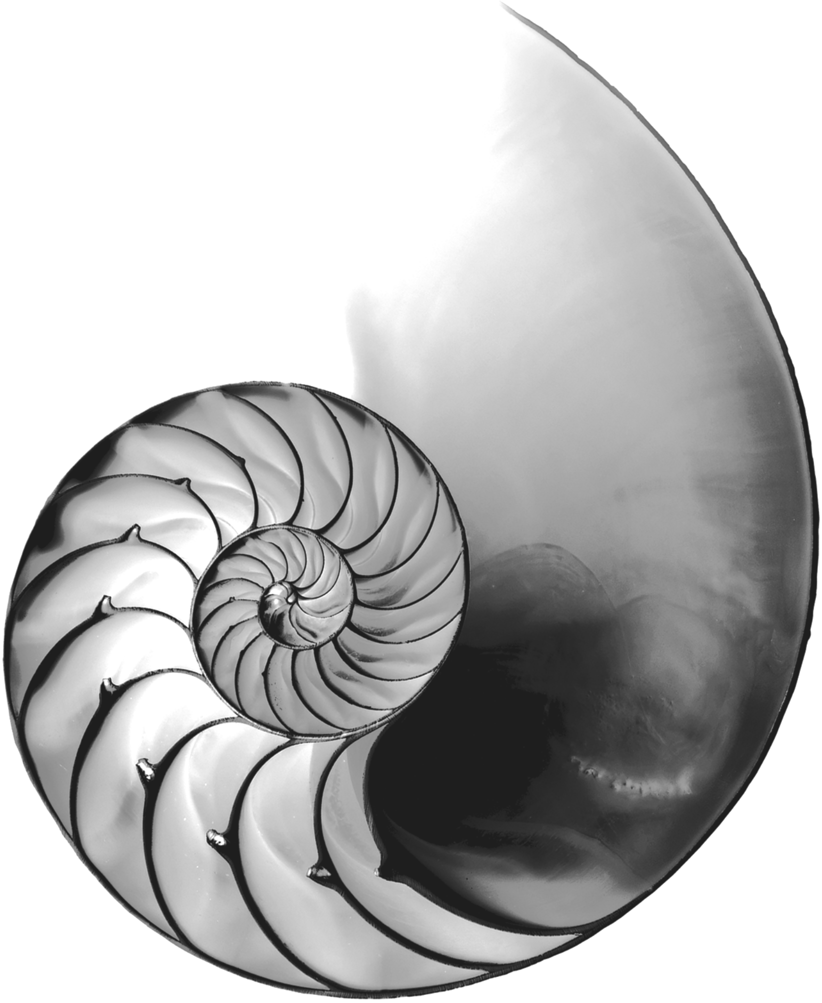
ATM Case Study, Part 1: Object-Oriented Design with the UML

33



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.

Outline

33.1 Case Study Introduction

Now we begin the *optional* portion of our object-oriented design and implementation case study. In this chapter and Chapter 34, 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 33.2–33.7 and 34.2–34.3, 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 34, you'll tune the design with inheritance, then fully implement the ATM as a Java application (Section 34.4). 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.

These chapters can be studied as a continuous unit after you've completed the introduction to object-oriented programming in Chapters 8–11.

33.2 Examining the Requirements Document

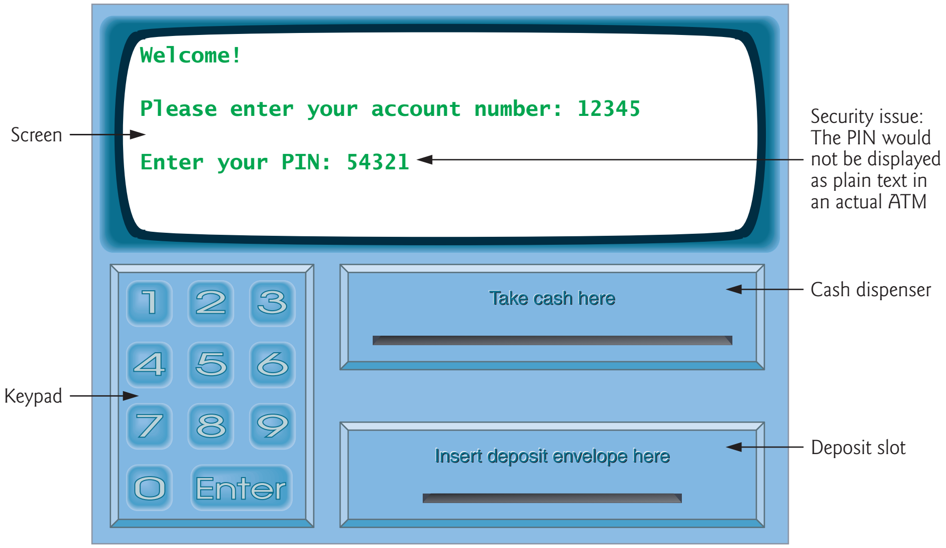
We begin our design process by presenting a **requirements document** that specifies the purpose of the ATM system and *what* it must do. Throughout the case study, we refer often to this requirements document.

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. 33.1). 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.]



```
Welcome!

Please enter your account number: 12345

Enter your PIN: 54321
```

Fig. 33.1 | 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; database access was presented in Chapter 24). 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 this case study. 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. 33.1):

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. 33.2). 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.

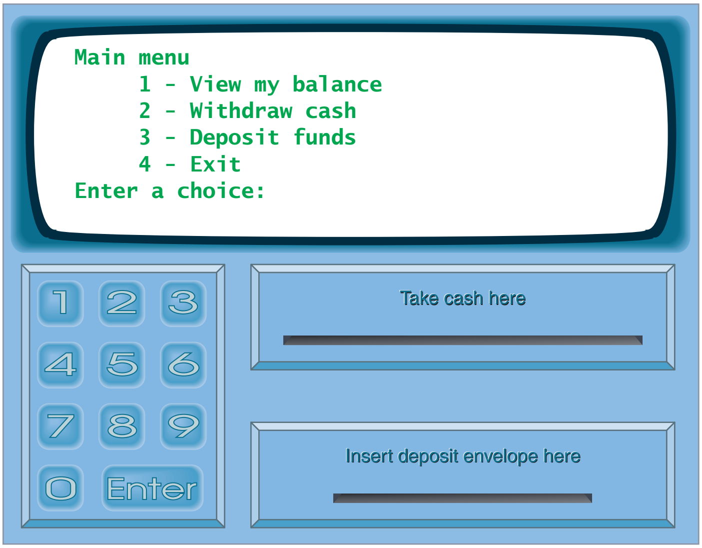

```
Main menu
     1 - View my balance
     2 - Withdraw cash
     3 - Deposit funds
     4 - Exit
Enter a choice:
```

Fig. 33.2 | ATM main menu.

