract class Transaction represents an ATM transaction
2 public abstract class Transaction
3 {
4     // attributes
5     private int accountNumber; // indicates account involved
6     private Screen screen; // ATM’s screen
7     private BankDatabase bankDatabase; // account info database
```

Fig. 13.26 | Java code for class Transaction based on Figs. 13.9 and 13.10. (Part 1 of 2.)
8 9 // no-argument constructor invoked by subclasses using super()
10 public Transaction()
11 {
12 } // end no-argument Transaction constructor
13
14 // return account number
15 public int getAccountNumber()
16 {
17 } // end method getAccountNumber
18
19 // return reference to screen
20 public Screen getScreen()
21 {
22 } // end method getScreen
23
24 // return reference to bank database
25 public BankDatabase getBankDatabase()
26 {
27 } // end method getBankDatabase
28
29 // abstract method overridden by subclasses
30 public abstract void execute();
31 } // end class Transaction

**Fig. 13.26**  |  Java code for class Transaction based on Figs. 13.9 and 13.10. (Part 2 of 2.)