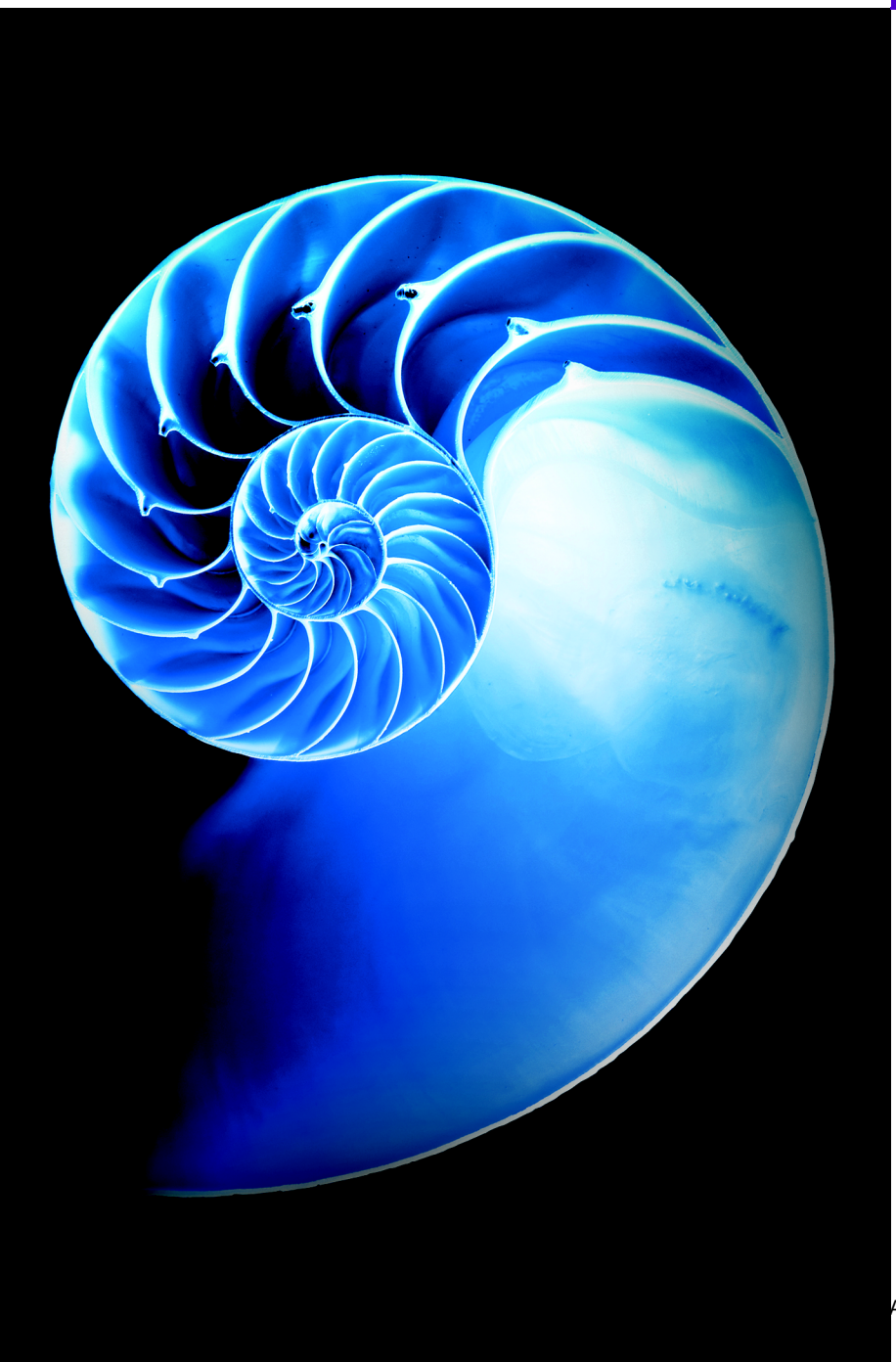
ATM Case Study Part 2: Implementing the Design

34



Objectives

In this chapter you'll:

- Incorporate inheritance into the design of the ATM.
- Incorporate polymorphism into the design of the ATM.
- Fully implement in Java the UML-based object-oriented design of the ATM software.
- Study a detailed code walkthrough of the ATM software system that explains the implementation issues.

Outline

34.1 Introduction

In Chapter 33, we developed an object-oriented design for our ATM system. We now implement our object-oriented design in Java. In Section 34.2, we show how to convert class diagrams to Java code. In Section 34.3, we tune the design with inheritance and polymorphism. Then we present a full Java code implementation of the ATM software in Section 34.4. The code is carefully commented and the discussions of the implementation are thorough and precise. Studying this application provides the opportunity for you to see a more substantial application of the kind you're likely to encounter in industry.

34.2 Starting to Program the Classes of the ATM System

[*Note:* **This section may be read after Chapter 8.**]

Visibility

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

We've observed that attributes normally should be `private` and that methods invoked by clients of a given class should be `public`. Methods that are called as "utility methods" only by other methods of the same class normally should be `private`. The UML employs **visibility markers** for modeling the visibility of attributes and operations. Public visibility is indicated by placing a plus sign (+) before an operation or an attribute, whereas a minus sign (–) indicates private visibility. Figure 34.1 shows our updated class diagram with visibility markers included. [*Note:* We do not include any operation parameters in Fig. 34.1—this is perfectly normal. Adding visibility markers does not affect the parameters already modeled in the class diagrams of Figs. 33.17–33.21.]

Navigability

Before we begin implementing our design in Java, we introduce an additional UML notation. The class diagram in Fig. 34.2 further refines the relationships among classes in the ATM system by adding navigability arrows to the association lines. **Navigability arrows**

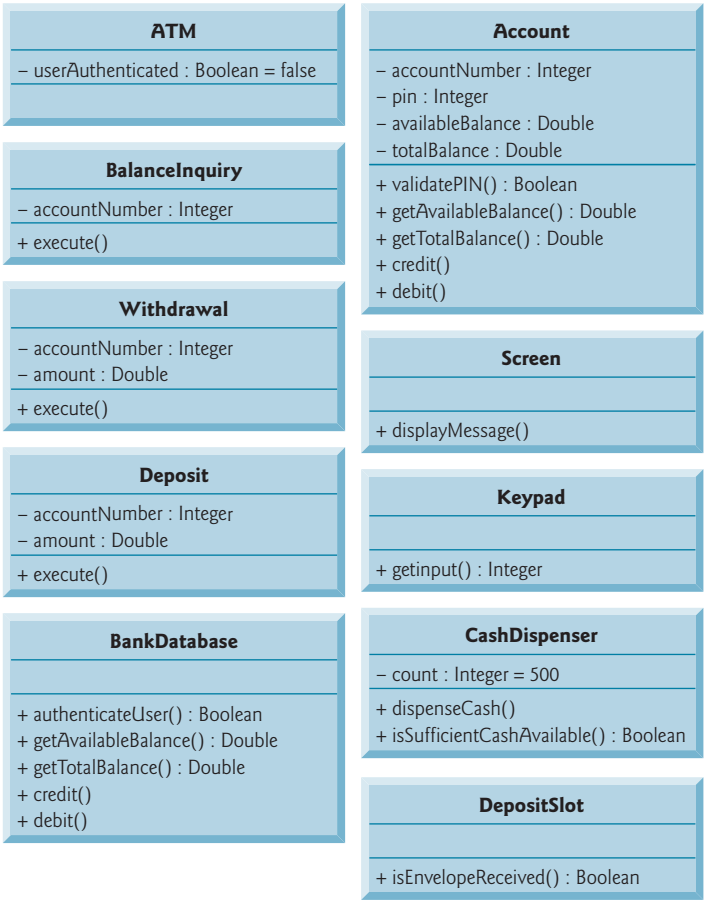


Fig. 34.1 | Class diagram with visibility markers.

