NEXT

- Table of Contents
- Index

C++ How to Program, Fifth Edition

By H. M. Deitel - Deitel & Associates, Inc., P. J. Deitel - Deitel & Associates, Inc.

Publisher: **Prentice Hall**
Pub Date: **January 05, 2005**
Print ISBN-10: **0-13-185757-6**
eText ISBN-10: **0-13-186103-4**
Print ISBN-13: **978-0-13-185757-5**
eText ISBN-13: **978-0-13-186103-9**
Pages: **1536**

Best-selling C++ text significantly revised to include new early objects coverage and new streamlined case studies.

NEXT

[Page 56 (continued)]

2.8. (Optional) Software Engineering Case Study: Examining the ATM Requirements Document

Now we begin our optional object-oriented design and implementation case study. The "Software Engineering Case Study" sections at the ends of this and the next several chapters will ease you into object orientation. We will develop software for a simple automated teller machine (ATM) system, providing you with a concise, carefully paced, complete design and implementation experience. In Chapters 37, 9 and 13, we will perform the various steps of an object-oriented design (OOD) process using the UML, while relating these steps to the object-oriented concepts discussed in the chapters. Appendix G implements the ATM using the techniques of object-oriented programming (OOP) in C++. We present the complete case study solution. This is not an exercise; rather, it is an end-to-end learning experience that concludes with a detailed walkthrough of the C++ code that implements our design. It will acquaint you with the kinds of substantial problems encountered in industry and their potential solutions.

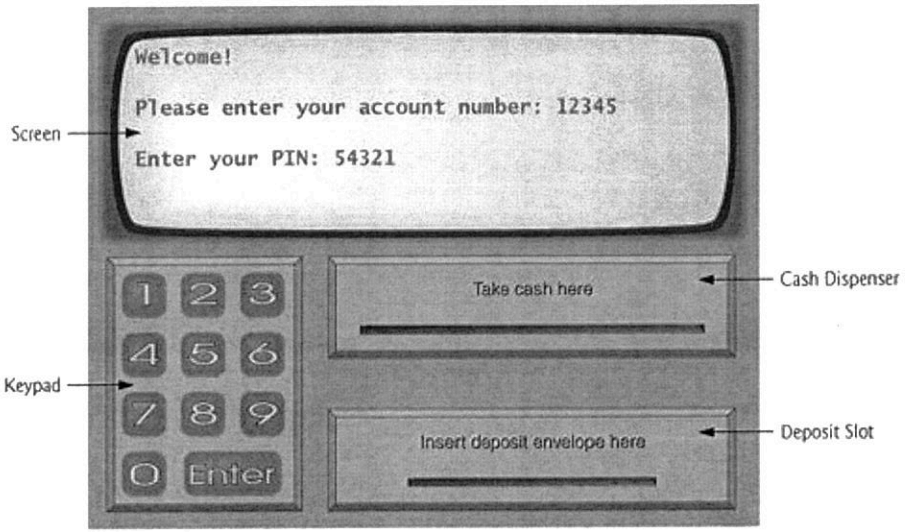
We begin our design process by presenting a **requirements document** that specifies the overall purpose of the ATM system and *what* it must do. Throughout the case study, we refer to the requirements document to determine precisely what functionality the system must include.

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. 2.15). 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).

Figure 2.15. Automated teller machine user interface.
(This item is displayed on page 57 in the print version)

[View full size image]



The user interface of the automated teller machine contains the following hardware components:

- 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 an 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.]

[Page 57]

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 of the software 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. [*Note:* A database is an organized collection of data stored on a computer.] For each bank account, the database stores an account number, a PIN and a balance indicating the amount of money in the account. [*Note:* For simplicity, we assume that the bank plans to build only one ATM, so we do not need to 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 reasonably complicated security issues that go well beyond the scope of a first- or second-semester computer science course. We make the simplifying assumption, however, that the bank trusts the ATM to access and manipulate the information in the database without significant security measures.]

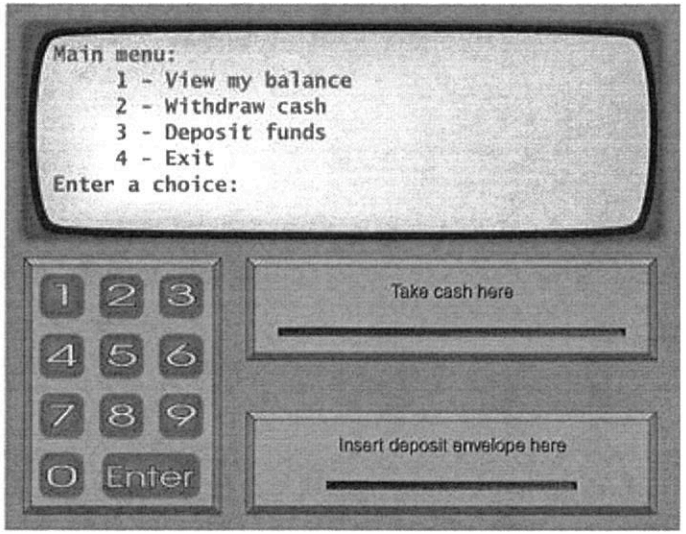
Upon first approaching the ATM, the user should experience the following sequence of events (shown in Fig. 2.15):

[Page 58]

1. The screen displays a welcome message 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.
5. If the user enters a valid account number and the correct PIN for that account, the screen displays the main menu (Fig. 2.16). 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.

Figure 2.16. ATM `main menu`.

[View full size image]



After the ATM authenticates the user, the main menu (Fig. 2.16) displays a numbered option for each of the three types of transactions: balance inquiry (option 1), withdrawal (option 2) and deposit (option 3). The main menu also displays an option that allows the user to exit the system (option 4). The user then chooses either to perform a transaction (by entering 1, 2 or 3) or to exit the system (by entering 4). If the user enters an invalid option, the screen displays an error message, then redisplays to the main menu.

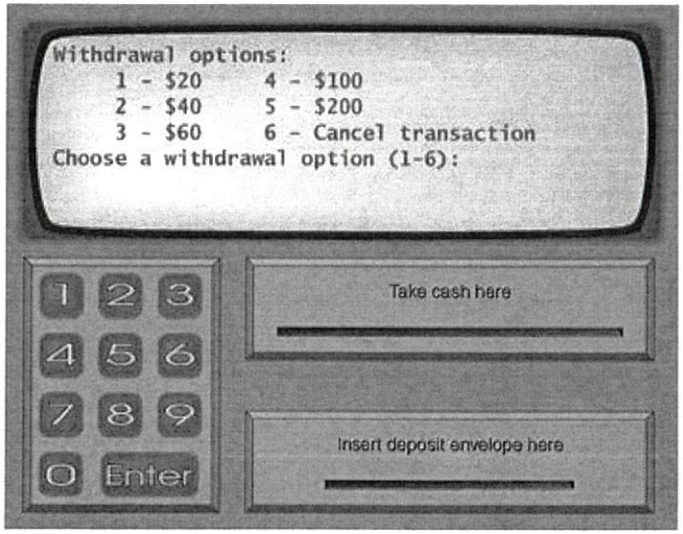
If the user enters 1 to make a balance inquiry, the screen displays the user's account balance. To do so, the ATM must retrieve the balance from the bank's database.

The following actions occur when the user enters 2 to make a withdrawal:

6. The screen displays a menu (shown in Fig. 2.17) 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).

Figure 2.17. ATM withdrawal menu.

[View full size image]



7. The user enters a menu selection (16) using the keypad.

8. 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 withdrawal amount), the ATM proceeds to *Step 4*. If the user chooses to cancel the transaction (option 6), the ATM displays the main menu (Fig. 2.16) and waits for user input.

9. If the cash dispenser contains enough cash to satisfy the request, 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*.

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

11. The cash dispenser dispenses the desired amount of money to the user.

12. The screen displays a message reminding the user to take the money.

The following actions occur when the user enters 3 (while the main menu is displayed) to make a deposit:

13. The screen prompts the user to enter a deposit amount or to type 0 (zero) to cancel the transaction.

14. 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., $1.25). Instead, the user must enter a deposit amount as a number of cents (e.g., 125). The ATM then divides this number by 100 to obtain a number representing a dollar amount (e.g., 125 ÷ 100 = 1.25).]

---

15. 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 (Fig. 2.16) and waits for user input.

16. The screen displays a message telling the user to insert a deposit envelope into the deposit slot.

17. If the deposit slot receives a deposit envelope within two minutes, the ATM credits (i.e., adds) the deposit amount to the user's account balance in the bank's database. [*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, the system should redisplay the main menu (Fig. 2.16) so that the user can perform additional transactions. If the user chooses to exit the system (option 4), 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** that might include interviews with potential 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 financial 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.

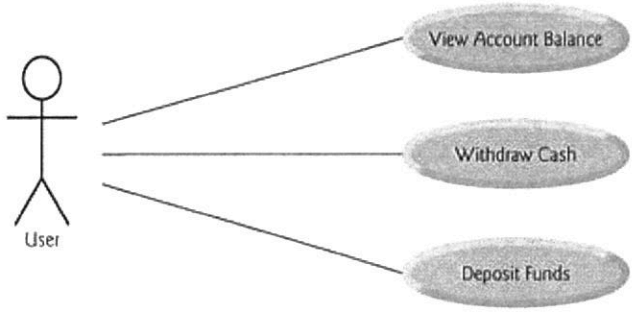
The process of requirements gathering is a key task of the first stage of the software life cycle. The **software life cycle** specifies the stages through which software evolves from the time it is first conceived to the time it is retired from use. These stages typically include: analysis, design, 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 of the software life cycle focuses on defining the problem to be solved. When designing any system, one must certainly *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 our ATM system in sufficient detail that you do not need to go through an extensive analysis stageit has 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 of which represents 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 use cases (Fig. 2.18).

Figure 2.18. Use case diagram for the ATM system from the user's perspective.

[View full size image]



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

Use Case Diagrams

We now introduce the first of several UML diagrams in our ATM case study. We create a **use case diagram** to model the interactions between a system's clients (in this case study, bank customers) and the system. The goal is to show the kinds of interactions users have with a system without providing the detailsthese are provided in other UML diagrams (which we present throughout the case study). Use case diagrams are often accompanied by informal text that describes the use cases in more detaillike 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 simple but indispensable tools that help system designers remain focused on satisfying the users' needs.

Figure 2.18 shows the use case diagram for our ATM system. The stick figure represents an **actor**, which defines the roles that an external entitysuch as a person or another systemplays 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 personwhen playing the part of a Usercan play while interacting with the ATM. Note that 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.

We identify the actor in our system by examining the requirements document, which states, "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 entitya real personplays the part of the User to perform financial transactions. Figure 2.18 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.