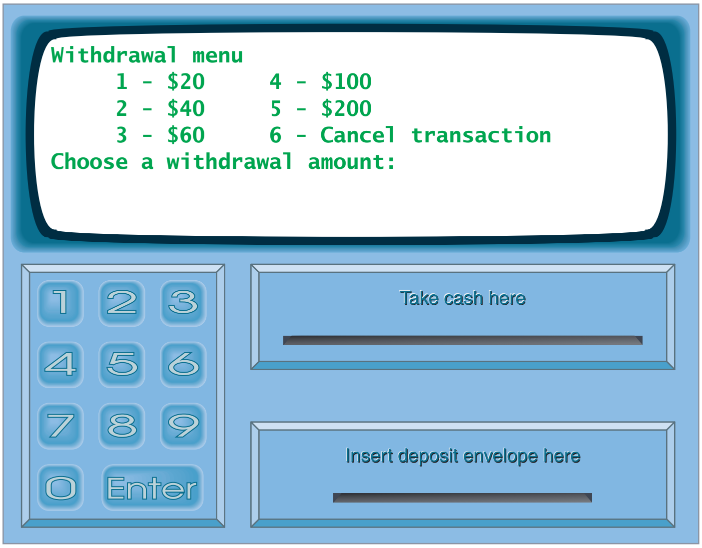
After the ATM authenticates the user, the main menu (Fig. 33.2) should contain a numbered option for each of the three types of transactions: balance inquiry (option 1), withdrawal (option 2) and deposit (option 3). It also should contain an option to allow the user to exit the system (option 4). The user then chooses either to perform a transaction (by entering 1, 2 or 3) or to exit the system (by entering 4).

---

[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.