(represented as arrows with stick (→) arrowheads in the class diagram) indicate the direction in which an association can be traversed. When implementing a system designed using the UML, you use navigability arrows to determine which objects need references to other objects. For example, the navigability arrow pointing from class `ATM` to class `BankDatabase` indicates that we can navigate from the former to the latter, thereby enabling the `ATM` to invoke the `BankDatabase`'s operations. However, since Fig. 34.2 does *not* contain a navigability arrow pointing from class `BankDatabase` to class `ATM`, the `BankDatabase` cannot access the `ATM`'s operations. Associations in a class diagram that have navigability arrows at both ends or have none at all indicate **bidirectional navigability**—navigation can proceed in either direction across the association.

Like the class diagram of Fig. 33.10, that of Fig. 34.2 omits classes `BalanceInquiry` and `Deposit` for simplicity. The navigability of the associations in which these classes participate closely parallels that of class `Withdrawal`. Recall from Section 33.3 that `BalanceInquiry` has an association with class `Screen`. We can navigate from class `BalanceInquiry` to class `Screen` along this association, but we cannot navigate from class `Screen` to class

`BalanceInquiry`. Thus, if we were to model class `BalanceInquiry` in Fig. 34.2, we would place a navigability arrow at class `Screen`'s end of this association. Also recall that class `Deposit` associates with classes `Screen`, `Keypad` and `DepositSlot`. We can navigate from class `Deposit` to each of these classes, but *not* vice versa. We therefore would place navigability arrows at the `Screen`, `Keypad` and `DepositSlot` ends of these associations. [*Note:* We model these additional classes and associations in our final class diagram in Section 34.3, after we've simplified the structure of our system by incorporating the object-oriented concept of inheritance.]

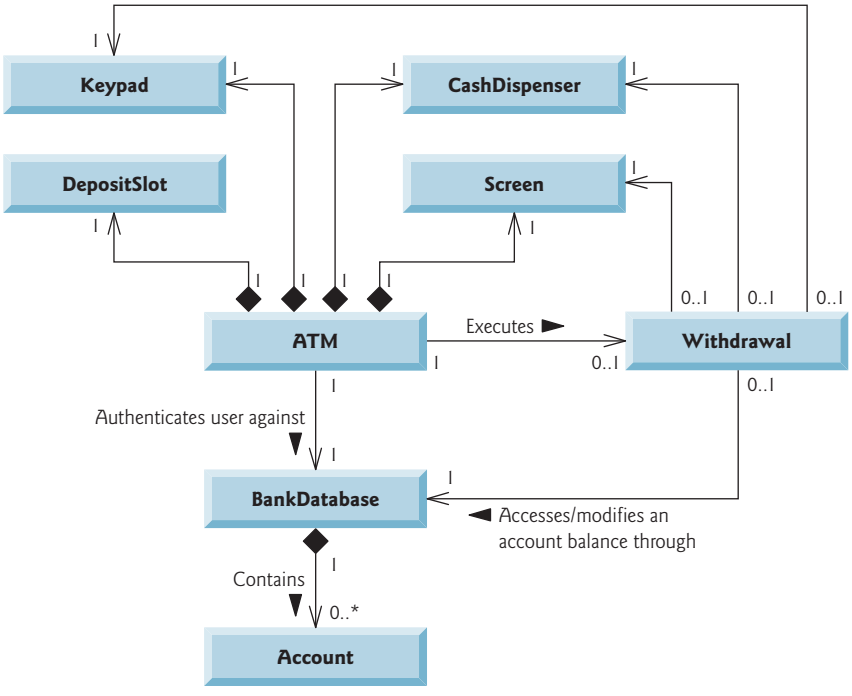


Fig. 34.2 | Class diagram with navigability arrows.

Implementing the ATM System from Its UML Design

We're now ready to begin implementing the ATM system. We first convert the classes in the diagrams of Fig. 34.1 and Fig. 34.2 into Java code. The code will represent the "skeleton" of the system. In Section 34.3, we modify the code to incorporate inheritance. In Section 34.4, we present the complete working Java code for our model.

As an example, we develop the code from our design of class `Withdrawal` in Fig. 34.1. We use this figure to determine the attributes and operations of the class. We use the UML model in Fig. 34.2 to determine the associations among classes. We follow the following four guidelines for each class:

1. Use the name located in the first compartment to declare the class as a `public` class with an empty no-argument constructor. We include this constructor simply as a placeholder to remind us that *most classes will indeed need custom construc-*

tors. In Section 34.4, when we complete a working version of this class, we'll add arguments and code the body of the constructor as needed. For example, class `Withdrawal` yields the code in Fig. 34.3. If we find that the class's instance variables require only default initialization, then we'll remove the empty no-argument constructor because it's unnecessary.

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

Fig. 34.3 | Java code for class `Withdrawal` based on Figs. 34.1–34.2.

2. Use the attributes located in the second compartment to declare the instance variables. For example, the `private` attributes `accountNumber` and `amount` of class `Withdrawal` yield the code in Fig. 34.4. [*Note:* The constructor of the complete working version of this class will assign values to these attributes.]

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

Fig. 34.4 | Java code for class `Withdrawal` based on Figs. 34.1–34.2.

3. Use the associations described in the class diagram to declare the references to other objects. For example, according to Fig. 34.2, `Withdrawal` can access one object of class `Screen`, one object of class `Keypad`, one object of class `CashDispenser` and one object of class `BankDatabase`. This yields the code in Fig. 34.5. [*Note:* The constructor of the complete working version of this class will initialize these instance variables with references to actual objects.]

```
1  // Class Withdrawal represents an ATM withdrawal transaction
2  public class Withdrawal {
3     // attributes
4     private int accountNumber; // account to withdraw funds from
5     private double amount; // amount to withdraw
6
7     // references to associated objects
8     private Screen screen; // ATM's screen
9     private Keypad keypad; // ATM's keypad
```

Fig. 34.5 | Java code for class `Withdrawal` based on Figs. 34.1–34.2. (Part 1 of 2.)

```
10      private CashDispenser cashDispenser; // ATM's cash dispenser
11      private BankDatabase bankDatabase; // account info database
12
13      // no-argument constructor
14      public Withdrawal() { }
15   }
```

Fig. 34.5 | Java code for class `Withdrawal` based on Figs. 34.1–34.2. (Part 2 of 2.)

4. Use the operations located in the third compartment of Fig. 34.1 to declare the shells of the methods. If we have not yet specified a return type for an operation, we declare the method with return type `void`. Refer to the class diagrams of Figs. 33.17–33.21 to declare any necessary parameters. For example, adding the `public` operation `execute` in class `Withdrawal`, which has an empty parameter list, yields the code in Fig. 34.6. [*Note:* We code the bodies of methods when we implement the complete system in Section 34.4.]

This concludes our discussion of the basics of generating classes from UML diagrams.

```
1   // Class Withdrawal represents an ATM withdrawal transaction
2   public class Withdrawal {
3      // attributes
4      private int accountNumber; // account to withdraw funds from
5      private double amount; // amount to withdraw
6
7      // references to associated objects
8      private Screen screen; // ATM's screen
9      private Keypad keypad; // ATM's keypad
10     private CashDispenser cashDispenser; // ATM's cash dispenser
11     private BankDatabase bankDatabase; // account info database
12
13     // no-argument constructor
14     public Withdrawal() { }
15
16     // operations
17     public void execute() { }
18  }
```

Fig. 34.6 | Java code for class `Withdrawal` based on Figs. 34.1–34.2.

Self-Review Exercises for Section 34.2

34.1 State whether the following statement is *true* or *false*, and if *false*, explain why: If an attribute of a class is marked with a minus sign (-) in a class diagram, the attribute is not directly accessible outside the class.