[Page 62]

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 stagea **design specification**should specify clearly *how* the system should be constructed to satisfy these requirements. In the next several "Software Engineering Case Study" sections, we perform the steps of a simple object-oriented design (OOD) process on the ATM system to produce a design specification containing a collection of UML diagrams and supporting text. Recall that 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 a division 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. 2.15), 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 behaviordesigners must specify both. There are several distinct 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 specifies 13 diagram types for documenting the models of systems. Each models a distinct characteristic of a system's structure or behaviorsix diagrams relate to system structure; the remaining seven relate to system behavior. We list here only the six types of diagrams used in our case studyone of these (class diagrams) models system structurethe remaining five model system behavior. We overview the remaining seven UML diagram types in Appendix H, UML 2: Additional Diagram Types

1. **Use case diagrams**, such as the one in Fig. 2.18, 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 will study in Section 3.11, 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 will study in Section 3.11, model the ways in which an object changes state. An object's **state** is indicated by the values of all the object's attributes at a given time. When an object changes state, that object 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 the ATM allows the user to perform financial transactions (e.g., view account balance, withdraw cash, deposit funds).

[Page 63]

4. **Activity diagrams**, which you will also study in Section 5.11, model an object's **activity**the object's workflow (sequence of events) during program execution. An activity diagram models the actions the object performs and specifies the order in which it performs these actions. 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 will learn in Section 7.12 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 will learn in Section 7.12 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 3.11, 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.

Internet and Web Resources

The following URLs provide information on object-oriented design with the UML.

www-306.ibm.com/software/rational/uml/

Lists frequently asked questions about the UML, provided by IBM Rational.

www.softdocwiz.com/Dictionary.htm

Hosts the Unified Modeling Language Dictionary, which lists and defines all terms used in the UML.

www-306.ibm.com/software/rational/offerings/design.html

Provides information about IBM Rational software available for designing systems. Provides downloads of 30-day trial versions of several products, such as IBM Rational Rose® XDE Developer.

www.embarcadero.com/products/describe/index.html

Provides a 15-day trial license for the Embarcadero Technologies® UML modeling tool Describe.™

PREV NEXT

[Page 110 (continued)]

3.11. (Optional) Software Engineering Case Study: Identifying the Classes in the ATM Requirements Document

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

Identifying the Classes in a System

We begin our OOD process by identifying the classes required to build the ATM system. We will eventually describe these classes using UML class diagrams and implement these classes in C++. First, we review the requirements document of Section 2.8 and find key nouns and noun phrases to help us identify classes that comprise the ATM system. We may decide that some of these nouns and noun phrases are attributes of other classes in the system. We may also conclude that some of the nouns do not correspond to parts of the system and thus should not be modeled at all. Additional classes may become apparent to us as we proceed through the design process.

Figure 3.18 lists the nouns and noun phrases in the requirements document. We list them from left to right in the order in which they appear in the requirements document. We list only the singular form of each noun or noun phrase.

We create classes only for the nouns and noun phrases that have significance in the ATM system. We do not need to model "bank" as a class, because the bank is not a part of the ATM systemthe bank simply wants us to build the ATM. "Customer" and "user" also represent entities outside of the systemthey are important because they interact with our ATM system, but we do not need to model them as classes in the ATM software. Recall that we modeled an ATM user (i.e., a bank customer) as the actor in the use case diagram of Fig. 2.18.

We do not model "$20 bill" or "deposit envelope" as classes. These are physical objects in the real world, but they are not part of what is being automated. We can adequately represent the presence of bills in the system using an attribute of the class that models the cash dispenser. (We assign attributes to classes in Section 4.13.) 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 envelopean operation performed by the class that models the deposit slotis sufficient to represent the presence of an envelope in the system. (We assign operations to classes in Section 6.22.)

[Page 111]

Figure 3.18. Nouns and noun phrases in the requirements document.

Nouns and noun phrases in the requirements document		
bank	money / fund	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	

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

Though the requirements document frequently describes a "transaction" in a general sense, we do not model the broad notion of a financial transaction at this time. Instead, we model the three types of transactions (i.e., "balance inquiry," "withdrawal" and "deposit") as individual classes. These classes possess specific attributes needed for executing the transactions they represent. For example, a withdrawal needs to know the amount of money the user wants to withdraw. A balance inquiry, however, does not require any additional data. Furthermore, the three transaction classes exhibit unique behaviors. A withdrawal includes dispensing cash to the user, whereas a deposit involves receiving deposit envelopes from the user. [*Note:* In Section 13.10, we "factor out" common features of all transactions into a general "transaction" class using the object-oriented concepts of abstract classes and inheritance.]

We determine the classes for our system based on the remaining nouns and noun phrases from Fig. 3.18. Each of these refers to one or more of the following:

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

- withdrawal
- deposit

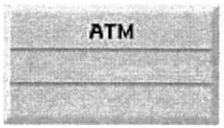
The elements of this list are likely to be classes we will need to implement our system.

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

Modeling Classes

The UML enables us to model, via **class diagrams**, the classes in the ATM system and their interrelationships. Figure 3.19 represents class `ATM`. In the UML, each class is modeled as a rectangle with three compartments. The top compartment contains the name of the class, centered horizontally and in boldface. The middle compartment contains the class's attributes. (We discuss attributes in Section 4.13 and Section 5.11.) The bottom compartment contains the class's operations (discussed in Section 6.22). In Fig. 3.19 the middle and bottom compartments are empty, because we have not yet determined this class's attributes and operations.

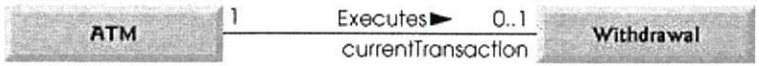
Figure 3.19. Representing a class in the UML using a class diagram.



Class diagrams also show the relationships between the classes of the system. Figure 3.20 shows how our classes `ATM` and `Withdrawal` relate to one another. For the moment, we choose to model only this subset of classes for simplicity. We present a more complete class diagram later in this section. Notice that the rectangles representing classes in this diagram are not subdivided into compartments. The UML allows the suppression of class attributes and operations in this manner, when appropriate, to create more readable diagrams. Such a diagram is said to be an **elided diagram**one in which some information, such as the contents of the second and third compartments, is not modeled. We will place information in these compartments in Section 4.13 and Section 6.22

Figure 3.20. Class diagram showing an association among classes.

[View full size image]



In Fig. 3.20, the solid line that connects the two classes represents an **association**a relationship between classes. The numbers near each end of the line are **multiplicity** values, which indicate how many objects of each class participate in the association. In this case, following the line from one end to the other reveals that, at any given moment, one `ATM` object participates in an association with either zero or one `Withdrawal` objectszero if the current user is not currently performing a transaction or has requested a different type of transaction, and one if the user has requested a withdrawal. The UML can model many types of multiplicity. Figure 3.21 lists and explains the multiplicity types.

An association can be named. For example, the word `Executes` above the line connecting classes `ATM` and `Withdrawal` in Fig. 3.20 indicates the name of that association. This part of the diagram reads "one object of class `ATM` executes zero or one objects of class `Withdrawal`." Note that association names are directional, as indicated by the filled arrow-headso 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. 3.20 is a **role name**, which identifies the role the `Withdrawal` object plays in its relationship with the `ATM`. A role name adds meaning to an association between classes by identifying the role a class plays in the context of an association. A class can play several roles in the same system. For example, in a school personnel system, a person may play the role of "professor" when relating to students. The same person may take on the role of "colleague" when participating in a relationship with another professor, and "coach" when coaching student athletes. In Fig. 3.20, 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.

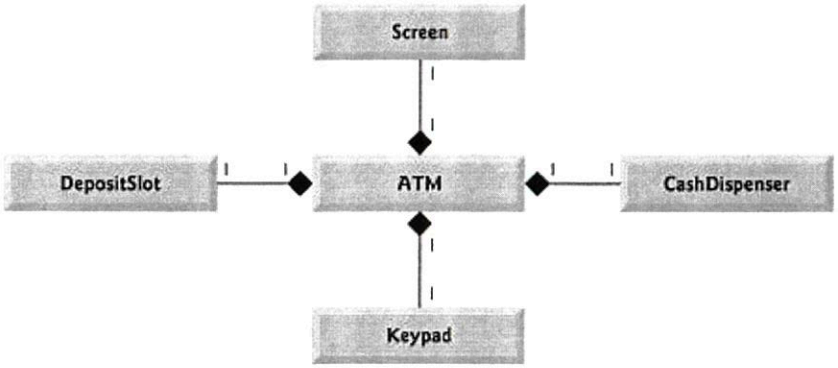
Figure 3.21. Multiplicity types.

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

In Fig. 3.22, the **solid diamonds** attached to the association lines of class `ATM` indicate that class `ATM` has a **composition** relationship with classes `Screen`, `Keypad`, `CashDispenser` and `DepositSlot`. Composition implies a whole/part relationship. The class that has the composition symbol (the solid diamond) on its end of the association line is the whole (in this case, `ATM`), and the classes on the other end of the association lines are the partsin this case, classes `Screen`, `Keypad`, `CashDispenser` and `DepositSlot`. The compositions in Fig. 3.22 indicate that an object of class `ATM` is formed from one object of class `Screen`, one object of class `CashDispenser`, one object of class `Keypad` and one object of class `DepositSlot`. The ATM "has a" screen, a keypad, a cash dispenser and a deposit slot. The **"has-a" relationship** defines composition. (We will see in the "Software Engineering Case Study" section in Chapter 13 that the "is-a" relationship defines inheritance.)

Figure 3.22. Class diagram showing composition relationships.

[View full size image]



According to the UML specification, composition relationships have the following properties:

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

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

[Page 115]

Figure 3.23 shows a class diagram for the ATM system. This diagram models most of the classes that we identified earlier in this section, as well as the associations between them that we can infer from the requirements document. [*Note:* Classes `BalanceInquiry` and `Deposit` participate in associations similar to those of class `Withdrawal`, so we have chosen to omit them from this diagram to keep it simple. In Chapter 13, we expand our class diagram to include all the classes in the ATM system.]

Figure 3.23. Class diagram for the ATM system model.

[View full size image]

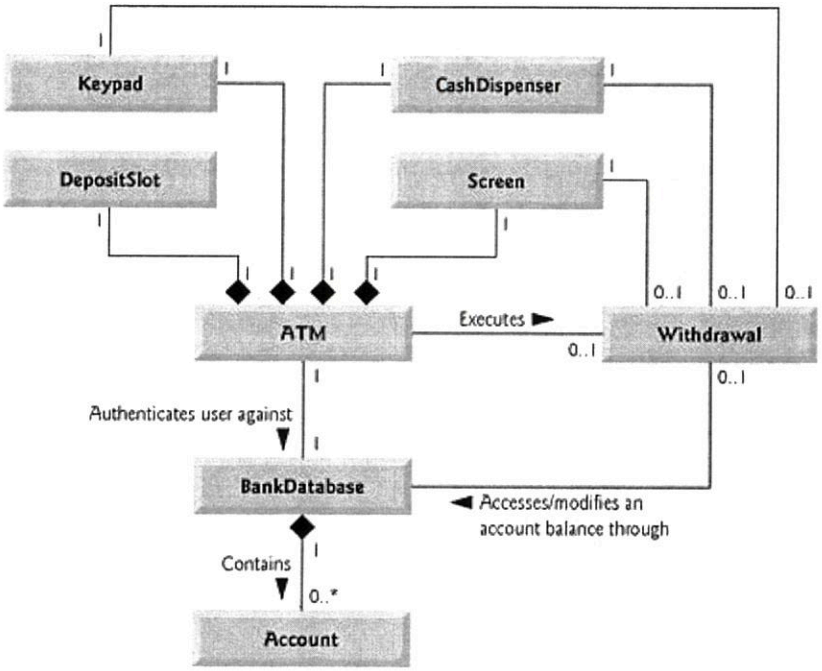


Figure 3.23 presents a graphical model of the structure of the ATM system. This class diagram includes classes `BankDatabase` and `Account`, and several associations that were not present in either Fig. 3.20 or Fig. 3.22. The class diagram shows that class `ATM` has a **one-to-one relationship** with class `BankDatabase`one `ATM` object authenticates users against one `BankDatabase` object. In Fig. 3.23, we also model the fact that the bank's database contains information about many accountsone object of class `BankDatabase` participates in a composition relationship with zero or more objects of class `Account`. Recall from Fig. 3.21 that the multiplicity value 0..* at the `Account` end of the association between class `BankDatabase` and class `Account` indicates that zero or more objects of class `Account` take part in the association. Class `BankDatabase` has a **one-to-many relationship** with class `Account`the `BankDatabase` stores many `Account`s. Similarly, class `Account` has a **many-to-one relationship** with class `BankDatabase`there can be many `Accounts` stored in the `BankDatabase`. [*Note:* Recall from Fig. 3.21 that the multiplicity value * is identical to 0..*. We include 0..* in our class diagrams for clarity.]