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. 33.3) 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
     1 - $20      4 - $100
     2 - $40      5 - $200
     3 - $60      6 - Cancel transaction
Choose a withdrawal amount:
```

Fig. 33.3 | 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. 33.2) 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 33.4 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 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 33.4 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

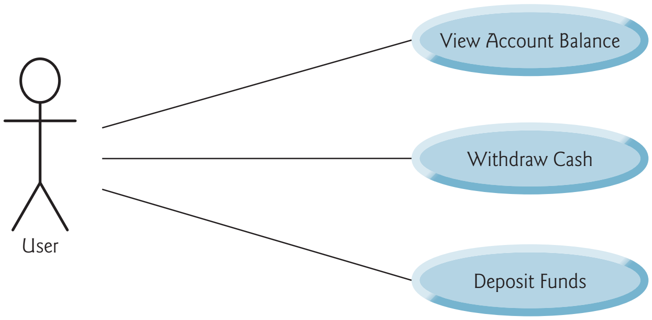


Fig. 33.4 | Use case diagram for the ATM system from the User's perspective.

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.

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. 33.1), 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.

1. **Use case diagrams**, such as the one in Fig. 33.4, 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 33.3, 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 33.5, 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 33.5, 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 33.7 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 33.7 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 33.3, 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/`.

Self-Review Exercises for Section 33.2

33.1 Suppose we enabled a user of our ATM system to transfer money between two bank accounts. Modify the use case diagram of Fig. 33.4 to reflect this change.

33.2 ________ 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

33.3 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

33.3 Identifying the Classes in a Requirements Document

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 33.2 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 33.5 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.

Nouns and noun phrases in the ATM requirements document			
bank	money / funds	account number	ATM
screen	PIN	user	keypad
bank database	customer	cash dispenser	balance inquiry
transaction	$20 bill / cash	withdrawal	account
deposit slot	deposit	balance	deposit envelope

Fig. 33.5 | 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 represent 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. 33.4.

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 33.4.) 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 33.6.

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 34.3, 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. 33.5. 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 33.6 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 33.4–33.5.) The bottom compartment contains the class's operations (discussed in Section 33.6). In Fig. 33.6, the middle and bottom compartments are empty because we've not yet determined this class's attributes and operations.

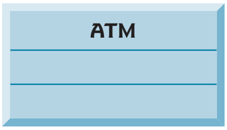


Fig. 33.6 | Representing a class in the UML using a class diagram.

Class diagrams also show the relationships between the classes of the system. Figure 33.7 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 33.4–33.6.

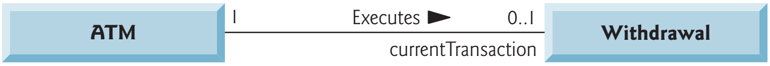


Fig. 33.7 | Class diagram showing an association among classes.

In Fig. 33.7, 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 33.8 lists and explains the multiplicity types.

An association can be named. For example, the word `Executes` above the line connecting classes `ATM` and `Withdrawal` in Fig. 33.7 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

Symbol	Meaning
0	None
1	One
m	An integer value
0..1	Zero or one
m, n	m or n
$m..n$	At least m, but not more than n
*	Any nonnegative integer (zero or more)
0..*	Zero or more (identical to *)
1..*	One or more

Fig. 33.8 | Multiplicity types.

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. 33.7 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. 33.7, 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. 33.9, 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. 33.9 indicate that an object of class `ATM` is formed from one object of class `Screen`, one object of class `CashDispenser`, one object of class `Keypad` and one object of class `DepositSlot`. The ATM *has a* screen, a keypad, a cash dispenser and a deposit slot. (As we saw in Chapter 9, the *is-a* relationship defines inheritance. We'll see in Section 34.3 that there's a nice opportunity to use inheritance in the ATM system design.)

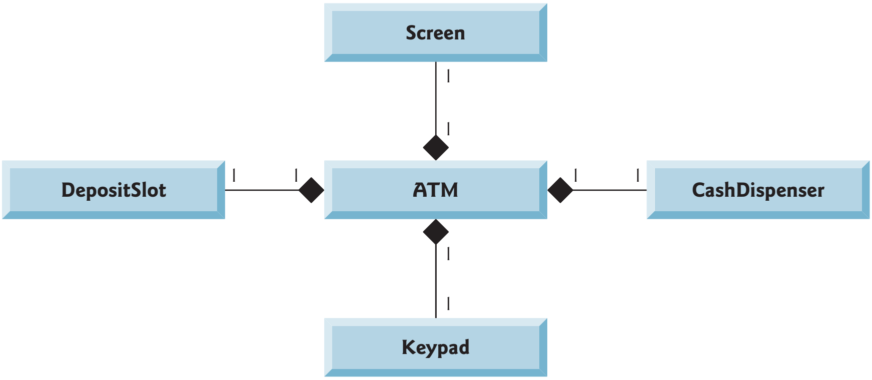


Fig. 33.9 | Class diagram showing composition relationships.

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

1. Only one class in the relationship can represent the *whole* (i.e., the diamond can be placed on only *one* end of the association line). For example, either the screen is part of the ATM or the ATM is part of the screen, but the screen and the ATM cannot both represent the whole in the relationship.
2. The *parts* in the composition relationship exist only as long as the whole 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 *part* may belong to only 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 *has-a* relationship does not satisfy one or more of these criteria, the UML specifies that **hollow diamonds** be attached to the ends of association lines to indicate **aggregation**—a weaker form of composition. For example, a personal computer and a computer monitor participate in an aggregation relationship—the computer *has a* monitor, but the two parts can exist independently, and the same monitor can be attached to multiple computers at once, thus violating composition's second and third properties.

Figure 33.10 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 `BalanceInquiry` and `Deposit` participate in associations similar to those of class `Withdrawal`, so we've chosen to omit them from this diagram to keep it simple. In Section 34.3, we expand our class diagram to include all the classes in the ATM system.

Figure 33.10 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. 33.7 or Fig. 33.9. It shows that class `ATM` has a **one-to-one relationship** with class `BankDatabase`—one `ATM` object *authenticates users against* one `BankDatabase` object. In

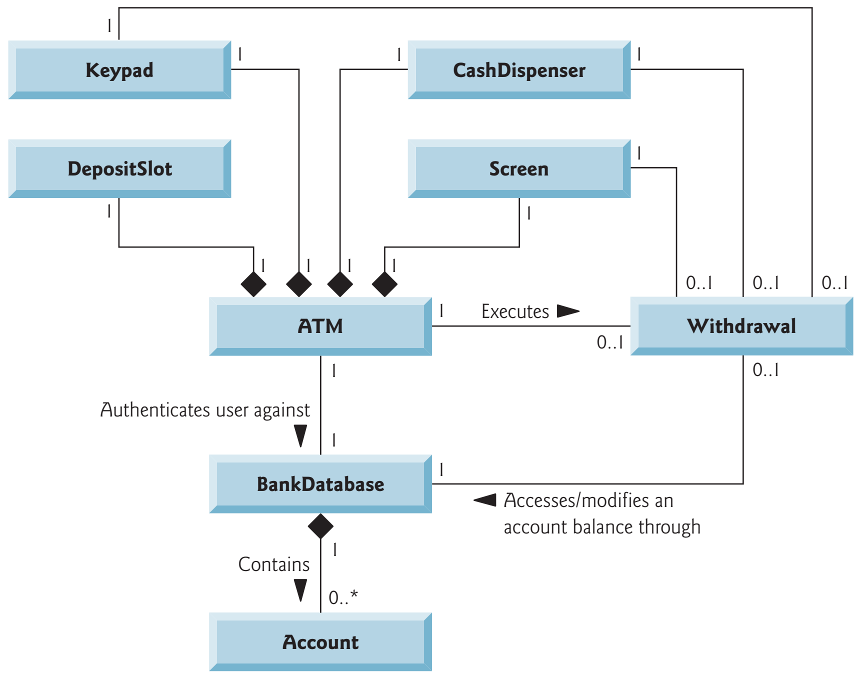


Fig. 33.10 | Class diagram for the ATM system model.

Fig. 33.10, 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. 33.8 that the multiplicity value * is identical to 0..*. We include 0..* in our class diagrams for clarity.

Figure 33.10 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 information (e.g., an account balance).

The class diagram in Fig. 33.10 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. 33.10, 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 33.4 we determine the attributes for each of these classes, and in Section 33.5 we use these attributes to examine how the system changes over time.

Self-Review Exercises for Section 33.3

33.4 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. 33.9) that models some of the composition relationships of class `Car`.

33.5 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`.

33.6 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.

33.7 Modify the class diagram of Fig. 33.10 to include class `Deposit` instead of class `Withdrawal`.

33.4 Identifying Class Attributes

[*Note:* **This section may be read 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 33.5 we examine how these attributes represent an object's state. In Section 33.6 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 attributes 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 33.3. We also create attributes to represent any additional data that a class may need, as such needs become clear throughout the design process.

Figure 33.11 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`.

Class	Descriptive words and phrases
`ATM`	user is authenticated
`BalanceInquiry`	account number
`Withdrawal`	account number amount
`Deposit`	account number amount
`BankDatabase`	*[no descriptive words or phrases]*
`Account`	account number PIN balance
`Screen`	*[no descriptive words or phrases]*
`Keypad`	*[no descriptive words or phrases]*
`CashDispenser`	begins each day loaded with 500 $20 bills
`DepositSlot`	*[no descriptive words or phrases]*

Fig. 33.11 | Descriptive words and phrases from the ATM requirements document.

Figure 33.11 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 33.5), 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. 33.12 lists some of the attributes for the classes in our system—the descriptive words and phrases in Fig. 33.11 lead us to identify these attributes. For simplicity, Fig. 33.12 does not show the associations among classes—we showed these in Fig. 33.10. This is a common practice of systems designers when designs are being developed. Recall from Section 33.3 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`:

```
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. 33.12—we discuss the reasoning behind this decision shortly. The attribute types in Fig. 33.12 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.