34.2 In Fig. 34.2, the association between the `ATM` and the `Screen` indicates that:
a) we can navigate from the `Screen` to the `ATM`
b) we can navigate from the `ATM` to the `Screen`
c) Both (a) and (b); the association is bidirectional
d) None of the above

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

34.3 Incorporating Inheritance and Polymorphism into the ATM System

[*Note:* **This section may be read after Chapter 10.**]
We now revisit our ATM system design to see how it might benefit from inheritance. To apply inheritance, we first look for *commonality among classes* in the system. We create an inheritance hierarchy to model similar (yet not identical) classes in a more elegant and efficient manner. We then modify our class diagram to incorporate the new inheritance relationships. Finally, we demonstrate how our updated design is translated into Java code.

In Section 33.3, we encountered the problem of representing a financial transaction in the system. Rather than create one class to represent all transaction types, we decided to create three individual transaction classes—`BalanceInquiry`, `Withdrawal` and `Deposit`—to represent the transactions that the ATM system can perform. Figure 34.7 shows the attributes and operations of classes `BalanceInquiry`, `Withdrawal` and `Deposit`. These classes have one attribute (`accountNumber`) and one operation (`execute`) in common. Each class requires attribute `accountNumber` to specify the account to which the transaction applies. Each class contains operation `execute`, which the `ATM` invokes to perform the transaction. Clearly, `BalanceInquiry`, `Withdrawal` and `Deposit` represent *types of* transactions. Figure 34.7 reveals commonality among the transaction classes, so using inheritance to factor out the common features seems appropriate for designing classes `BalanceInquiry`, `Withdrawal` and `Deposit`. We place the common functionality in a superclass, `Transaction`, that classes `BalanceInquiry`, `Withdrawal` and `Deposit` extend.

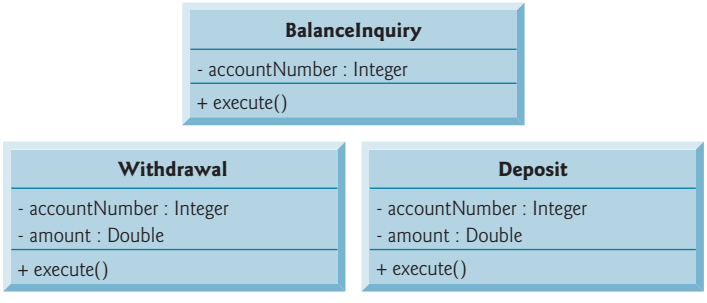


Fig. 34.7 | Attributes and operations of `BalanceInquiry`, `Withdrawal` and `Deposit`.

Generalization

The UML specifies a relationship called a **generalization** to model inheritance. Figure 34.8 is the class diagram that models the generalization of superclass `Transaction` and subclasses `BalanceInquiry`, `Withdrawal` and `Deposit`. The arrows with triangular hollow arrowheads indicate that classes `BalanceInquiry`, `Withdrawal` and `Deposit` extend class `Transaction`. Class `Transaction` is said to be a generalization of classes `BalanceInquiry`, `Withdrawal` and `Deposit`. Class `BalanceInquiry`, `Withdrawal` and `Deposit` are said to be **specializations** of class `Transaction`.

Classes `BalanceInquiry`, `Withdrawal` and `Deposit` share integer attribute `accountNumber`, so we *factor out* this *common attribute* and place it in superclass `Transaction`. We no longer list `accountNumber` in the second compartment of each subclass, because the

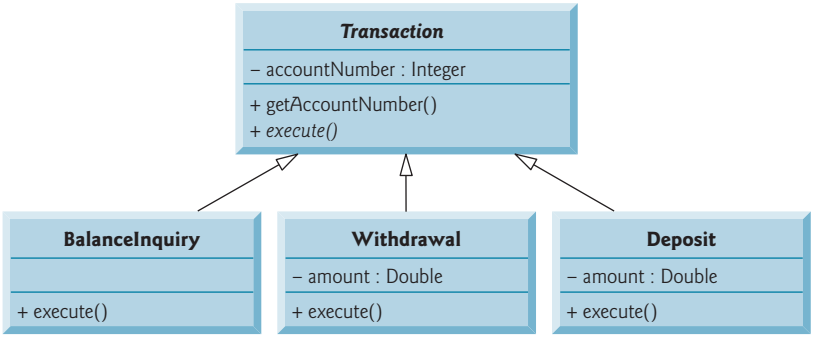


Fig. 34.8 | Class diagram modeling generalization of superclass `Transaction` and subclasses `BalanceInquiry`, `Withdrawal` and `Deposit`. Abstract class names (e.g., `Transaction`) and method names (e.g., `execute` in class `Transaction`) appear in italics.

three subclasses *inherit* this attribute from `Transaction`. Recall, however, that subclasses cannot directly access `private` attributes of a superclass. We therefore include `public` method `getAccountNumber` in class `Transaction`. Each subclass will inherit this method, enabling the subclass to access its `accountNumber` as needed to execute a transaction.

According to Fig. 34.7, classes `BalanceInquiry`, `Withdrawal` and `Deposit` also share operation `execute`, so we placed `public` method `execute` in superclass `Transaction`. However, it does *not* make sense to implement `execute` in class `Transaction`, because the functionality that this method provides *depends on the type of the actual transaction*. We therefore declare method `execute` as `abstract` in superclass `Transaction`. Any class that contains at least one abstract method must also be declared `abstract`. This forces any subclass of `Transaction` that must be a *concrete* class (i.e., `BalanceInquiry`, `Withdrawal` and `Deposit`) to implement method `execute`. The UML requires that we place abstract class names (and abstract methods) in italics, so `Transaction` and its method `execute` appear in italics in Fig. 34.8. Method `execute` is *not* italicized in subclasses `BalanceInquiry`, `Withdrawal` and `Deposit`. Each subclass overrides superclass `Transaction`'s `execute` method with a concrete implementation that performs the steps appropriate for completing that type of transaction. Figure 34.8 includes operation `execute` in the third compartment of classes `BalanceInquiry`, `Withdrawal` and `Deposit`, because each class has a different concrete implementation of the overridden method.

Processing `Transaction`s Polymorphically

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

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

The ATM could execute transactions of the new type using the current code, because it executes all transactions *polymorphically* using a general Transaction reference.

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

Additional Attribute of Classes `Withdrawal` *and* `Deposit`

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

Class Diagram with `Transaction` *Hierarchy Incorporated*

Figure 34.9 presents an updated class diagram of our model that incorporates inheritance and introduces class Transaction. We model an association between class ATM and class Transaction to show that the ATM, at any given moment, either is executing a transaction or is not (i.e., zero or one objects of type Transaction exist in the system at a time). Because a Withdrawal is a type of Transaction, we no longer draw an association line directly between class ATM and class Withdrawal. Subclass Withdrawal inherits superclass Transaction's association with class ATM. Subclasses BalanceInquiry and Deposit inherit this association, too, so the previously omitted associations between ATM and classes BalanceInquiry and Deposit no longer exist either.

We also add an association between class Transaction and the BankDatabase (Fig. 34.9). All Transactions require a reference to the BankDatabase so they can access and modify account information. Because each Transaction subclass inherits this reference, we no longer model the association between class Withdrawal and the BankDatabase. Similarly, the previously omitted associations between the BankDatabase and classes BalanceInquiry and Deposit no longer exist.