[Page 116]

Figure 3.23 also indicates that if the user is performing a withdrawal, "one object of class `Withdrawal` accesses/modifies an account balance through one object of class `BankDatabase`." We could have created an association directly between class `Withdrawal` and class `Account`. The requirements document, however, states that the "ATM must interact with the bank's account information database" to perform transactions. A bank account contains sensitive information, and systems engineers must always consider the security of personal data when designing a system. Thus, only the `BankDatabase` can access and manipulate an account directly. All other parts of the system must interact with the database to retrieve or update account information (e.g., an account balance).

The class diagram in Fig. 3.23 also models associations between class `Withdrawal` and classes `Screen`, `CashDispenser` and `Keypad`. A withdrawal transaction includes prompting the user to choose a withdrawal amount and receiving numeric input. These actions require the use of the screen and the keypad, respectively. Furthermore, dispensing cash to the user requires access to the cash dispenser.

Classes `BalanceInquiry` and `Deposit`, though not shown in Fig. 3.23, 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.

PREV NEXT

4.13. (Optional) Software Engineering Case Study: Identifying Class Attributes in the ATM System

In Section 3.11, we began the first stage of an object-oriented design (OOD) for our ATM systemanalyzing the requirements document and identifying the classes needed to implement the system. We listed the nouns and noun phrases in the requirements document and identified a separate class for each one that plays a significant role in the ATM system. We then modeled the classes and their relationships in a UML class diagram (Fig. 3.23). Classes have attributes (data) and operations (behaviors). Class attributes are implemented in C++ programs as data members, and class operations are implemented as member functions. In this section, we determine many of the attributes needed in the ATM system. In Chapter 5, we examine how these attributes represent an object's state. In Chapter 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 setting, its volume setting and its AM or FM setting. A car's attributes include its speedometer and odometer readings, the amount of gas in its tank and what gear it is 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 one 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 3.11. We also create attributes to represent any additional data that a class may need, as such needs become clear throughout the design process.

Figure 4.23 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`.

Figure 4.23. Descriptive words and phrases from the ATM requirements.

[Page 166]

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]

[Page 166]

Figure 4.23 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 5.11), so we include userAuthenticated as a **Boolean attribute** (i.e., an attribute that has a value of either true or false). The UML Boolean type is equivalent to the bool type in C++. This attribute indicates whether the ATM has successfully authenticated the current userUserAuthenticated 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 enter 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 them to take place. Class BalanceInquiry, however, needs no additional data to perform its taskit 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 of UML type `Double`: `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` soon 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`.

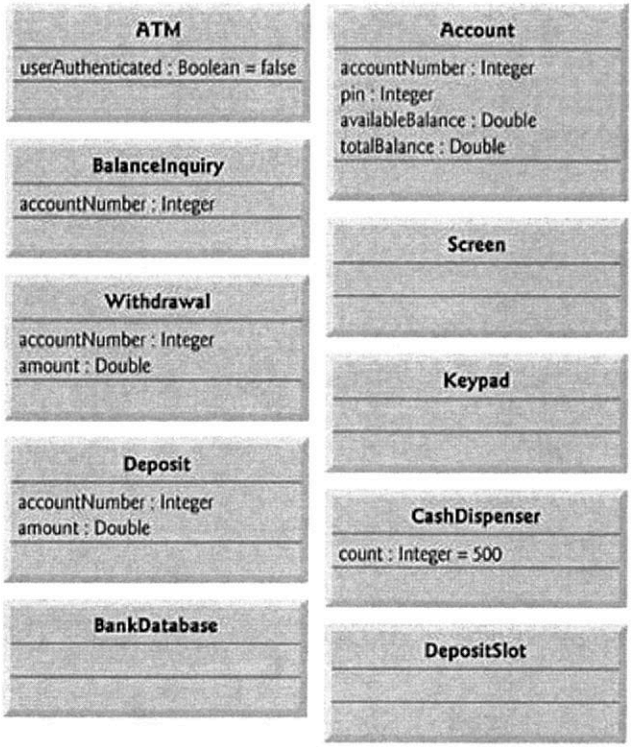
For real problems in industry, there is no guarantee that requirements documents will be rich enough and precise enough for the object-oriented systems designer to determine all the attributes or even all the classes. The need for additional classes, attributes and behaviors may become clear as the design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

Modeling Attributes

The class diagram in Fig. 4.24 lists some of the attributes for the classes in our systemthe descriptive words and phrases in Fig. 4.23 helped us identify these attributes. For simplicity, Fig. 4.24 does not show the associations among classeswe showed these in Fig. 3.23. This is a common practice of systems designers when designs are being developed. Recall from Section 5.11 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.

Figure 4.24. Classes with attributes.
(This item is displayed on page 168 in the print version)

[View full size image]



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 C++, an attribute can be represented by a fundamental type, such as `bool`, `int` or `double`, or a class typeas discussed in Chapter 3. We have chosen to model only primitive-type attributes in Fig. 4.24we discuss the reasoning behind this decision shortly. [*Note:* Figure 4.24 lists UML data types for the attributes. When we implement the system, we will associate the UML types `Boolean`, `Integer` and `Double` with the C++ fundamental types bool, `int` and `double`, respectively.]

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.

[Page 168]

Figure 4.24 does not include any attributes for classes `Screen`, `Keypad` and `DepositSlot`. These are important components of our system, for which our design process simply has not yet revealed any attributes. We may still discover some, however, in the remaining phases of design or when we implement these classes in C++. This is perfectly normal for the iterative process of software engineering.

Software Engineering Observation 4.8

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.

Note that Fig. 4.24 also does not include attributes for class `BankDatabase`. Recall from Chapter 3 that in C++, attributes can be represented by either fundamental types or class types. We have chosen to include only fundamental-type attributes in the class diagram in Fig. 4.24 (and in similar class diagrams throughout the case study). A class-type attribute is modeled more clearly as an association (in particular, a composition) between the class with the attribute and the class of the object of which the attribute is an instance. For example, the class diagram in Fig. 3.23 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 C++, we will be required to create an attribute of class `BankDatabase` to hold zero or more `Account` objects. Similarly, we will assign attributes to 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. 4.24, because the compositions modeled in Fig. 3.23 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. 4.24 provides a solid basis for the structure of our model, but the diagram is not complete. In Section 5.11, we identify the states and activities of the objects in the model, and in Section 6.22 we identify the operations that the objects perform. As we present more of the UML and object-oriented design, we will continue to strengthen the structure of our model.

Software Engineering Case Study Self-Review Exercises

4.1 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.

4.2 Which of the following is not an attribute of an airplane?

- **a.** length
- **b.** wingspan
- **c.** fly
- **d.** number of seats

PREV NEXT

[Page 222]

5.11. (Optional) Software Engineering Case Study: Identifying Objects' States and Activities in the ATM System

In Section 4.13, we identified many of the class attributes needed to implement the ATM system and added them to the class diagram in Fig. 4.24. In this section, we 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. We present the activities of `BalanceInquiry` and `Withdrawal` TRansaction objects in this section, as they represent two of the key activities in the ATM system.

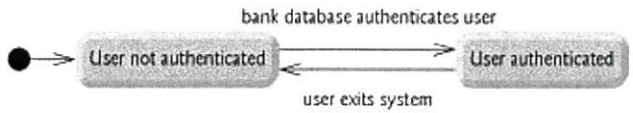
State Machine Diagrams

Each object in a system goes through a series of discrete states. An object's current state is indicated by the values of the object's attributes at a given time. **State machine diagrams** (commonly called **state diagrams**) model key 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 structure of the system, state diagrams model some of the behavior of the system.

Figure 5.26 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. 4.24. This attribute is initialized to `false`, or the "User not authenticated" state, according to the state diagram.

Figure 5.26. State diagram for the `ATM` object.

[View full size image]



[Page 223]

The arrows with stick arrowheads 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" state 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 the corresponding account in the database. If the database indicates that 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 in preparation for the next ATM user.

Software Engineering Observation 5.3

Software designers do not generally create state diagrams showing every possible state and state transition for all attributesthere are simply too many of them. State diagrams typically show only the most important or complex states and state transitions.

Activity Diagrams

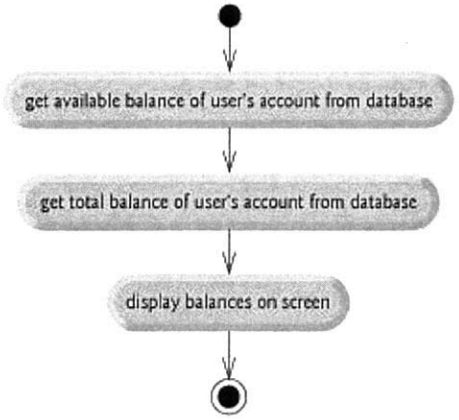
Like a state diagram, an activity diagram models aspects of system behavior. Unlike a state diagram, an activity diagram models an object's workflow (sequence of events) during program execution. An activity diagram models the actions the object will perform and in what order. Recall that we used UML activity diagrams to illustrate the flow of control for the control statements presented in Chapter 4 and Chapter 5.

The activity diagram in Fig. 5.27 models the actions involved in executing a `BalanceInquiry` 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 menua `BalanceInquiry` object does not perform or initiate these actions, so we do not model them here. The diagram begins with retrieving the available balance of the user's account from the database. Next, the `BalanceInquiry` retrieves the total balance of the account. Finally, the transaction displays the balances on the screen. This action completes the execution of the transaction.