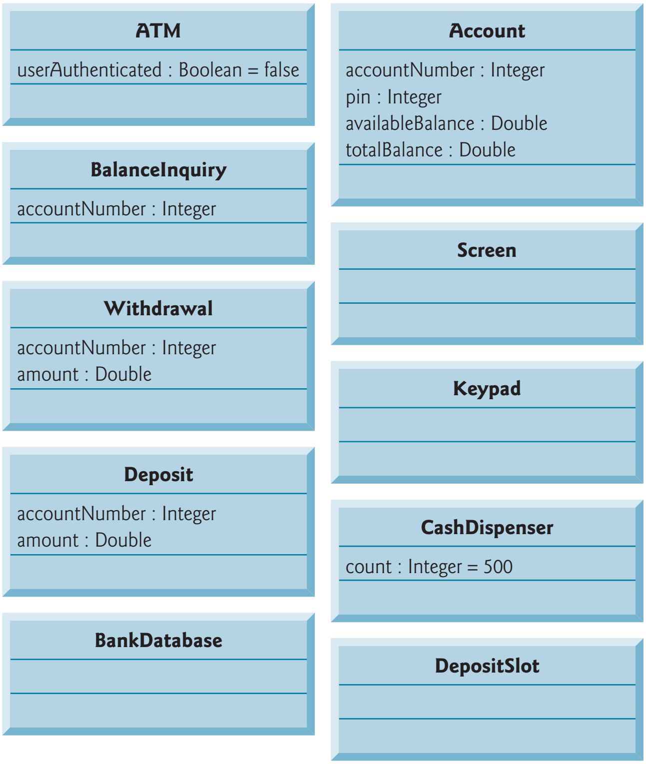


Fig. 33.12 | Classes with attributes.

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 33.12 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 33.1

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 33.12 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. 33.12 (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. 33.10 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. 33.12, because the compositions modeled in Fig. 33.10 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. 33.12 provides a solid basis for the structure of our model, but the diagram is not complete. In Section 33.5 we identify the states and activities of the objects in the model, and in Section 33.6 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 33.4

33.8 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.

33.9 Which of the following is *not* an attribute of an airplane?

a) length
b) wingspan
c) fly
d) number of seats

33.10 Describe the meaning of the following attribute declaration of class `CashDispenser` in the class diagram in Fig. 33.12:

```
count : Integer = 500
```

33.5 Identifying Objects' States and Activities

In Section 33.4, we identified many of the class attributes needed to implement the ATM system and added them to the class diagram in Fig. 33.12. 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 33.13 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. 33.12. This attribute is initialized to `false`, or the "User not authenticated" state, according to the state diagram.

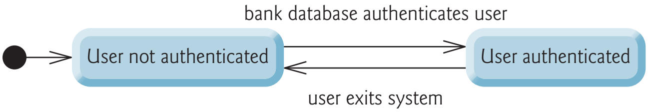


Fig. 33.13 | State diagram for the `ATM` object.

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 33.2

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. 33.14 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. 33.14 refer to the retrieval and display of *both* balance attributes.

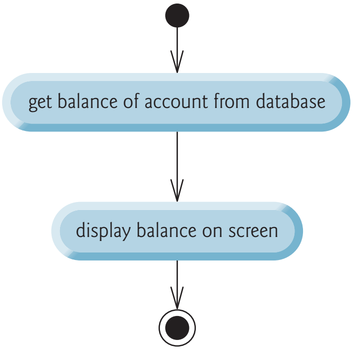


Fig. 33.14 | Activity diagram for a `BalanceInquiry` object.

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. 33.14) 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. 33.14) represents the *final state*—the end of the workflow 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 3–4.