We show an association between class Transaction and the Screen. All Transactions display output to the user via the Screen. Thus, we no longer include the association previously modeled between Withdrawal and the Screen, although Withdrawal still participates in associations with the CashDispenser and the Keypad. Our class diagram incor-

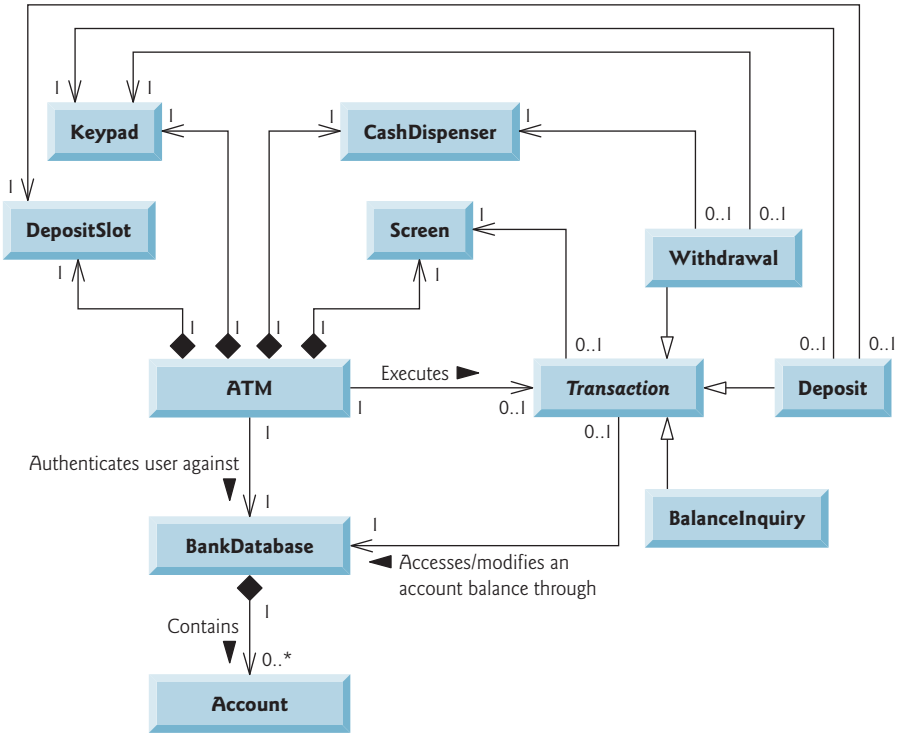


Fig. 34.9 | Class diagram of the ATM system (incorporating inheritance). The abstract class name `Transaction` appears in italics.

porating inheritance also models `Deposit` and `BalanceInquiry`. We show associations between `Deposit` and both the `DepositSlot` and the `Keypad`. Class `BalanceInquiry` takes part in no associations other than those inherited from class `Transaction`—a `BalanceInquiry` needs to interact only with the `BankDatabase` and with the `Screen`.

Figure 34.1 showed attributes and operations with visibility markers. Now in Fig. 34.10 we present a modified class diagram that incorporates inheritance. This abbreviated diagram does not show inheritance relationships, but instead shows the attributes and methods after we've employed inheritance in our system. To save space, as we did in Fig. 33.12, we do not include those attributes shown by associations in Fig. 34.9—we do, however, include them in the Java implementation in Section 34.4. We also omit all operation parameters, as we did in Fig. 34.1—incorporating inheritance does not affect the parameters already modeled in Figs. 33.17–33.21.

Software Engineering Observation 34.1

A complete class diagram shows all the associations among classes and all the attributes and operations for each class. When the number of class attributes, methods and associations is substantial (as in Figs. 34.9 and 34.10), a good practice that promotes readability is to divide this information between two class diagrams—one focusing on associations and the other on attributes and methods.

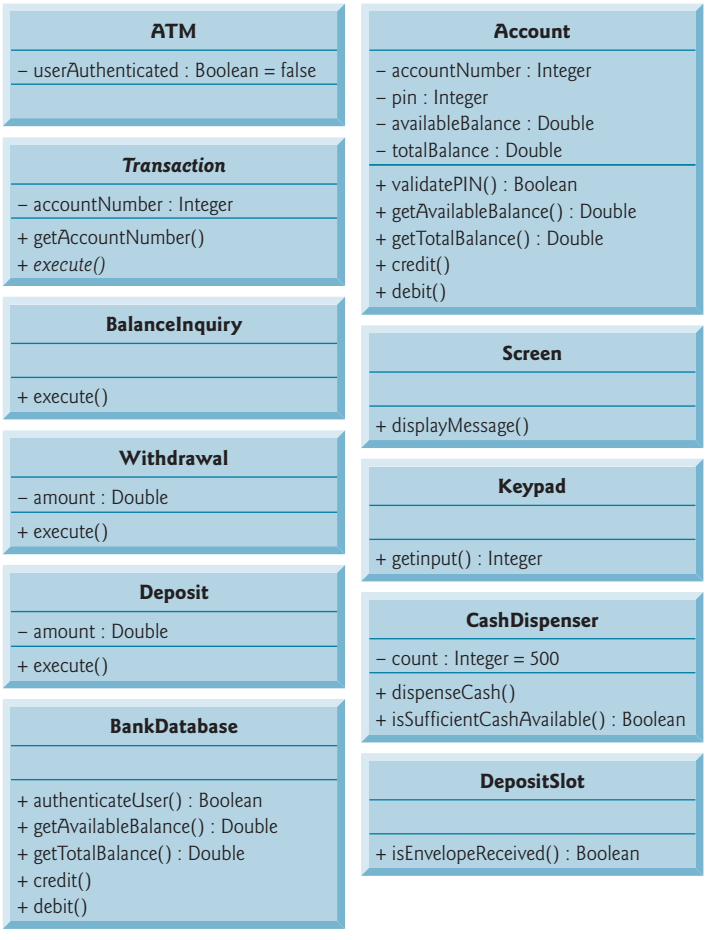


Fig. 34.10 | Class diagram with attributes and operations (incorporating inheritance). The abstract class name `Transaction` and the abstract method name `execute` in class `Transaction` appear in italics.

Implementing the ATM System Design (Incorporating Inheritance)

In Section 34.2, we began implementing the ATM system design in Java code. We now incorporate inheritance, using class `Withdrawal` as an example.

1. If a class `A` is a generalization of class `B`, then class `B` extends class `A` in the class declaration. For example, abstract superclass `Transaction` is a generalization of class `Withdrawal`. Figure 34.11 shows the declaration of class `Withdrawal`.

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

Fig. 34.11 | Java code for shell of class `Withdrawal`.

2. If class `A` is an abstract class and class `B` is a subclass of class `A`, then class `B` must implement the *abstract* methods of class `A` if class `B` is to be a *concrete* class. For example, class `Transaction` contains abstract method `execute`, so class `Withdrawal` must implement this method if we want to instantiate a `Withdrawal` object. Figure 34.12 is the Java code for class `Withdrawal` from Fig. 34.9 and Fig. 34.10. Class `Withdrawal` inherits field `accountNumber` from superclass `Transaction`, so `Withdrawal` does not need to declare this field. Class `Withdrawal` also inherits references to the `Screen` and the `BankDatabase` from its superclass `Transaction`, so we do not include these references in our code. Figure 34.10 specifies attribute `amount` and operation `execute` for class `Withdrawal`. Line 5 of Fig. 34.12 declares a field for attribute `amount`. Lines 13–14 declare the shell of a method for operation `execute`. Recall that subclass `Withdrawal` must provide a concrete implementation of the `abstract` method `execute` in superclass `Transaction`. The `keypad` and `cashDispenser` references (lines 6–7) are fields derived from `Withdrawal`'s associations in Fig. 34.9. The constructor in the complete working version of this class will initialize these references to actual objects.