[Page 224]

Figure 5.27. Activity diagram for a `BalanceInquiry` transaction.
(This item is displayed on page 223 in the print version)

[View full size image]

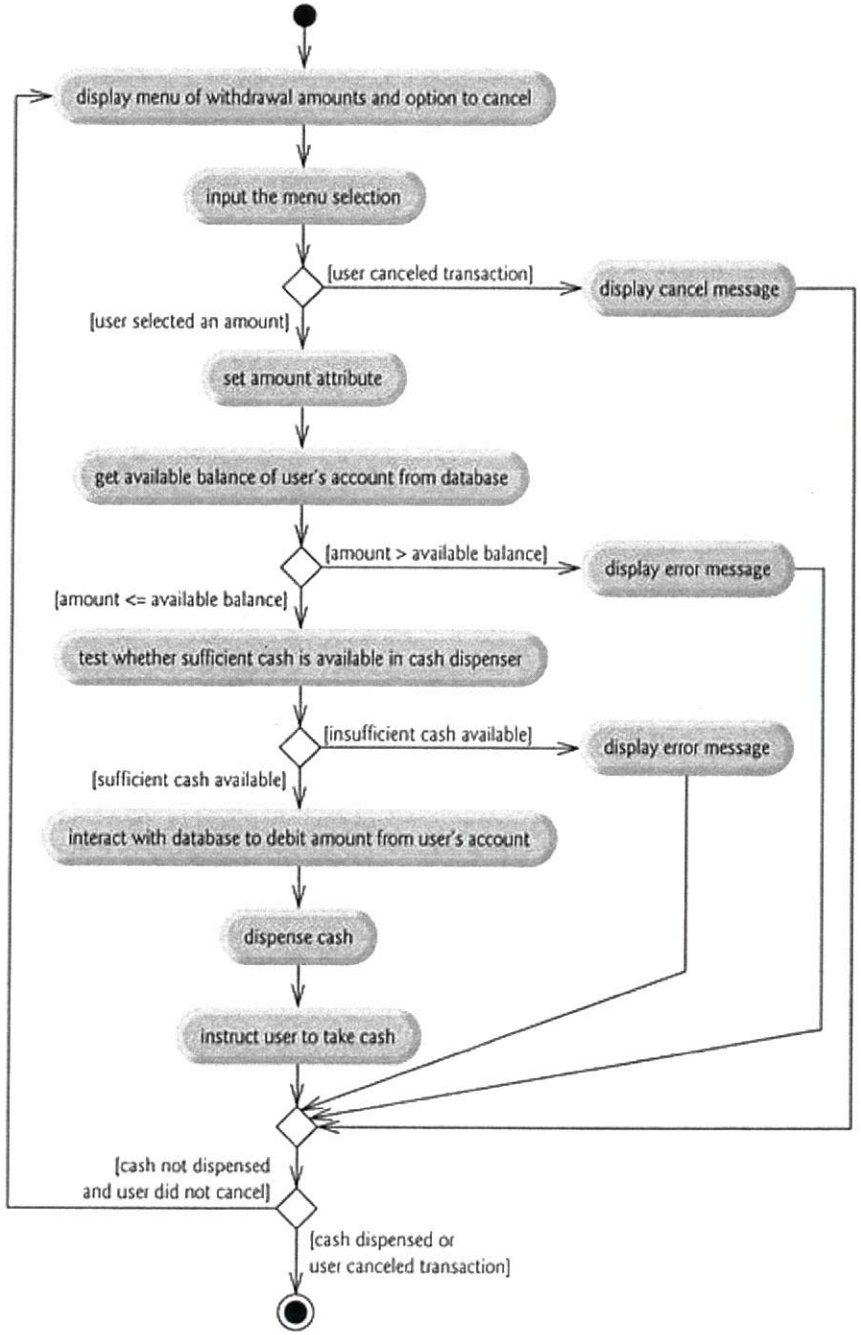


The UML represents an action in an activity diagram as an action state modeled by a rectangle with its left and right sides replaced by arcs curving outward. Each action state contains an action expressionfor example, "get available balance of user's account from database"that specifies an action to 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. 5.27) represents the activity's initial statethe beginning of the workflow before the object performs the modeled actions. In this case, the transaction first executes the "get available balance of user's account from database" action expression. Second, the transaction retrieves the total balance. Finally, the transaction displays both balances on the screen. The solid circle enclosed in an open circle (at the bottom of Fig. 5.27) represents the final statethe end of the workflow after the object performs the modeled actions.

Figure 5.28 shows an activity diagram for a `Withdrawal` transaction. We assume that a `Withdrawal` object has been assigned a valid account number. We do not model the user selecting a withdrawal from the main menu or the ATM returning the user to the main menu because these are not actions performed by a `Withdrawal` object. The transaction first displays a menu of standard withdrawal amounts (Fig. 2.17) and an option to cancel the transaction. The transaction then inputs a menu selection from the user. The activity flow now arrives at a decision symbol. This point determines the next action based on the associated guard conditions. If the user cancels the transaction, the system displays an appropriate message. Next, the cancellation flow reaches a merge symbol, where this activity flow joins the transaction's other possible activity flows (which we discuss shortly). Note that a merge can have any number of incoming transition arrows, but only one outgoing transition arrow. The decision at the bottom of the diagram determines whether the transaction should repeat from the beginning. When the user has canceled the transaction, the guard condition "cash dispensed or user canceled transaction" is true, so control transitions to the activity's final state.

Figure 5.28. Activity diagram for a `Withdrawal` TRansaction.
(This item is displayed on page 225 in the print version)

[View full size image]


display menu of withdrawal amounts and option to cancel
input the menu selection
[user canceled transaction]
display cancel message
[user selected an amount]
set amount attribute
get available balance of user's account from database
[amount > available balance]
display error message
[amount <= available balance]
test whether sufficient cash is available in cash dispenser
[insufficient cash available]
display error message
[sufficient cash available]
interact with database to debit amount from user's account
dispense cash
instruct user to take cash
[cash not dispensed
and user did not cancel]
[cash dispensed or
user canceled transaction]

If the user selects a withdrawal amount from the menu, the transaction sets `amount` (an attribute of class `Withdrawal` originally modeled in Fig. 4.24) to the value chosen by the user. The transaction next gets 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. Control then merges with the other activity flows before reaching the decision at the bottom of the diagram. The guard decision "cash not dispensed and user did not cancel" is true, so the activity flow returns to the top of the diagram, and the transaction 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 tests whether the cash dispenser has enough cash to satisfy the withdrawal request. If it does not, the transaction displays an appropriate error message and passes through the merge before reaching the final decision. Cash was not dispensed, so the activity flow returns to the beginning of the activity diagram, and the transaction 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 the cash that is dispensed. The main flow of activity next merges with the two error flows and the cancellation flow. In this case, cash was dispensed, so the activity flow reaches the final state.

We have taken the first steps in modeling the behavior of the ATM system and have shown how an object's attributes participate in the object's activities. In Section 6.22, we investigate the operations of our classes to create a more complete model of the system's behavior.

Software Engineering Case Study Self-Review Exercises

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

5.2 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

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

Answers to Software Engineering Case Study Self-Review Exercises

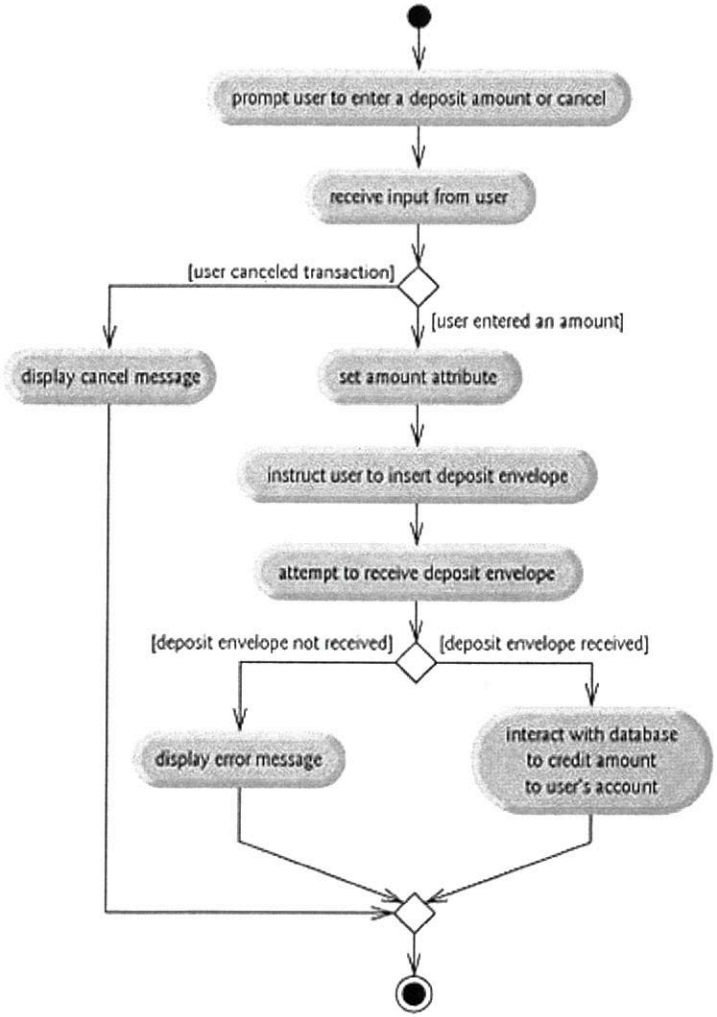
5.1 False. State diagrams model some of the behavior of a system.

5.2 a.

5.3 Figure 5.29 presents an activity diagram for a deposit transaction. The diagram 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 involves increasing 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 clearthis occurs independently of the ATM system.

Figure 5.29. Activity diagram for a `Deposit` TRansaction.
(This item is displayed on page 227 in the print version)

[View full size image]



PREV NEXT

[Page 298]

6.22. (Optional) Software Engineering Case Study: Identifying Class Operations in the ATM System

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

Identifying Operations

An operation is a service that objects of a class provide to clients of the class. Consider the operations of some real-world objects. A radio's operations include setting its station and volume (typically invoked by a person adjusting the radio's controls). A car's operations include accelerating (invoked by the driver pressing the accelerator pedal), decelerating (invoked by the driver pressing the brake pedal or releasing the gas pedal), turning and shifting gears. Software objects can offer operations as wellfor 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.

[Page 299]

We can derive many of the operations of each class by examining the key verbs and verb phrases in the requirements document. We then relate each of these to particular classes in our system (Fig. 6.34). The verb phrases in Fig. 6.34 help us determine the operations of each class.

Figure 6.34. Verbs and verb phrases for each class in the ATM system.