Figure 33.15 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

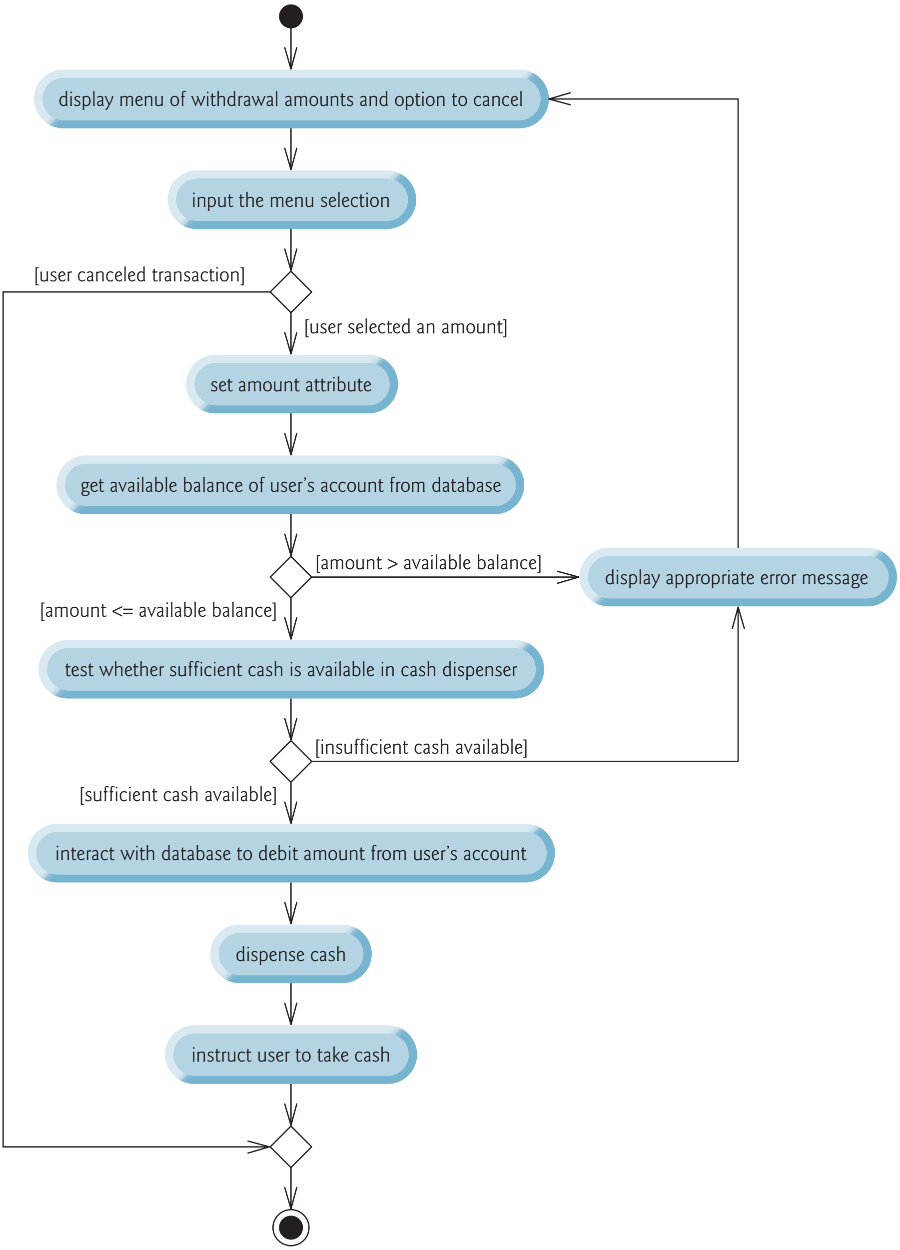


Fig. 33.15 | Activity diagram for a withdrawal transaction.

menu because these are not actions performed by a `Withdrawal` object. The transaction first displays a menu of standard withdrawal amounts (shown in Fig. 33.3) 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. 33.12) 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 33.6, 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 33.5

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

33.12 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

33.13 Based on the requirements document, create an activity diagram for a deposit transaction.

33.6 Identifying Class Operations

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. 33.16). The verb phrases in Fig. 33.16 help us determine the operations of each class.

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

Fig. 33.16 | 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. 33.16. 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. 33.17. 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:

```
operationName(parameter1, parameter2, …, parameterN) : return type
```

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

```
parameterName : parameterType
```

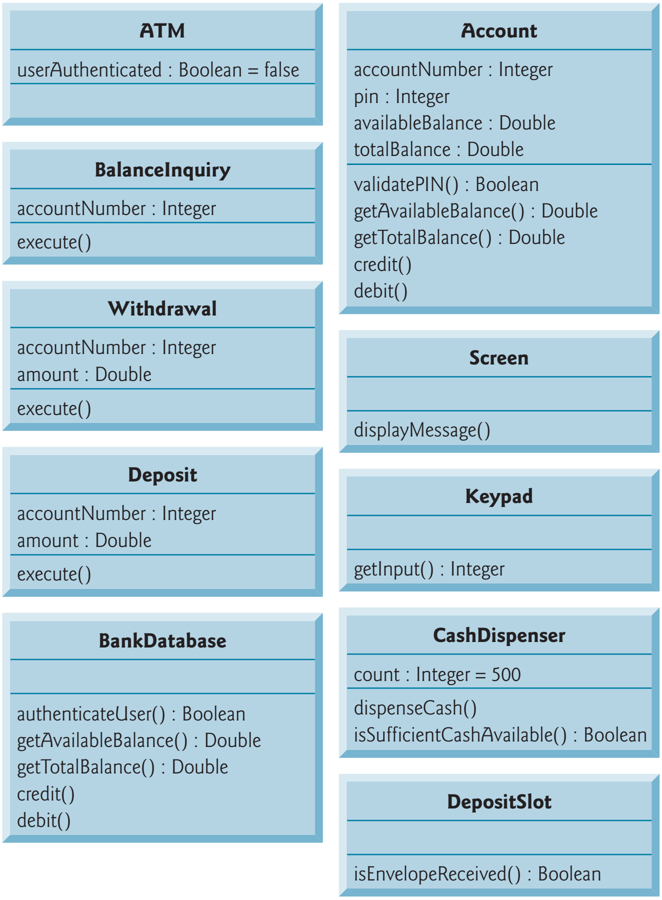


Fig. 33.17 | Classes in the ATM system with attributes and operations.

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 33.16 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. 33.17). 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 33.16 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 33.3 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 33.7, 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. 33.17). 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. 33.17). 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. 33.16, 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 nonnumeric input using keys on the computer keyboard that do not appear on the ATM's keypad.]

Dispensing Cash

Figure 33.16 lists "dispenses cash" for class `CashDispenser`. Therefore, we create operation `dispenseCash` and list it under class `CashDispenser` in Fig. 33.17. 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 33.16 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. 33.18 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.]

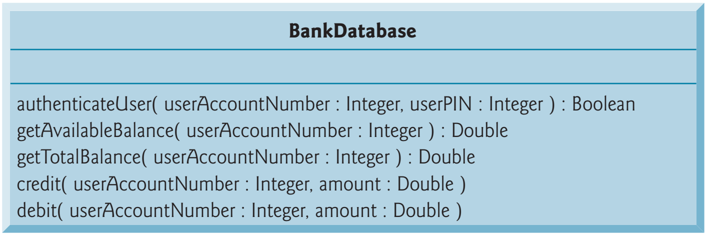


Fig. 33.18 | Class `BankDatabase` with operation parameters.

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 33.18 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. 33.18. 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 33.19 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 `Account`s, because these operations can be invoked only on a specific `Account` object.

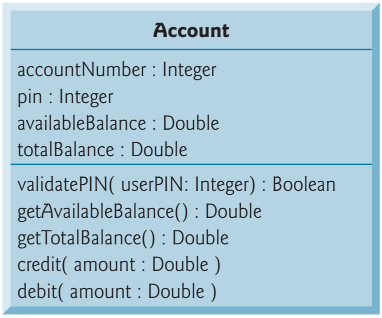


Fig. 33.19 | Class `Account` with operation parameters.