Software Engineering Observation 34.2

Several UML modeling tools can convert UML-based designs into Java code, speeding the implementation process considerably. For more information on these tools, visit our UML Resource Center at `www.deitel.com/UML/`.

```
1  // Withdrawal.java
2  // Generated using the class diagrams in Fig. 34.9 and Fig. 34.10
3  public class Withdrawal extends Transaction {
4     // attributes
5     private double amount; // amount to withdraw
6     private Keypad keypad; // reference to keypad
7     private CashDispenser cashDispenser; // reference to cash dispenser
8
9     // no-argument constructor
10    public Withdrawal() { }
11
12    // method overriding execute
13    @Override
14    public void execute() { }
15 }
```

Fig. 34.12 | Java code for class `Withdrawal` based on Figs. 34.9 and 34.10.

Congratulations on completing the case study's design portion! We implement the ATM system in Java code in Section 34.4. We recommend that you carefully read the code and its description. The code is abundantly commented and precisely follows the design with which you're now familiar. The accompanying description is carefully written to guide your understanding of the implementation based on the UML design. Mastering this code is a wonderful culminating accomplishment after studying Sections 33.2–33.7 and 34.2–34.3.

Self-Review Exercises for Section 34.3

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

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

34.6 Write Java code to begin implementing the design for class `Transaction` specified in Figs. 34.9 and 34.10. Be sure to include `private` reference-type attributes based on class `Transaction`'s associations. Also be sure to include `public` *get* methods that provide access to any of these `private` attributes that the subclasses require to perform their tasks.

34.4 ATM Case Study Implementation

This section contains the complete implementation of the ATM system. We consider the classes in the order in which we identified them in Section 33.3—`ATM`, `Screen`, `Keypad`, `CashDispenser`, `DepositSlot`, `Account`, `BankDatabase`, `Transaction`, `BalanceInquiry`, `Withdrawal` and `Deposit`.

We apply the guidelines from Sections 34.2–34.3 to code these classes based on their UML class diagrams of Figs. 34.9 and 34.10. To develop the bodies of methods, we refer to the activity diagrams in Section 33.5 and the communication and sequence diagrams presented in Section 33.7. Our ATM design does *not* specify all the program logic and may not specify all the attributes and operations required to complete the ATM implementation. This is a *normal* part of the object-oriented design process. As we implement the system, we complete the program logic and add attributes and behaviors as necessary to construct the ATM system specified by the requirements document in Section 33.2.

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

34.4.1 Class ATM

Class `ATM` (Fig. 34.13) represents the ATM as a whole. Lines 5–11 implement the class's attributes. We determine all but one of these attributes from the UML class diagrams of Figs. 34.9 and 34.10. We implement the UML `Boolean` attribute `userAuthenticated` in Fig. 34.10 as a `boolean` in Java (line 5). Line 6 declares an attribute not found in our UML design—an `int` attribute `currentAccountNumber` that keeps track of the account number of the current authenticated user. We'll soon see how the class uses this attribute. Lines 7–11 declare reference-type attributes corresponding to the `ATM` class's associations modeled in the class diagram of Fig. 34.9. These attributes allow the ATM to access its parts (i.e., its `Screen`, `Keypad`, `CashDispenser` and `DepositSlot`) and interact with the bank's account-information database (i.e., a `BankDatabase` object).

```
1  // ATM.java
2  // Represents an automated teller machine
3
4  public class ATM {
5     private boolean userAuthenticated; // whether user is authenticated
6     private int currentAccountNumber; // current user's account number
7     private Screen screen; // ATM's screen
8     private Keypad keypad; // ATM's keypad
9     private CashDispenser cashDispenser; // ATM's cash dispenser
10    private DepositSlot depositSlot; // ATM's deposit slot
11    private BankDatabase bankDatabase; // account information database
12
13    // constants corresponding to main menu options
14    private static final int BALANCE_INQUIRY = 1;
15    private static final int WITHDRAWAL = 2;
16    private static final int DEPOSIT = 3;
17    private static final int EXIT = 4;
18
19    // no-argument ATM constructor initializes instance variables
20    public ATM() {
21       userAuthenticated = false; // user is not authenticated to start
22       currentAccountNumber = 0; // no current account number to start
23       screen = new Screen(); // create screen
24       keypad = new Keypad(); // create keypad
25       cashDispenser = new CashDispenser(); // create cash dispenser
26       depositSlot = new DepositSlot(); // create deposit slot
27       bankDatabase = new BankDatabase(); // create acct info database
28    }
29
30    // start ATM
31    public void run() {
32       // welcome and authenticate user; perform transactions
33       while (true) {
34          // loop while user is not yet authenticated
35          while (!userAuthenticated) {
36             screen.displayMessageLine("\nWelcome!");
37             authenticateUser(); // authenticate user
38          }
39
40          performTransactions(); // user is now authenticated
41          userAuthenticated = false; // reset before next ATM session
42          currentAccountNumber = 0; // reset before next ATM session
43          screen.displayMessageLine("\nThank you! Goodbye!");
44       }
45    }
46
47    // attempts to authenticate user against database
48    private void authenticateUser() {
49       screen.displayMessage("\nPlease enter your account number: ");
50       int accountNumber = keypad.getInput(); // input account number
51       screen.displayMessage("\nEnter your PIN: "); // prompt for PIN
52       int pin = keypad.getInput(); // input PIN
53
```

Fig. 34.13 | Class ATM represents the ATM. (Part 1 of 3.)

```
54          // set userAuthenticated to boolean value returned by database
55          userAuthenticated =
56             bankDatabase.authenticateUser(accountNumber, pin);
57
58          // check whether authentication succeeded
59          if (userAuthenticated) {
60             currentAccountNumber = accountNumber; // save user's account #
61          }
62          else {
63             screen.displayMessageLine(
64                "Invalid account number or PIN. Please try again.");
65          }
66       }
67
68       // display the main menu and perform transactions
69       private void performTransactions() {
70          // local variable to store transaction currently being processed
71          Transaction currentTransaction = null;
72
73          boolean userExited = false; // user has not chosen to exit
74
75          // loop while user has not chosen option to exit system
76          while (!userExited) {
77             // show main menu and get user selection
78             int mainMenuSelection = displayMainMenu();
79
80             // decide how to proceed based on user's menu selection
81             switch (mainMenuSelection) {
82                // user chose to perform one of three transaction types
83                case BALANCE_INQUIRY:
84                case WITHDRAWAL:
85                case DEPOSIT:
86
87                   // initialize as new object of chosen type
88                   currentTransaction =
89                      createTransaction(mainMenuSelection);
90
91                   currentTransaction.execute(); // execute transaction
92                   break;
93                case EXIT: // user chose to terminate session
94                   screen.displayMessageLine("\nExiting the system...");
95                   userExited = true; // this ATM session should end
96                   break;
97                default: // user did not enter an integer from 1-4
98                   screen.displayMessageLine(
99                      "\nYou did not enter a valid selection. Try again.");
100                  break;
101            }
102         }
103      }
104
```

Fig. 34.13 | Class ATM represents the ATM. (Part 2 of 3.)

```
105    // display the main menu and return an input selection
106    private int displayMainMenu() {
107       screen.displayMessageLine("\nMain menu:");
108       screen.displayMessageLine("1 - View my balance");
109       screen.displayMessageLine("2 - Withdraw cash");
110       screen.displayMessageLine("3 - Deposit funds");
111       screen.displayMessageLine("4 - Exit\n");
112       screen.displayMessage("Enter a choice: ");
113       return keypad.getInput(); // return user's selection
114    }
115
116    // return object of specified Transaction subclass
117    private Transaction createTransaction(int type) {
118       Transaction temp = null; // temporary Transaction variable
119
120       // determine which type of Transaction to create
121       switch (type) {
122          case BALANCE_INQUIRY: // create new BalanceInquiry transaction
123             temp = new BalanceInquiry(
124                currentAccountNumber, screen, bankDatabase);
125             break;
126          case WITHDRAWAL: // create new Withdrawal transaction
127             temp = new Withdrawal(currentAccountNumber, screen,
128                bankDatabase, keypad, cashDispenser);
129             break;
130          case DEPOSIT: // create new Deposit transaction
131             temp = new Deposit(currentAccountNumber, screen,
132                bankDatabase, keypad, depositSlot);
133             break;
134       }
135
136       return temp; // return the newly created object
137    }
138 }
```

Fig. 34.13 | Class `ATM` represents the ATM. (Part 3 of 3.)