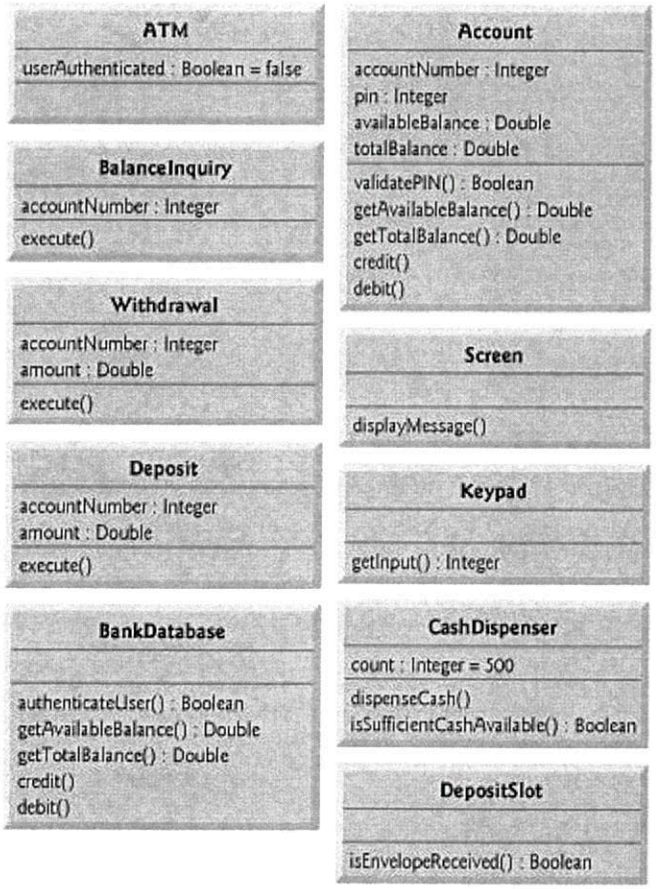
Class	Verbs and verb phrases
`ATM`	executes financial transactions
`BalanceInquiry`	[none in the requirements document]
`Withdrawal`	[none in the requirements document]
`Deposit`	[none in the requirements document]
`BankDatabase`	authenticates a user, retrieves an account balance, credits a deposit amount 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

Modeling Operations

To identify operations, we examine the verb phrases listed for each class in Fig. 6.34. The "executes financial transactions" phrase associated with class `ATM` implies that class `ATM` instructs transactions to execute. Therefore, classes `BalanceInquiry`, `Withdrawal` and `Deposit` each need an operation to provide this service to the ATM. We place this operation (which we have named `execute`) in the third compartment of the three transaction classes in the updated class diagram of Fig. 6.35. During an ATM session, the `ATM` object will invoke the `execute` operation of each transaction object to tell it to execute.

Figure 6.35. Classes in the ATM system with attributes and operations.
(This item is displayed on page 300 in the print version)

[View full size image]



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

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

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

parameterName : *parameterType*

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

Operations of Class BankDatabase and Class Account

Figure 6.34 lists the phrase "authenticates a user" next to class BankDatabasethe database is the object that contains the account information necessary to determine whether the account number and PIN entered by a user match those of an account held at the bank. Therefore, class BankDatabase needs an operation that provides an authentication service to the ATM. We place the operation authenticateUser in the third compartment of class BankDatabase (Fig. 6.35). However, an object of class Account, not class BankDatabase, stores the account number and PIN that must be accessed to authenticate a user, so class Account must provide a service to validate a PIN obtained through user input against a PIN stored in an Account object. Therefore, we add a validatePIN operation to class Account. Note that we specify a return type of Boolean for the authenticateUser and validatePIN operations. Each operation returns a value 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).

[Page 301]

Figure 6.34 lists several additional verb phrases for class BankDatabase: "retrieves an account balance," "credits a deposit amount to an account" and "debits a withdrawal amount from an account." Like "authenticates a user," these remaining phrases refer to services that the database must provide to the ATM, because the database holds all the account data used to authenticate a user and perform ATM transactions. However, objects of class Account actually perform the operations to which these phrases refer. Thus, we assign an operation to both class BankDatabase and class Account to correspond to each of these phrases. Recall from Section 3.11 that, because a bank account contains sensitive information, we do not allow the ATM to access accounts directly. The database acts as an intermediary between the ATM and the account data, thus preventing unauthorized access. As we will see in Section 7.12, class ATM invokes the operations of class BankDatabase, each of which in turn invokes the operation with the same name in class Account.

The phrase "retrieves an account balance" suggests that classes BankDatabase and Account each need a getBalance operation. However, recall that we created two attributes in class Account to represent a balanceavailableBalance and totalBalance. A balance inquiry requires access to both balance attributes so that it can display them to the user, but a withdrawal needs to check only the value of availableBalance. To allow objects in the system to obtain each balance attribute individually, we add operations getAvailableBalance and getTotalBalance to the third compartment of classes BankDatabase and Account (Fig. 6.35). We specify a return type of Double for each of these operations, because the balance attributes which they retrieve are of type Double.

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

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

Operations of Class Screen

Class Screen "displays a message to the user" at various times in an ATM session. All visual output occurs through the screen of the ATM. The requirements document describes many types of messages (e.g., a welcome message, an error message, a thank-you message) that the screen displays to the user. The requirements document also indicates that the screen displays prompts and menus to the user. However, a prompt is really just a message describing what the user should input next, and a menu is essentially a type of prompt consisting of a series of messages (i.e., menu options) displayed consecutively. Therefore, rather than assign class Screen an individual operation to display each type of message, prompt and menu, we simply create one operation that can display any message specified by a parameter. We place this operation (displayMessage) in the third compartment of class Screen in our class diagram (Fig. 6.35). Note that we do not worry about the parameter of this operation at this timewe model the parameter later in this section.

[Page 302]

Operations of Class Keypad

From the phrase "receives numeric input from the user" listed by class Keypad in Fig. 6.34, we conclude that class Keypad should perform a getInput operation. Because the ATM's keypad,

unlike a computer keyboard, contains only the numbers 09, 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 classit does not determine whether the value meets any specific criteria. Any class that uses this operation must verify that the user enters appropriate numbers, and if not, display error messages 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. Later in the book, you will learn how to examine inputs to determine if they are of particular types.]

Operations of Class `CashDispenser` and Class `DepositSlot`

Figure 6.34 lists "dispenses cash" for class `CashDispenser`. Therefore, we create operation `dispenseCash` and list it under class `CashDispenser` in Fig. 6.35. Class `CashDispenser` also "indicates whether it contains enough cash to satisfy a withdrawal request." Thus, we include `isSufficientCashAvailable`, an operation that returns a value of UML type `Boolean`, in class `CashDispenser`. Figure 6.34 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.]

Operations of Class `ATM`

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

Identifying and Modeling Operation Parameters

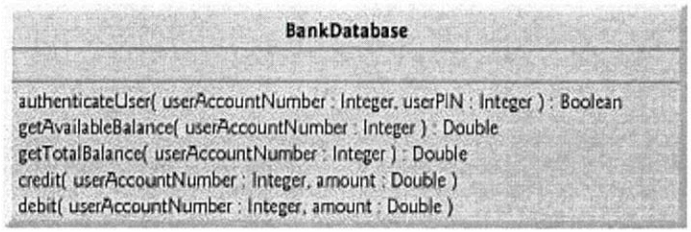
So far, we have not been concerned with the parameters of our operationswe have attempted to gain only a basic understanding of the operations of each class. Let's now take a closer look at some operation parameters. We identify an operation's parameters by examining what data the operation requires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

[Page 303]

Consider the `authenticateUser` operation of class `BankDatabase`. To authenticate a user, this operation must know the account number and PIN supplied by the user. Thus we specify that operation `authenticateUser` takes integer parameters `userAccountNumber` and `userPIN`, which the operation must compare to the account number and PIN of an `Account` object in the database. We prefix these parameter names with "user" to avoid confusion between the operation's parameter names and the attribute names that belong to class `Account`. We list these parameters in the class diagram in Fig. 6.36 that models only class `BankDatabase`. [*Note:* It is perfectly normal to model only one class in a class diagram. In this case, we are most concerned with examining the parameters of this one class in particular, so we omit the other classes. In class diagrams later in the case study, in which parameters are no longer the focus of our attention, we omit these parameters to save space. Remember, however, that the operations listed in these diagrams still have parameters.]

Figure 6.36. Class `BankDatabase` with operation parameters.

[View full size image]



Recall that the UML models each parameter in an operation's comma-separated parameter list by listing the parameter name, followed by a colon and the parameter type (in UML notation). Figure 6.36 thus specifies that operation `authenticateUser` takes two parameters`userAccountNumber` and `userPIN`, both of type `Integer`. When we implement the system in C++, we will 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. 6.36. In addition, operations `credit` and `debit` each require a `Double` parameter `amount` to specify the amount of money to be credited or debited, respectively.

[Page 304]

The class diagram in Fig. 6.37 models the parameters of class `Account`'s operations. Operation `validatePIN` requires only a `userPIN` parameter, which contains the user-specified PIN to be compared with the PIN associated with the account. Like their counterparts in class `BankDatabase`, operations `credit` and `debit` in class `Account` each require a `Double` parameter `amount` that indicates the amount of money involved in the operation. Operations `getAvailableBalance` and `getTotalBalance` in class `Account` require no additional data to perform their tasks. Note that class `Account`'s operations do not require an account number parametereach of these operations can be invoked only on a specific `Account` object, so including a parameter to specify an `Account` is unnecessary.

Figure 6.37. Class `Account` with operation parameters.
(This item is displayed on page 303 in the print version)

[View full size image]

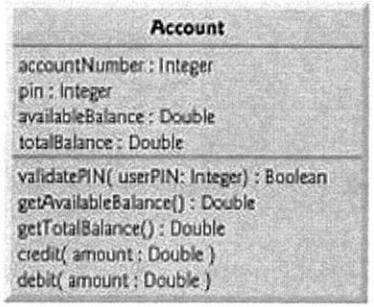
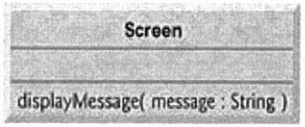


Figure 6.38 models class `Screen` with a parameter specified for operation `displayMessage`. This operation requires only a `String` parameter `message` that indicates the text to be displayed. Recall that the parameter types listed in our class diagrams are in UML notation, so the `String` type listed in Fig. 6.38 refers to the UML type. When we implement the system in C++, we will in fact use a C++ `string` object to represent this parameter.

Figure 6.38. Class `Screen` with operation parameters.

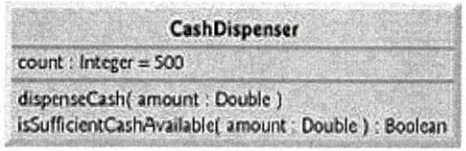
[View full size image]



The class diagram in Fig. 6.39 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.

Figure 6.39. Class `CashDispenser` with operation parameters.

[View full size image]



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

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

[Page 305]

Software Engineering Case Study Self-Review Exercises

6.1 Which of the following is not a behavior?

- **a.** reading data from a file
- **b.** printing output
- **c.** text output
- **d.** obtaining input from the user

PREV NEXT

[Page 377 (continued)]

7.12. (Optional) Software Engineering Case Study: Collaboration Among Objects in the ATM System

In this section, we concentrate on the collaborations (interactions) among objects in our ATM system. When two objects communicate with each other to accomplish a task, they are said to **collaborate**they 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 C++ via member-function calls.

In Section 6.18, we determined many of the operations of the classes in our system. In this section, we concentrate on the messages that invoke these operations. To identify the collaborations in the system, we return to the requirements document in Section 2.8. 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 and the remaining "Software Engineering Case Study" sections, 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 in the requirements document, we decide which objects in our system must interact to achieve the desired result. We identify one object as the sending object (i.e., the object that sends the message) and another as the receiving object (i.e., the object that offers that operation to clients of the class). We then select one of the receiving object's operations (identified in Section 6.18) 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 each object simply by using its class name preceded by an article ("a," "an" or "the")for example, "the `ATM`" refers to an object of class `ATM`.]

[Page 378]

Figure 7.27 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 are discussed in the requirements document. We list each collaboration involving a unique sender, message and recipient only once, even though the collaboration may occur several times during an ATM session. For example, the first row in Fig. 7.27 indicates that the `ATM` collaborates with the `Screen` whenever the `ATM` needs to display a message to the user.