Identifying and Modeling Operation Parameters for Class `Screen`

Figure 33.20 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. 33.20 refers to the UML type. When we implement the system in Java, we'll use the Java class `String` to represent this parameter.

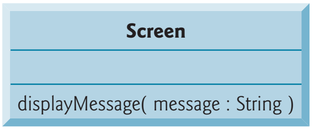


Fig. 33.20 | Class `Screen` with operation parameters.

Identifying and Modeling Operation Parameters for Class `CashDispenser`

Figure 33.21 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.

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 `isEnvelope-`

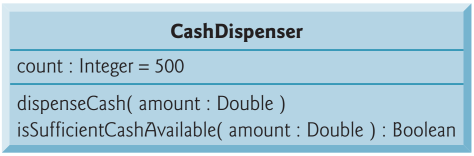


Fig. 33.21 | Class `CashDispenser` with operation parameters.

`Received` 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 33.6

33.14 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

33.15 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. 33.17?

33.16 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
```

33.7 Indicating Collaboration Among Objects

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 33.6, 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 33.2. 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 34, 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 33.6) 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 33.22 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. 33.22 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. 33.22. 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. 33.22. 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 33.22 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 collaborations, so we do not add anything to Fig. 33.22. 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`.

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

An object of class...	sends the message...	to an object of class...
`ATM`	`displayMessage`	`Screen`
	`getInput`	`Keypad`
	`authenticateUser`	`BankDatabase`
	`execute`	`BalanceInquiry`
	`execute`	`Withdrawal`
	`execute`	`Deposit`
`BalanceInquiry`	`getAvailableBalance`	`BankDatabase`
	`getTotalBalance`	`BankDatabase`
	`displayMessage`	`Screen`
`Withdrawal`	`displayMessage`	`Screen`
	`getInput`	`Keypad`
	`getAvailableBalance`	`BankDatabase`
	`isSufficientCashAvailable`	`CashDispenser`
	`debit`	`BankDatabase`
	`dispenseCash`	`CashDispenser`
`Deposit`	`displayMessage`	`Screen`
	`getInput`	`Keypad`
	`isEnvelopeReceived`	`DepositSlot`
	`credit`	`BankDatabase`
`BankDatabase`	`validatePIN`	`Account`
	`getAvailableBalance`	`Account`
	`getTotalBalance`	`Account`
	`debit`	`Account`
	`credit`	`Account`

Fig. 33.22 | Collaborations in the ATM system.

`Account`. Similarly, the `BalanceInquiry` retrieves the amount of money 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 33.23 shows a communication diagram that models the `ATM` executing a `BalanceInquiry`. 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).

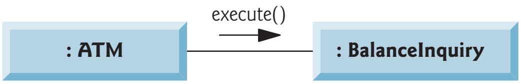


Fig. 33.23 | Communication diagram of the ATM executing a balance inquiry.

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. 33.23, 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 23.]