Lines 14–17 declare integer constants that correspond to the four options in the ATM's main menu (i.e., balance inquiry, withdrawal, deposit and exit). Lines 20–28 declare the constructor, which initializes the class's attributes. When an `ATM` object is first created, no user is authenticated, so line 21 initializes `userAuthenticated` to `false`. Likewise, line 22 initializes `currentAccountNumber` to `0` because there's no current user yet. Lines 23–26 instantiate new objects to represent the ATM's parts. Recall that class `ATM` has composition relationships with classes `Screen`, `Keypad`, `CashDispenser` and `DepositSlot`, so class `ATM` is responsible for their creation. Line 27 creates a new `BankDatabase`. [*Note:* If this were a real ATM system, the `ATM` class would receive a reference to an existing database object created by the bank. However, in this implementation we're only simulating the bank's database, so class `ATM` creates the `BankDatabase` object with which it interacts.]

`ATM` Method `run`

The class diagram of Fig. 34.10 does not list any operations for class `ATM`. We now implement one operation (i.e., `public` method) in class `ATM` that allows an external client of the

class (i.e., class ATMCaseStudy) to tell the ATM to run. ATM method run (lines 31–45) uses an infinite loop to repeatedly welcome a user, attempt to authenticate the user and, if authentication succeeds, allow the user to perform transactions. After an authenticated user performs the desired transactions and chooses to exit, the ATM resets itself, displays a goodbye message to the user and restarts the process. We use an infinite loop here to simulate the fact that an ATM appears to run continuously until the bank turns it off (an action beyond the user's control). An ATM user has the option to exit the system but not the ability to turn off the ATM completely.

Authenticating a User

In method run's infinite loop, lines 35–38 cause the ATM to repeatedly welcome and attempt to authenticate the user as long as the user has not been authenticated (i.e., !userAuthenticated is true). Line 36 invokes method displayMessageLine of the ATM's screen to display a welcome message. Like Screen method displayMessage designed in the case study, method displayMessageLine (declared in lines 11–13 of Fig. 34.14) displays a message to the user, but this method also outputs a newline after the message. We've added this method during implementation to give class Screen's clients more control over the placement of displayed messages. Line 37 invokes class ATM's private utility method authenticateUser (declared in lines 48–66) to attempt to authenticate the user.

We refer to the requirements document to determine the steps necessary to authenticate the user before allowing transactions to occur. Line 49 of method authenticateUser invokes method displayMessage of the screen to prompt the user to enter an account number. Line 50 invokes method getInput of the keypad to obtain the user's input, then stores the integer value entered by the user in a local variable accountNumber. Method authenticateUser next prompts the user to enter a PIN (line 51), and stores the PIN input by the user in a local variable pin (line 52). Next, lines 55–56 attempt to authenticate the user by passing the accountNumber and pin entered by the user to the bankDatabase's authenticateUser method. Class ATM sets its userAuthenticated attribute to the boolean value returned by this method—userAuthenticated becomes true if authentication succeeds (i.e., accountNumber and pin match those of an existing Account in bankDatabase) and remains false otherwise. If userAuthenticated is true, line 60 saves the account number entered by the user (i.e., accountNumber) in the ATM attribute currentAccountNumber. The other ATM methods use this variable whenever an ATM session requires access to the user's account number. If userAuthenticated is false, lines 63–64 use the screen's displayMessageLine method to indicate that an invalid account number and/or PIN was entered and the user must try again. We set currentAccountNumber only after authenticating the user's account number and the associated PIN—if the database could not authenticate the user, currentAccountNumber remains 0.

After method run attempts to authenticate the user (line 37), if userAuthenticated is still false, the while loop in lines 35–38 executes again. If userAuthenticated is now true, the loop terminates and control continues with line 40, which calls class ATM's utility method performTransactions.

Performing Transactions

Method performTransactions (lines 69–103) carries out an ATM session for an authenticated user. Line 71 declares a local Transaction variable to which we'll assign a Balance-

`Inquiry`, `Withdrawal` or `Deposit` object representing the ATM transaction the user selected. We use a `Transaction` variable here to allow us to take advantage of polymorphism. Also, we name this variable after the *role name* included in the class diagram of Fig. 33.7—`currentTransaction`. Line 73 declares another local variable—a `boolean` called `userExited` that keeps track of whether the user has chosen to exit. This variable controls a `while` loop (lines 76–102) that allows the user to execute an unlimited number of transactions before choosing to exit. Within this loop, line 78 displays the main menu and obtains the user's menu selection by calling an `ATM` utility method `displayMainMenu` (declared in lines 106–114). This method displays the main menu by invoking methods of the `ATM`'s `screen` and returns a menu selection obtained from the user through the `ATM`'s `keypad`. Line 78 stores the user's selection returned by `displayMainMenu` in local variable `mainMenuSelection`.

After obtaining a main menu selection, method `performTransactions` uses a `switch` statement (lines 81–101) to respond to the selection appropriately. If `mainMenuSelection` is equal to any of the three integer constants representing transaction types (i.e., if the user chose to perform a transaction), lines 88–89 call utility method `createTransaction` (declared in lines 117–137) to return a newly instantiated object of the type that corresponds to the selected transaction. Variable `currentTransaction` is assigned the reference returned by `createTransaction`, then line 91 invokes method `execute` of this transaction to execute it. We'll discuss `Transaction` method `execute` and the three `Transaction` subclasses shortly. We assign the `Transaction` variable `currentTransaction` an object of one of the three `Transaction` subclasses so that we can execute transactions *polymorphically*. For example, if the user chooses to perform a balance inquiry, `mainMenuSelection` equals `BALANCE_INQUIRY`, leading `createTransaction` to return a `BalanceInquiry` object. Thus, `currentTransaction` refers to a `BalanceInquiry`, and invoking `currentTransaction.execute()` results in `BalanceInquiry`'s version of `execute` being called.

Creating a Transaction

Method `createTransaction` (lines 117–137) uses a `switch` statement to instantiate a new `Transaction` subclass object of the type indicated by the parameter `type`. Recall that method `performTransactions` passes `mainMenuSelection` to this method only when `mainMenuSelection` contains a value corresponding to one of the three transaction types. Therefore `type` is `BALANCE_INQUIRY`, `WITHDRAWAL` or `DEPOSIT`. Each `case` in the `switch` statement instantiates a new object by calling the appropriate `Transaction` subclass constructor. Each constructor has a unique parameter list, based on the specific data required to initialize the subclass object. A `BalanceInquiry` requires only the account number of the current user and references to the `ATM`'s `screen` and the `bankDatabase`. In addition to these parameters, a `Withdrawal` requires references to the `ATM`'s `keypad` and `cashDispenser`, and a `Deposit` requires references to the `ATM`'s `keypad` and `depositSlot`. We discuss the transaction classes in more detail in Sections 34.4.8–34.4.11.

Exiting the Main Menu and Processing Invalid Selections

After executing a transaction (line 91 in `performTransactions`), `userExited` remains `false` and lines 76–102 repeat, returning the user to the main menu. However, if a user does not perform a transaction and instead selects the main menu option to exit, line 95 sets `userExited` to `true`, causing the condition of the `while` loop (`!userExited`) to be-

come `false`. This `while` is the final statement of method `performTransactions`, so control returns to the calling method `run`. If the user enters an invalid main menu selection (i.e., not an integer from 1–4), lines 98–99 display an appropriate error message, `userExited` remains `false` and the user returns to the main menu to try again.