Figure 7.27. Collaborations in the ATM system.

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

An object of class...	sends the message...	to an object of class...

Let's consider the collaborations in Fig. 7.27. 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 each of these tasks by sending a `displayMessage` message to the `Screen`. Both of these actions refer to the same collaboration between the `ATM` and the `Screen`, which is already listed in Fig. 7.27. 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 directlyonly the user's `Account` (i.e., the `Account` that contains the account number specified by the user) can access the user's PIN to authenticate the user. Figure 7.27 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 have already accounted for these collaborations. 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 `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 measures of the user's balance at the same time, the `BalanceInquiry` sends a `displayMessage` message to the `Screen`.

A `Withdrawal` sends a series of `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 menu selection. Next, the `Withdrawal` determines whether the requested withdrawal amount is less than or equal to the user's account balance. The `Withdrawal` can obtain the amount of money available in the user's account 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`. Recall that debiting funds from an `Account` 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

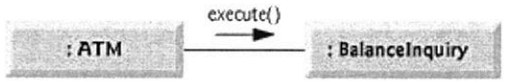
Now that we have identified a set of possible collaborations between the objects in our ATM system, let us graphically model these interactions using the UML. The UML provides several types of **interaction diagrams** that model the behavior of a system by modeling how objects interact with one another. The **communication diagram** emphasizes which objects participate in collaborations. [*Note:* Communication diagrams were called **collaboration diagrams** in earlier versions of the UML.] 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 7.28 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 the name of each object 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 member function) belonging to the receiving objectthink of the name as a service that the receiving object provides to sending objects (its "clients").

Figure 7.28. Communication diagram of the ATM executing a balance inquiry.

[View full size image]



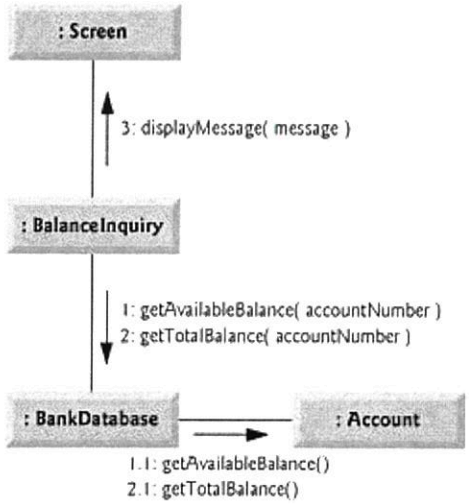
The solid filled arrow in Fig. 7.28 represents a messageor **synchronous call**in the UML and a function call in C++. This arrow indicates that the flow of control is from the sending object (the `ATM`) to the receiving object (a `BalanceInquiry`). Since this is a synchronous call, the sending object may not 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. For example, in Fig. 7.28, the `ATM` calls member function `execute` of a `BalanceInquiry` and may not 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 controlit would continue sending additional messages immediately following the asynchronous call. Asynchronous calls often can be implemented in C++ using platform-specific libraries provided with your compiler. Such techniques are beyond the scope of this book.]

Sequence of Messages in a Communication Diagram

Figure 7.29 shows a communication diagram that models the interactions among objects in the system 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. 7.29 begin after the `ATM` sends an `execute` message to a `BalanceInquiry` (i.e., the interaction modeled in Fig. 7.28). 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 C++ function call)the `BalanceInquiry` passes attribute `accountNumber` with its messages to the `BankDatabase` to indicate which `Account`'s balance information to retrieve. Recall from Fig. 6.33 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.

Figure 7.29. Communication diagram for executing a balance inquiry.

[View full size image]



Note, however, that Fig. 7.29 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. 7.29 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`.

[Page 382]

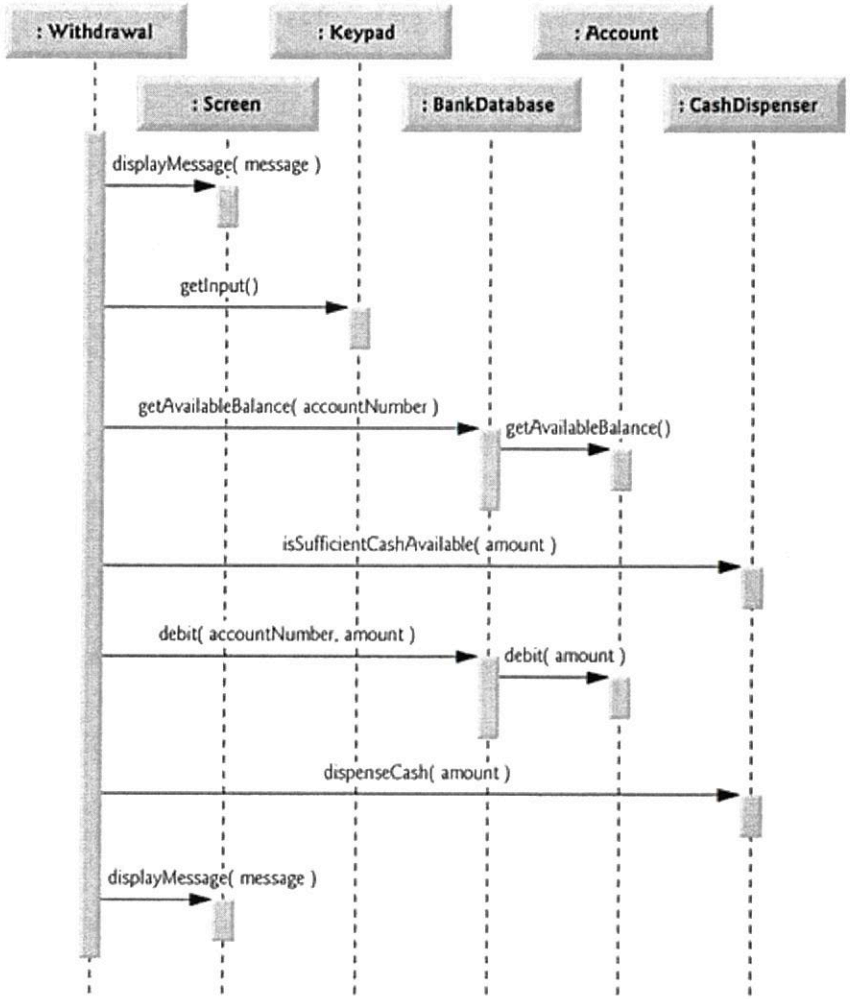
Sequence Diagrams

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 7.30 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 typically occur along an object's lifeline in chronological order from top to bottoman action near the top typically happens before one near the bottom.

[Page 383]

Figure 7.30. Sequence diagram that models a `Withdrawal` executing.
(This item is displayed on page 382 in the print version)

[View full size image]



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 arrowsthe 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. 7.30 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 have already modeled the control logic involved in a `Withdrawal` in the activity diagram of Fig. 5.28, so we do not show this logic in the sequence diagram of Fig. 7.30. 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. For information on how to model control logic in a sequence diagram, please refer to the Web resources and recommended readings listed at the end of Section 2.8.

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 is enough cash available, the `Withdrawal` decreases the balance of the user's account (i.e., both the `totalBalance` and the `availableBalance`) by sending a `debit` message to the `BankDatabase`. The `BankDatabase` responds by sending a `debit` message to the user's `Account`. Finally, the `Withdrawal` sends a `dispenseCash` message to the `CashDispenser` and a `displayMessage` message to the `Screen`, telling the user to remove the cash from the machine.

We have identified the collaborations among objects in the ATM system and modeled some of these collaborations using UML interaction diagramsboth communication diagrams and sequence diagrams. In the next "Software Engineering Case Study" section (Section 9.12), we enhance the structure of our model to complete a preliminary object-oriented design, then we begin implementing the ATM system.

Software Engineering Case Study Self-Review Exercises

7.1 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

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

[Page 384]

7.3 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.

Answers to Software Engineering Case Study Self-Review Exercises

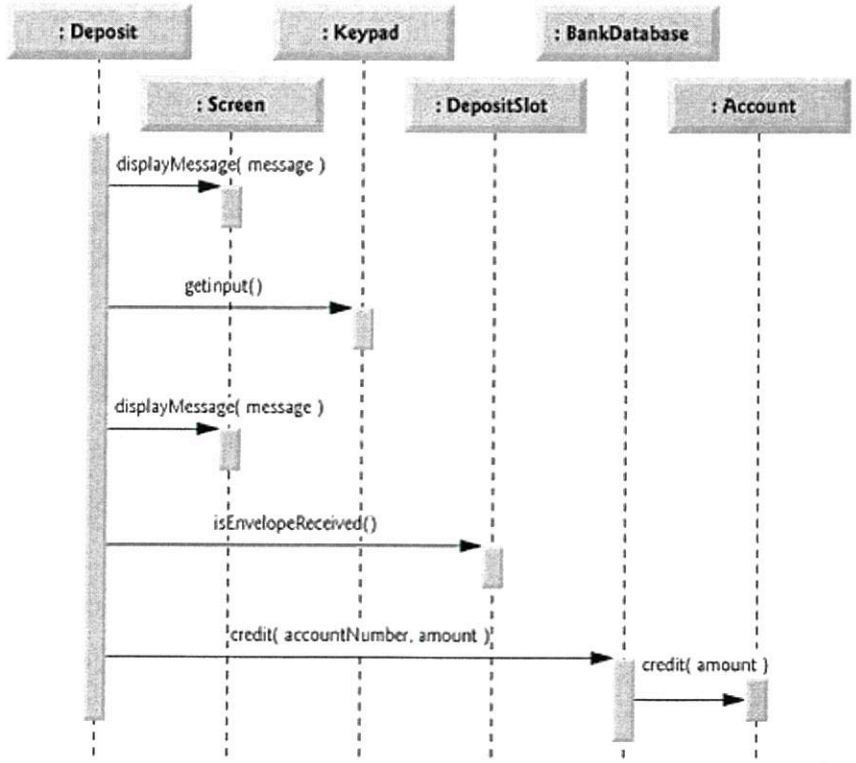
7.1 c.

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

7.3 Figure 7.31 presents a sequence diagram that models the interactions between objects in the ATM system that occur when a `Deposit` executes successfully. Figure 7.31 indicates that 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`.

Figure 7.31. Sequence diagram that models a `Deposit` executing.

[View full size image]



PREV NEXT

[Page 509 (continued)]

9.12. (Optional) Software Engineering Case Study: Starting to Program the Classes of the ATM System

In the "Software Engineering Case Study" sections in Chapters 17, we introduced the fundamentals of object orientation and developed an object-oriented design for our ATM system. Earlier in this chapter, we discussed many of the details of programming with C++ classes. We now begin implementing our object-oriented design in C++. At the end of this section, we show how to convert class diagrams to C++ header files. In the final "Software Engineering Case Study" section (Section 13.10), we modify the header files to incorporate the object-oriented concept of inheritance. We present the full C++ code implementation in Appendix G.