Sequence of Messages in a Communication Diagram

Figure 33.24 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. 33.24 begin after the `ATM` sends an `execute` message to a `BalanceInquiry` (i.e., the interaction modeled in Fig. 33.23). 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. 33.18 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.

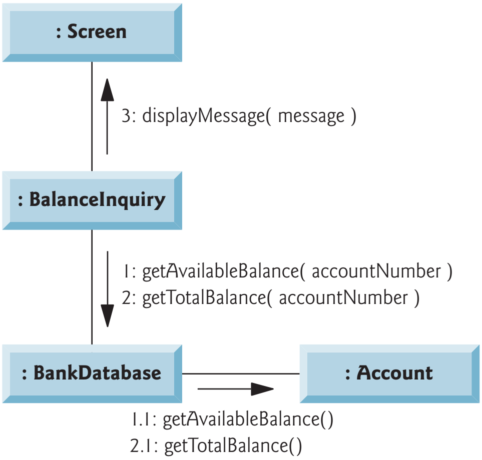


Fig. 33.24 | Communication diagram for executing a balance inquiry.

Figure 33.24 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. 33.24 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 33.25 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 (⇢) 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. 33.25 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. 33.15, so we do not show this logic in the sequence diagram of Fig. 33.25. 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

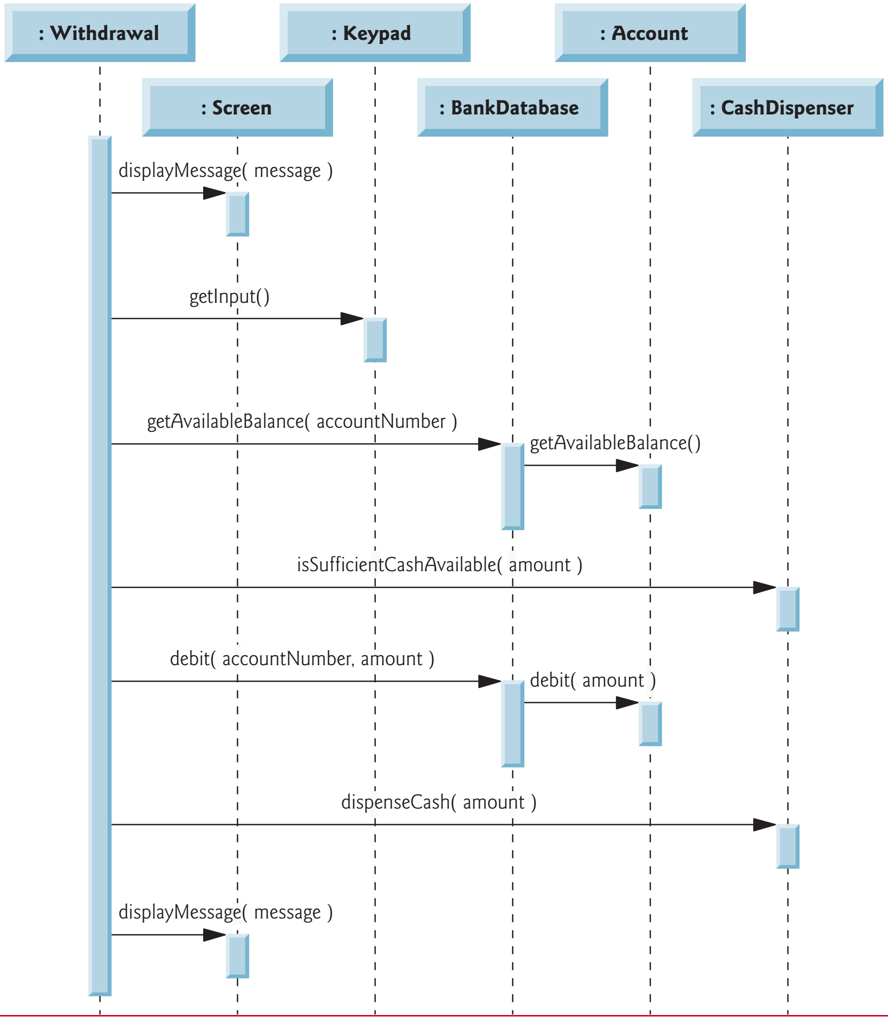


Fig. 33.25 | Sequence diagram that models a `Withdrawal` executing.

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 34.2, 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 33.7

33.17 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

33.18 Which form of interaction diagram emphasizes *what* collaborations occur? Which form emphasizes *when* collaborations occur?

33.19 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.

33.8 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 34, we tune the design using inheritance, then completely implement the design as a Java application.

Answers to Self-Review Exercises

33.1 Figure 33.26 contains a use case diagram for a modified version of our ATM system that also allows users to transfer money between accounts.

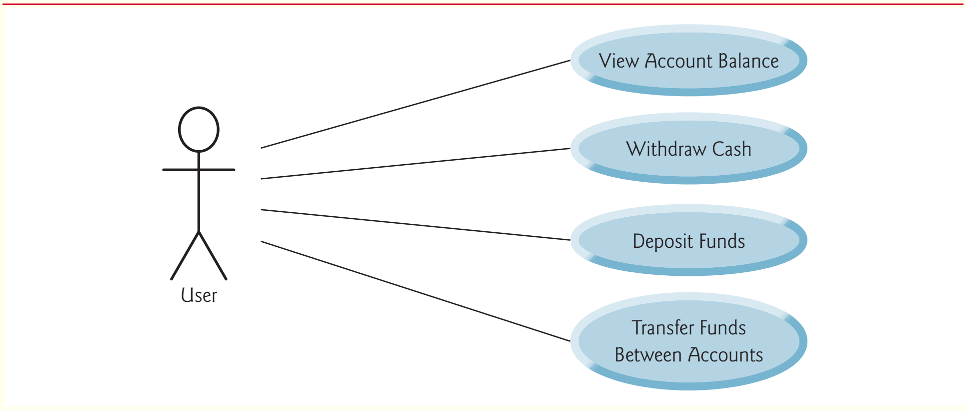


Fig. 33.26 | Use case diagram for a modified version of our ATM system that also allows users to transfer money between accounts.

33.2 b.

33.3 d.

33.4 [*Note:* Answers may vary.] Figure 33.27 presents a class diagram that shows some of the composition relationships of a class `Car`.

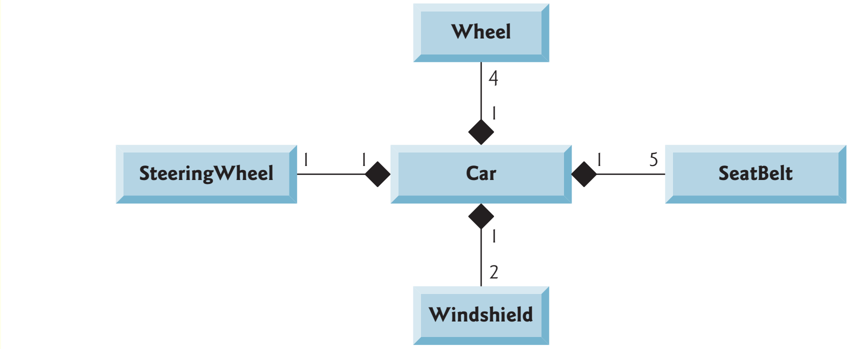


Fig. 33.27 | Class diagram showing composition relationships of a class `Car`.

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

33.6 True.

33.7 Figure 33.28 presents a class diagram for the ATM including class `Deposit` instead of class `Withdrawal` (as in Fig. 33.10). `Deposit` does not access `CashDispenser`, but does access `DepositSlot`.

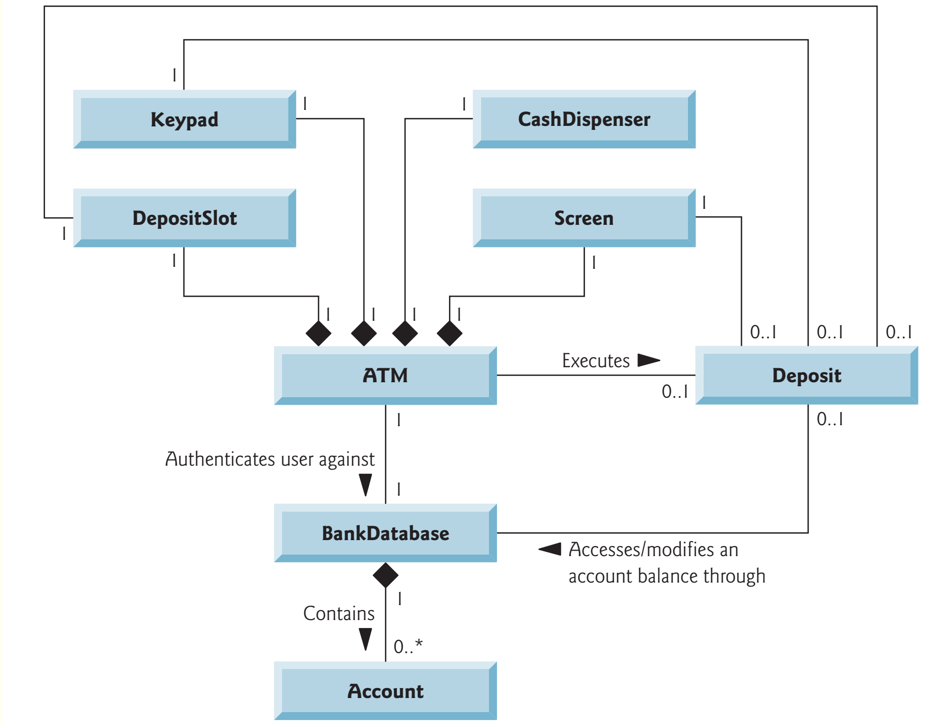


Fig. 33.28 | Class diagram for the ATM system model including class `Deposit`.

33.8 b.

33.9 c. Fly is an operation or behavior of an airplane, not an attribute.

33.10 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.

33.11 False. State diagrams model some of the behavior of a system.

33.12 a.

33.13 Figure 33.29 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 `totalBalance` 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.

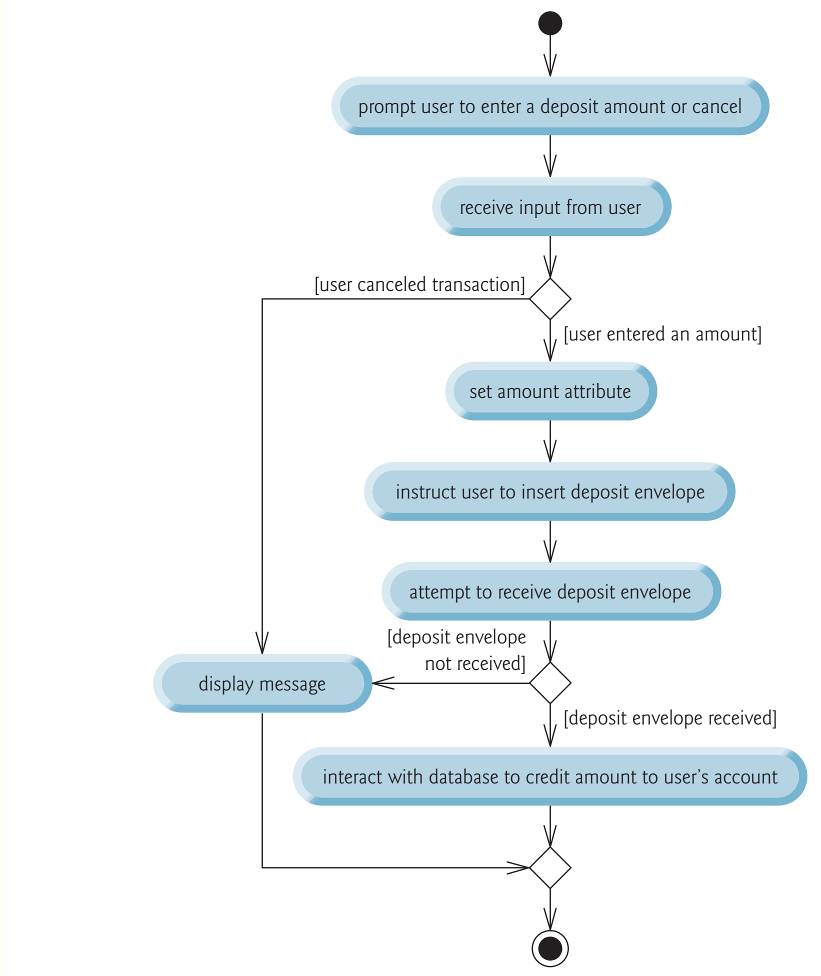


Fig. 33.29 | Activity diagram for a deposit transaction.

33.14 c.

33.15 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`:

```
getAmount() : Double
```

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

33.17 c.

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

33.19 Figure 33.30 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`.

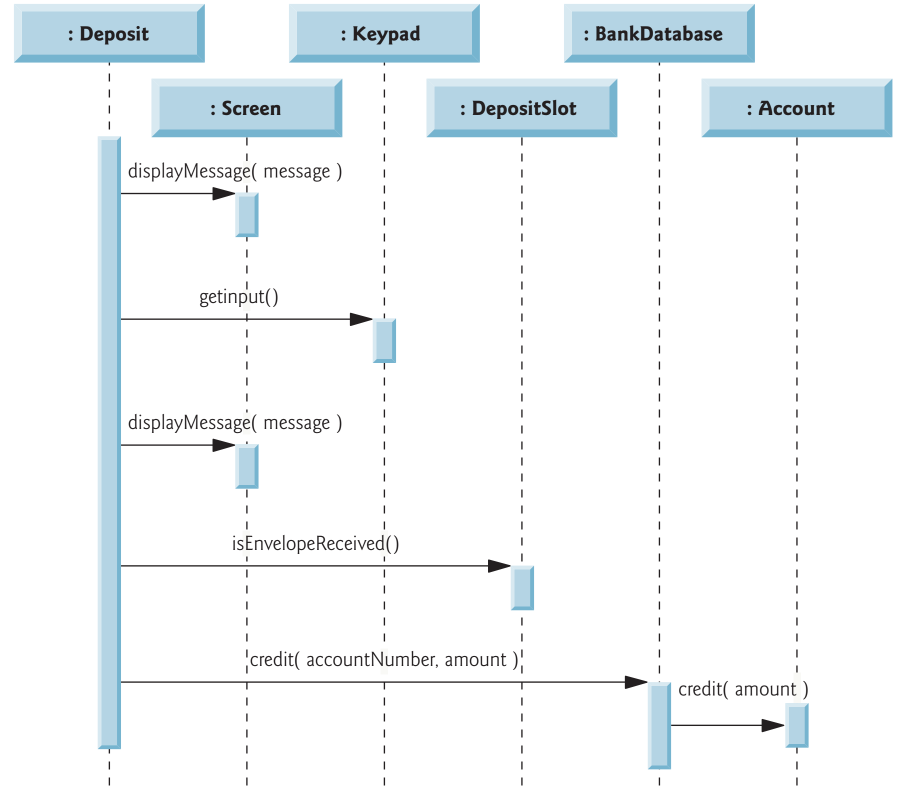


Fig. 33.30 | Sequence diagram that models a `Deposit` executing.