Awaiting the Next ATM User

When `performTransactions` returns control to method `run`, the user has chosen to exit the system, so lines 41–42 reset the `ATM`'s attributes `userAuthenticated` and `currentAccountNumber` to prepare for the next ATM user. Line 43 displays a goodbye message before the ATM starts over and welcomes the next user.

34.4.2 Class Screen

Class `Screen` (Fig. 34.14) represents the screen of the ATM and encapsulates all aspects of displaying output to the user. Class `Screen` approximates a real ATM's screen with a computer monitor and outputs text messages using standard console output methods `System.out.print`, `System.out.println` and `System.out.printf`. In this case study, we designed class `Screen` to have one operation—`displayMessage`. For greater flexibility in displaying messages to the `Screen`, we now declare three `Screen` methods—`displayMessage`, `displayMessageLine` and `displayDollarAmount`.

```
1  // Screen.java
2  // Represents the screen of the ATM
3
4  public class Screen {
5     // display a message without a carriage return
6     public void displayMessage(String message) {
7        System.out.print(message);
8     }
9
10    // display a message with a carriage return
11    public void displayMessageLine(String message) {
12       System.out.println(message);
13    }
14
15    // displays a dollar amount
16    public void displayDollarAmount(double amount) {
17       System.out.printf("$%,.2f", amount);
18    }
19 }
```

Fig. 34.14 | Class `Screen` represents the screen of the ATM.

Method `displayMessage` (lines 6–8) takes a `String` argument and prints it to the console. The cursor stays on the same line, making this method appropriate for displaying prompts to the user. Method `displayMessageLine` (lines 11–13) does the same using `System.out.println`, which outputs a newline to move the cursor to the next line. Finally, method `displayDollarAmount` (lines 16–18) outputs a properly formatted dollar amount (e.g., `$1,234.56`). Line 17 uses `System.out.printf` to output a `double` value formatted with commas to increase readability and two decimal places.

34.4.3 Class Keypad

Class `Keypad` (Fig. 34.15) represents the keypad of the ATM and is responsible for receiving all user input. Recall that we're simulating this hardware, so we use the computer's keyboard to approximate the keypad. We use class `Scanner` to obtain console input from the user. A computer keyboard contains many keys not found on the ATM's keypad. However, we assume that the user presses only the keys on the computer keyboard that also appear on the keypad—the keys numbered 0–9 and the *Enter* key.

```
1  // Keypad.java
2  // Represents the keypad of the ATM
3  import java.util.Scanner; // program uses Scanner to obtain user input
4
5  public class Keypad {
6     private Scanner input; // reads data from the command line
7
8     // no-argument constructor initializes the Scanner
9     public Keypad() {
10       input = new Scanner(System.in);
11    }
12
13    // return an integer value entered by user
14    public int getInput() {
15       return input.nextInt(); // we assume that user enters an integer
16    }
17 }
```

Fig. 34.15 | Class `Keypad` represents the ATM's keypad.

Line 6 declares `Scanner` variable `input` as an instance variable. Line 10 in the constructor creates a new `Scanner` object that reads input from the standard input stream (`System.in`) and assigns the object's reference to variable `input`. Method `getInput` (lines 14–16) invokes `Scanner` method `nextInt` (line 15) to return the next integer input by the user. [*Note:* Method `nextInt` can throw an `InputMismatchException` if the user enters non-integer input. Because the real ATM's keypad permits only integer input, we assume that no exception will occur and do not attempt to fix this problem. See Chapter 11, Exception Handling: A Deeper Look, for information on catching exceptions.] Recall that `nextInt` obtains all the input used by the ATM. `Keypad`'s `getInput` method simply returns the integer input by the user. If a client of class `Keypad` requires input that satisfies some criteria (i.e., a number corresponding to a valid menu option), the client must perform the error checking.

34.4.4 Class CashDispenser

Class `CashDispenser` (Fig. 34.16) represents the cash dispenser of the ATM. Line 6 declares constant `INITIAL_COUNT`, which indicates the initial count of bills in the cash dispenser when the ATM starts (i.e., `500`). Line 7 implements attribute `count` (modeled in Fig. 34.10), which keeps track of the number of bills remaining in the `CashDispenser` at any time. The constructor (lines 10–12) sets `count` to the initial count. `CashDispenser` has two `public` methods—`dispenseCash` (lines 15–18) and `isSufficientCashAvail-`

able (lines 21–30). The class trusts that a client (i.e., `Withdrawal`) calls `dispenseCash` only after establishing that sufficient cash is available by calling `isSufficientCashAvailable`. Thus, `dispenseCash` simply simulates dispensing the requested amount without checking whether sufficient cash is available.

```
1  // CashDispenser.java
2  // Represents the cash dispenser of the ATM
3
4  public class CashDispenser {
5     // the default initial number of bills in the cash dispenser
6     private final static int INITIAL_COUNT = 500;
7     private int count; // number of $20 bills remaining
8
9     // no-argument CashDispenser constructor initializes count to default
10    public CashDispenser() {
11       count = INITIAL_COUNT; // set count attribute to default
12    }
13
14    // simulates dispensing of specified amount of cash
15    public void dispenseCash(int amount) {
16       int billsRequired = amount / 20; // number of $20 bills required
17       count -= billsRequired; // update the count of bills
18    }
19
20    // indicates whether cash dispenser can dispense desired amount
21    public boolean isSufficientCashAvailable(int amount) {
22       int billsRequired = amount / 20; // number of $20 bills required
23
24       if (count >= billsRequired) {
25          return true; // enough bills available
26       }
27       else {
28          return false; // not enough bills available
29       }
30    }
31 }
```

Fig. 34.16 | Class `CashDispenser` represents the ATM's cash dispenser.

Method `isSufficientCashAvailable` has a parameter `amount` that specifies the amount of cash in question. Line 22 calculates the number of $20 bills required to dispense the specified `amount`. The ATM allows the user to choose only withdrawal amounts that are multiples of $20, so we divide `amount` by `20` to obtain the number of `billsRequired`. Lines 24–29 return `true` if the `CashDispenser`'s `count` is greater than or equal to `billsRequired` (i.e., enough bills are available) and `false` otherwise (i.e., not enough bills). For example, if a user wishes to withdraw $80 (i.e., `billsRequired` is `4`), but only three bills remain (i.e., `count` is `3`), the method returns `false`.

Method `dispenseCash` (lines 15–18) simulates cash dispensing. If our system were hooked up to a real hardware cash dispenser, this method would interact with the device to physically dispense cash. Our version of the method simply decreases the `count` of bills remaining by the number required to dispense the specified `amount`. It's the responsibility

of the client of the class (i.e., `Withdrawal`) to inform the user that cash has been dispensed—`CashDispenser` cannot interact directly with `Screen`.

34.4.5 Class `DepositSlot`

Class `DepositSlot` (Fig. 34.17) represents the ATM's deposit slot. Like class `CashDispenser`, class `DepositSlot` merely simulates the functionality of a real hardware deposit slot. `DepositSlot` has no attributes and only one method—`isEnvelopeReceived`—which indicates whether a deposit envelope was received.

Recall from the requirements document that the ATM allows the user up to two minutes to insert an envelope. The current version of method `isEnvelopeReceived` simply returns `true` immediately (line 8), because this is only a software simulation, and we assume that the user has inserted an envelope within the required time frame. If an actual hardware deposit slot were connected to our system, method `isEnvelopeReceived` might be implemented to wait for a maximum of two minutes to receive a signal from the hardware deposit slot indicating that the user has indeed inserted a deposit envelope. If `isEnvelopeReceived` were to receive such a signal within two minutes, the method would return `true`. If two minutes elapsed and the method still had not received a signal, then the method would return `false`.