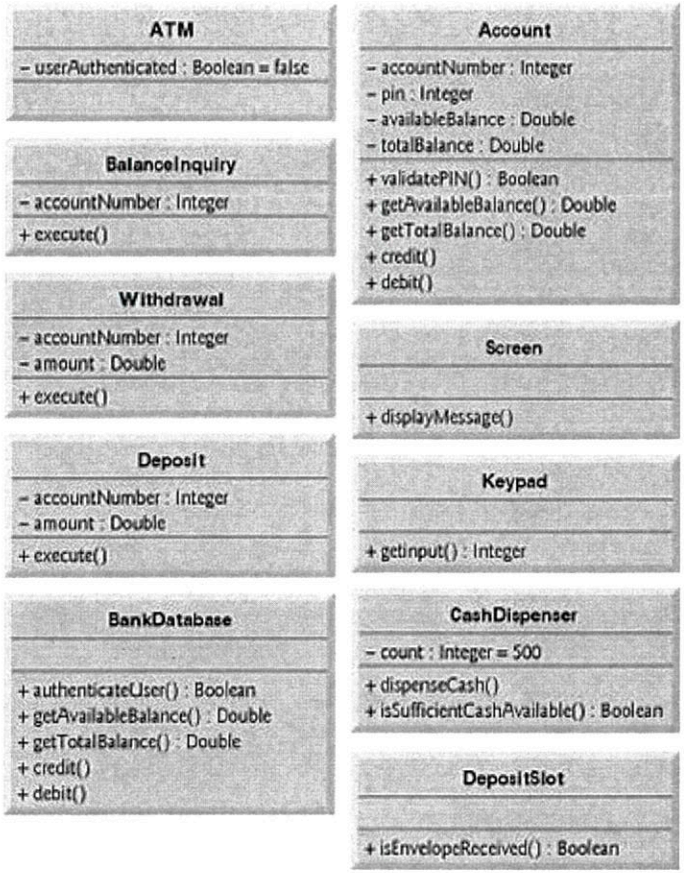
Visibility

We now apply access specifiers to the members of our classes. In Chapter 3, we introduced access specifiers `public` and `private`. Access specifiers determine the **visibility** or accessibility of an object's attributes and operations to other objects. Before we can begin implementing our design, we must consider which attributes and operations of our classes should be `public` and which should be `private`.

In Chapter 3, we observed that data members normally should be `private` and that member functions invoked by clients of a given class should be `public`. Member functions that are called only by other member functions of the class as "utility functions," however, normally should be `private`. The UML employs **visibility markers** for modeling the visibility of attributes and operations. Public visibility is indicated by placing a plus sign (+) before an operation or an attribute; a minus sign () indicates private visibility. Figure 9.20 shows our updated class diagram with visibility markers included. [*Note:* We do not include any operation parameters in Fig. 9.20. This is perfectly normal. Adding visibility markers does not affect the parameters already modeled in the class diagrams of Figs. 6.226.25.]

Figure 9.20. Class diagram with visibility markers.
(This item is displayed on page 510 in the print version)

[View full size image]

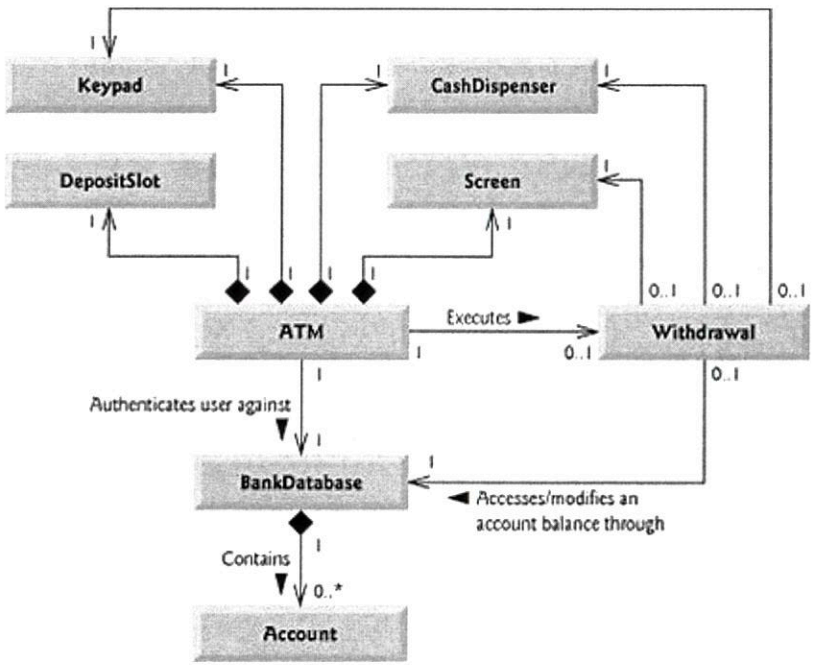


Navigability

Before we begin implementing our design in C++, we introduce an additional UML notation. The class diagram in Fig. 9.21 further refines the relationships among classes in the ATM system by adding navigability arrows to the association lines. **Navigability arrows** (represented as arrows with stick arrowheads in the class diagram) indicate in which direction an association can be traversed and are based on the collaborations modeled in communication and sequence diagrams (see Section 7.12). When implementing a system designed using the UML, programmers use navigability arrows to help determine which objects need references or pointers to other objects. For example, the navigability arrow pointing from class `ATM` to class `BankDatabase` indicates that we can navigate from the former to the latter, thereby enabling the `ATM` to invoke the `BankDatabase`'s operations. However, since Fig. 9.21 does not contain a navigability arrow pointing from class `BankDatabase` to class `ATM`, the `BankDatabase` cannot access the `ATM`'s operations. Note that associations in a class diagram that have navigability arrows at both ends or do not have navigability arrows at all indicate **bidirectional navigability**navigation can proceed in either direction across the association.

Figure 9.21. Class diagram with navigability arrows.
(This item is displayed on page 511 in the print version)

[View full size image]



Like the class diagram of Fig. 3.23 the class diagram of Fig. 9.21 omits classes `BalanceInquiry` and `Deposit` to keep the diagram simple. The navigability of the associations in which these classes participate closely parallels the navigability of class `Withdrawal`'s associations. Recall from Section 3.11 that `BalanceInquiry` has an association with class `Screen`. We can navigate from class `BalanceInquiry` to class `Screen` along this association, but we cannot navigate from class `Screen` to class `BalanceInquiry`. Thus, if we were to model class `BalanceInquiry` in Fig. 9.21, we would place a navigability arrow at class `Screen`'s end of this association. Also recall that class `Deposit` associates with classes `Screen`, `Keypad` and `DepositSlot`. We can navigate from class `Deposit` to each of these classes, but not vice versa. We therefore would place navigability arrows at the `Screen`, `Keypad` and `DepositSlot` ends of these associations. [*Note:* We model these additional classes and associations in our final class diagram in Section 13.10, after we have simplified the structure of our system by incorporating the object-oriented concept of inheritance.]

[Page 511]

Implementing the ATM System from Its UML Design

We are now ready to begin implementing the ATM system. We first convert the classes in the diagrams of Fig. 9.20 and Fig. 9.21 into C++ header files. This code will represent the "skeleton" of the system. In Chapter 13, we modify the header files to incorporate the object-oriented concept of inheritance. In Appendix G, ATM Case Study Code, we present the complete working C++ code for our model.

PREV NEXT

[Page 736 (continued)]

13.10. (Optional) Software Engineering Case Study: Incorporating Inheritance into the ATM System

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

In Section 3.11, we encountered the problem of representing a financial transaction in the system. Rather than create one class to represent all transaction types, we decided to create three individual transaction classes`BalanceInquiry`, `Withdrawal` and `Deposit`to represent the transactions that the ATM system can perform. Figure 13.26 shows the attributes and operations of these classes. Note that they have one attribute (`accountNumber`) and one operation (`execute`) in common. Each class requires attribute `accountNumber` to specify the account to which the transaction applies. Each class contains operation `execute`, which the `ATM` invokes to perform the transaction. Clearly, `BalanceInquiry`, `Withdrawal` and `Deposit` represent *types of* transactions. Figure 13.26 reveals commonality among the transaction classes, so using inheritance to factor out the common features seems appropriate for designing these classes. We place the common functionality in base class `TRansaction` and derive classes `BalanceInquiry`, `Withdrawal` and `Deposit` from `transaction` (Fig. 13.27).

Figure 13.26. Attributes and operations of classes `BalanceInquiry`, `Withdrawal` and `Deposit`.

[View full size image]

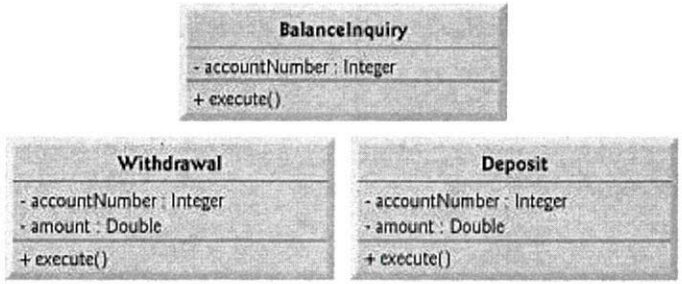
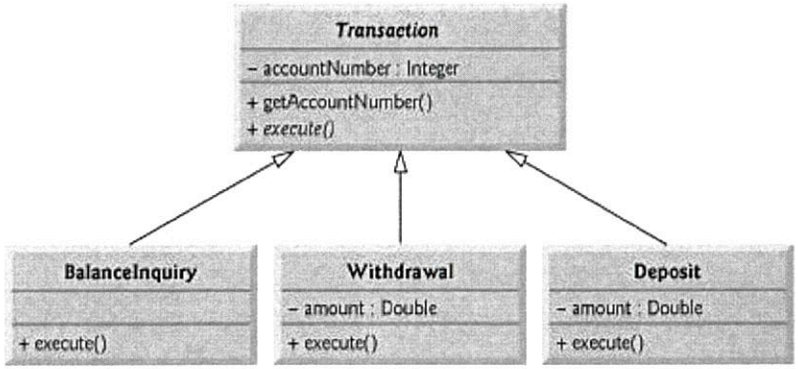


Figure 13.27. Class diagram modeling generalization relationship between base class `TRansaction` and derived classes `BalanceInquiry`, `Withdrawal` and `Deposit`.

(This item is displayed on page 737 in the print version)

[View full size image]



[Page 737]

The UML specifies a relationship called a **generalization** to model inheritance. Figure 13.27 is the class diagram that models the inheritance relationship between base class `transaction` and its three derived classes. The arrows with triangular hollow arrowheads indicate that classes `BalanceInquiry`, `Withdrawal` and `Deposit` are derived from class `transaction`. Class `transaction` is said to be a generalization of its derived classes. The derived classes are said to be **specializations** of class `TRansaction`.

Classes `BalanceInquiry`, `Withdrawal` and `Deposit` share integer attribute `accountNumber`, so we factor out this common attribute and place it in base class `TRansaction`. We no longer list `accountNumber` in the second compartment of each derived class, because the three derived classes inherit this attribute from `transaction`. Recall, however, that derived classes cannot access `private` attributes of a base class. We therefore include `public` member function `getAccountNumber` in class `TRansaction`. Each derived class inherits this member function, enabling the derived class to access its `accountNumber` as needed to execute a transaction.

According to Fig. 13.26, classes `BalanceInquiry`, `Withdrawal` and `Deposit` also share operation `execute`, so base class `transaction` should contain `public` member function `execute`. However, it does not make sense to implement `execute` in class `transaction`, because the functionality that this member function provides depends on the specific type of the actual transaction. We therefore declare member function `execute` as a pure `virtual` function in base class `transaction`. This makes `transaction` an abstract class and forces any class derived from `transaction` that must be a concrete class (i.e., `BalanceInquiry`, `Withdrawal` and `Deposit`) to implement pure `virtual` member function `execute` to make the derived class concrete. The UML requires that we place abstract class names (and pure `virtual` functions**abstract operations** in the UML) in italics, so `transaction` and its member function `execute` appear in italics in Fig. 13.27. Note that operation `execute` is not italicized in derived classes `BalanceInquiry`, `Withdrawal` and `Deposit`. Each derived class overrides base class `transaction`'s `execute` member function with an appropriate implementation. Note that Fig. 13.27 includes operation execute in the third compartment of classes `BalanceInquiry`, `Withdrawal` and `Deposit`, because each class has a different concrete implementation of the overridden member function.

[Page 738]

As you learned in this chapter, a derived class can inherit interface or implementation from a base class. Compared to a hierarchy designed for implementation inheritance, one designed for interface inheritance tends to have its functionality lower in the hierarchya base class signifies one or more functions that should be defined by each class in the hierarchy, but the individual derived classes provide their own implementations of the function(s). The inheritance hierarchy designed for the `ATM` system takes advantage of this type of inheritance, which provides the `ATM` with an elegant way to execute all transactions "in the general." Each class derived from `transaction` inherits some implementation details (e.g., data member `accountNumber`), but the primary benefit of incorporating inheritance into our system is that the derived classes share a common interface (e.g., pure `virtual` member function `execute`). The `ATM` can aim a `transaction` pointer at any transaction, and when the `ATM` invokes `execute` tHRough this pointer, the version of `execute` appropriate to that transaction (i.e., implemented in that derived class's `.cpp` file) runs automatically. For example, suppose a user chooses to perform a balance inquiry. The `ATM` aims a `transaction` pointer at a new object of class `BalanceInquiry`, which the C++ compiler allows because a `BalanceInquiry` *is a* `transaction`. When the `ATM` uses this pointer to invoke `execute`, `BalanceInquiry`'s version of `execute` is called.

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

As you learned earlier in the chapter, an abstract class like `TRansaction` is one for which the programmer never intends to instantiate objects. An abstract class simply declares common

attributes and behaviors for its derived classes in an inheritance hierarchy. Class `TRansaction` defines the concept of what it means to be a transaction that has an account number and executes. You may wonder why we bother to include pure `virtual` member function `execute` in class `TRansaction` if `execute` lacks a concrete implementation. Conceptually, we include this member function because it is the defining behavior of all transactionsexecuting. Technically, we must include member function `execute` in base class `TRansaction` so that the `ATM` (or any other class) can polymorphically invoke each derived class's overridden version of this function through a `TRansaction` pointer or reference.

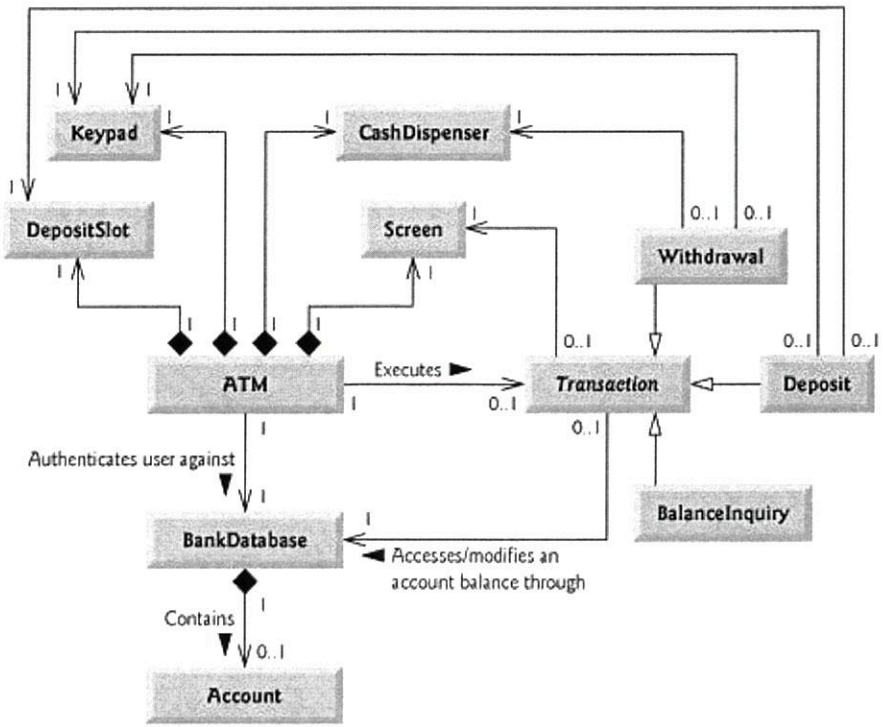
Derived classes `BalanceInquiry`, `Withdrawal` and `Deposit` inherit attribute `accountNumber` from base class `transaction`, but classes `Withdrawal` and `Deposit` contain the additional attribute `amount` that distinguishes them from class `BalanceInquiry`. Classes `Withdrawal` and `Deposit` require this additional attribute to store the amount of money that the user wishes to withdraw or deposit. Class `BalanceInquiry` has no need for such an attribute and requires only an account number to execute. Even though two of the three `transaction` derived classes share this attribute, we do not place it in base class `transaction`we place only features common to *all* the derived classes in the base class, so derived classes do not inherit unnecessary attributes (and operations).

Figure 13.28 presents an updated class diagram of our model that incorporates inheritance and introduces class `TRansaction`. We model an association between class `ATM` and class `transaction` to show that the `ATM`, at any given moment, either is executing a transaction or is not (i.e., zero or one objects of type `TRansaction` exist in the system at a time). Because a `Withdrawal` is a type of `TRansaction`, we no longer draw an association line directly between class `ATM` and class `Withdrawal`derived class `Withdrawal` inherits base class `transaction`'s association with class `ATM`. Derived classes `BalanceInquiry` and `Deposit` also inherit this association, which replaces the previously omitted associations between classes `BalanceInquiry` and `Deposit` and class `ATM`. Note again the use of triangular hollow arrowheads to indicate the specializations of class `TRansaction`, as indicated in Fig. 13.27.

[Page 739]

Figure 13.28. Class diagram of the `ATM` system (incorporating inheritance). Note that abstract class name `transaction` appears in italics.

[View full size image]



We also add an association between class `TRansaction` and the `BankDatabase` (Fig. 13.28). All `TRansactions` require a reference to the `BankDatabase` so they can access and modify account information. Each `TRansaction` derived class inherits this reference, so we no longer model the association between class `Withdrawal` and the `BankDatabase`. Note that the association between class `transaction` and the `BankDatabase` replaces the previously omitted associations between classes `BalanceInquiry` and `Deposit` and the `BankDatabase`.

We include an association between class `transaction` and the `Screen` because all `transactions` display output to the user via the `Screen`. Each derived class inherits this association. Therefore, we no longer include the association previously modeled between `Withdrawal` and the `Screen`. Class `Withdrawal` still participates in associations with the `CashDispenser` and the `Keypad`, howeverthese associations apply to derived class `Withdrawal` but not to derived classes `BalanceInquiry` and `Deposit`, so we do not move these associations to base class `transaction`.

[Page 740]

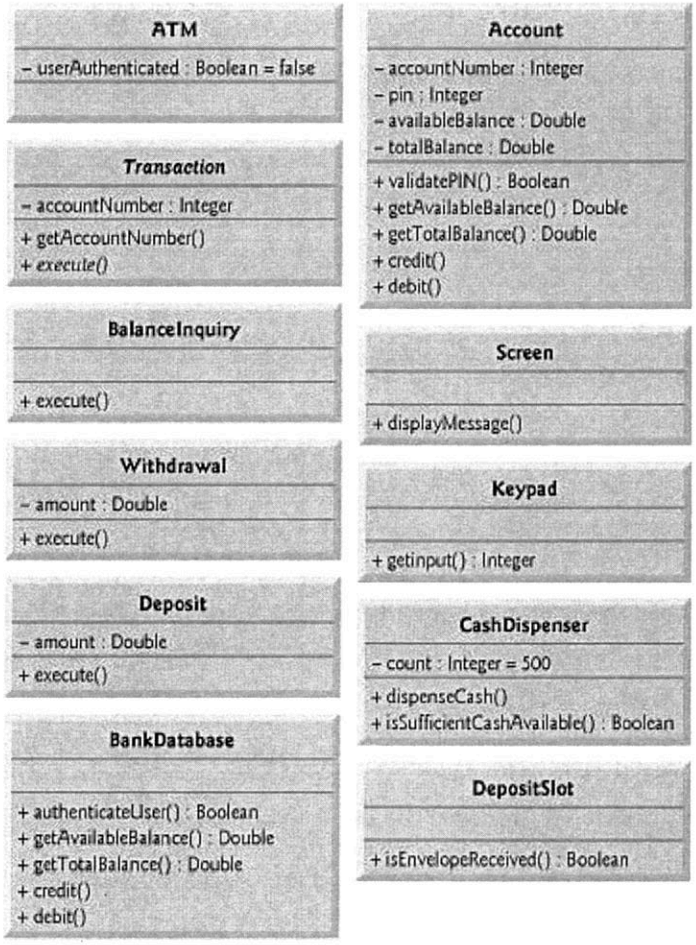
Our class diagram incorporating inheritance (Fig. 13.28) also models `Deposit` and `BalanceInquiry`. We show associations between `Deposit` and both the `DepositSlot` and the `Keypad`. Note that class `BalanceInquiry` takes part in no associations other than those inherited from class `transactiona` `BalanceInquiry` interacts only with the `BankDatabase` and the `Screen`.

The class diagram of Fig. 9.20 showed attributes and operations with visibility markers. Now we present a modified class diagram in Fig. 13.29 that includes abstract base class `transaction`. This abbreviated diagram does not show inheritance relationships (these appear in Fig. 13.28), but instead shows the attributes and operations after we have employed inheritance in our system. Note that abstract class name `transaction` and abstract operation name `execute` in class `transaction` appear in italics. To save space, as we did in Fig. 4.24, we do not include those attributes shown by associations in Fig. 13.28we do, however, include them in the C++ implementation in Appendix G. We also omit all operation parameters, as we did in Fig. 9.20incorporating inheritance does not affect the parameters already modeled in Figs. 6.226.25.

[Page 741]

Figure 13.29. Class diagram after incorporating inheritance into the system.
(This item is displayed on page 740 in the print version)

[View full size image]



Software Engineering Observation 13.12

A complete class diagram shows all the associations among classes and all the attributes and operations for each class. When the number of class attributes, operations and associations is substantial (as in Fig. 13.28 and Fig. 13.29), a good practice that promotes readability is to divide this information between two class diagramsone focusing on associations and the other on attributes and operations. However, when examining classes modeled in this fashion, it is crucial to consider both class diagrams to get a complete view of the classes. For example, one must refer to Fig. 13.28 to observe the inheritance relationship between `transaction` *and its derived classes that is omitted from Fig. 13.29.*