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

Fig. 34.17 | Class `DepositSlot` represents the ATM's deposit slot.

34.4.6 Class `Account`

Class `Account` (Fig. 34.18) represents a bank account. Each `Account` has four attributes (modeled in Fig. 34.10)—`accountNumber`, `pin`, `availableBalance` and `totalBalance`. Lines 5–8 implement these attributes as `private` fields. Variable `availableBalance` represents the amount of funds available for withdrawal. Variable `totalBalance` represents the amount of funds available, plus the amount of deposited funds still pending confirmation or clearance.

```
1  // Account.java
2  // Represents a bank account
3
4  public class Account {
5     private int accountNumber; // account number
```

Fig. 34.18 | Class `Account` represents a bank account. (Part 1 of 2.)

```
6      private int pin; // PIN for authentication
7      private double availableBalance; // funds available for withdrawal
8      private double totalBalance; // funds available + pending deposits
9
10     // Account constructor initializes attributes
11     public Account(int theAccountNumber, int thePIN,
12        double theAvailableBalance, double theTotalBalance) {
13        accountNumber = theAccountNumber;
14        pin = thePIN;
15        availableBalance = theAvailableBalance;
16        totalBalance = theTotalBalance;
17     }
18
19     // determines whether a user-specified PIN matches PIN in Account
20     public boolean validatePIN(int userPIN) {
21        if (userPIN == pin) {
22           return true;
23        }
24        else {
25           return false;
26        }
27     }
28
29     // returns available balance
30     public double getAvailableBalance() {
31        return availableBalance;
32     }
33
34     // returns the total balance
35     public double getTotalBalance() {
36        return totalBalance;
37     }
38
39     // credits an amount to the account
40     public void credit(double amount) {
41        totalBalance += amount; // add to total balance
42     }
43
44     // debits an amount from the account
45     public void debit(double amount) {
46        availableBalance -= amount; // subtract from available balance
47        totalBalance -= amount; // subtract from total balance
48     }
49
50     // returns account number
51     public int getAccountNumber() {
52        return accountNumber;
53     }
54  }
```

Fig. 34.18 | Class `Account` represents a bank account. (Part 2 of 2.)

The `Account` class has a constructor (lines 11–17) that takes an account number, the PIN established for the account, the account's initial available balance and the account's

initial total balance as arguments. Lines 13–16 assign these values to the class's attributes (i.e., fields).

Method `validatePIN` (lines 20–27) determines whether a user-specified PIN (i.e., parameter `userPIN`) matches the PIN associated with the account (i.e., attribute `pin`). Recall that we modeled this method's parameter `userPIN` in Fig. 33.19. If the two PINs match, the method returns `true`; otherwise, it returns `false`.

Methods `getAvailableBalance` (lines 30–32) and `getTotalBalance` (lines 35–37) return the values of `double` attributes `availableBalance` and `totalBalance`, respectively.

Method `credit` (lines 40–42) adds an amount of money (i.e., parameter `amount`) to an `Account` as part of a deposit transaction. This method adds the `amount` only to attribute `totalBalance`. The money credited to an account during a deposit does *not* become available immediately, so we modify only the total balance. We assume that the bank updates the available balance appropriately at a later time. Our implementation of class `Account` includes only methods required for carrying out ATM transactions. Therefore, we omit the methods that some other bank system would invoke to add to attribute `availableBalance` (to confirm a deposit) or subtract from attribute `totalBalance` (to reject a deposit).

Method `debit` (lines 45–48) subtracts an amount of money (i.e., parameter `amount`) from an `Account` as part of a withdrawal transaction. This method subtracts the `amount` from *both* attribute `availableBalance` and attribute `totalBalance`, because a withdrawal affects *both* measures of an account balance.

Method `getAccountNumber` (lines 51–53) provides access to an `Account`'s `accountNumber`. We include this method in our implementation so that a client of the class (i.e., `BankDatabase`) can identify a particular `Account`. For example, `BankDatabase` contains many `Account` objects, and it can invoke this method on each of its `Account` objects to locate the one with a specific account number.

34.4.7 Class BankDatabase

Class `BankDatabase` (Fig. 34.19) models the bank's database with which the ATM interacts to access and modify a user's account information. We study database access in Chapter 24. For now we model the database as an array. An exercise in Chapter 24 asks you to reimplement this portion of the ATM using an actual database.

```
1  // BankDatabase.java
2  // Represents the bank account information database
3
4  public class BankDatabase {
5     private Account[] accounts; // array of Accounts
6
7     // no-argument BankDatabase constructor initializes accounts
8     public BankDatabase() {
9        accounts = new Account[2]; // just 2 accounts for testing
10       accounts[0] = new Account(12345, 54321, 1000.0, 1200.0);
11       accounts[1] = new Account(98765, 56789, 200.0, 200.0);
12    }
13
```

Fig. 34.19 | Class `BankDatabase` represents the bank's account information database. (Part 1 of 2.)

```
14      // retrieve Account object containing specified account number
15      private Account getAccount(int accountNumber) {
16         // loop through accounts searching for matching account number
17         for (Account currentAccount : accounts) {
18            // return current account if match found
19            if (currentAccount.getAccountNumber() == accountNumber) {
20               return currentAccount;
21            }
22         }
23
24         return null; // if no matching account was found, return null
25      }
26
27      // determine whether user-specified account number and PIN match
28      // those of an account in the database
29      public boolean authenticateUser(int userAccountNumber, int userPIN) {
30         // attempt to retrieve the account with the account number
31         Account userAccount = getAccount(userAccountNumber);
32
33         // if account exists, return result of Account method validatePIN
34         if (userAccount != null) {
35            return userAccount.validatePIN(userPIN);
36         }
37         else {
38            return false; // account number not found, so return false
39         }
40      }
41
42      // return available balance of Account with specified account number
43      public double getAvailableBalance(int userAccountNumber) {
44         return getAccount(userAccountNumber).getAvailableBalance();
45      }
46
47      // return total balance of Account with specified account number
48      public double getTotalBalance(int userAccountNumber) {
49         return getAccount(userAccountNumber).getTotalBalance();
50      }
51
52      // credit an amount to Account with specified account number
53      public void credit(int userAccountNumber, double amount) {
54         getAccount(userAccountNumber).credit(amount);
55      }
56
57      // debit an amount from Account with specified account number
58      public void debit(int userAccountNumber, double amount) {
59         getAccount(userAccountNumber).debit(amount);
60      }
61   }
```

Fig. 34.19 | Class `BankDatabase` represents the bank's account information database. (Part 2 of 2.)

We determine one reference-type attribute for class `BankDatabase` based on its composition relationship with class `Account`. Recall from Fig. 34.9 that a `BankDatabase` is composed of zero or more objects of class `Account`. Line 5 implements attribute `accounts`—an

array of Account objects—to implement this composition relationship. Class BankDatabase's no-argument constructor (lines 8–12) initializes accounts with new Account objects. For the sake of testing the system, we declare accounts to hold just two array elements, which we instantiate as new Account objects with test data. The Account constructor has four parameters—the account number, the PIN assigned to the account, the initial available balance and the initial total balance. Recall that class BankDatabase serves as an intermediary between class ATM and the actual Account objects that contain a user's account information. Thus, the methods of class BankDatabase do nothing more than invoke the corresponding methods of the Account object belonging to the current ATM user.

We include private utility method getAccount (lines 15–25) to allow the BankDatabase to obtain a reference to a particular Account within array accounts. To locate the user's Account, the BankDatabase compares the value returned by method getAccountNumber for each element of accounts to a specified account number until it finds a match. Lines 17–22 traverse the accounts array. If the account number of currentAccount equals the value of parameter accountNumber, the method immediately returns the currentAccount. If no account has the given account number, then line 24 returns null.

Method authenticateUser (lines 29–40) proves or disproves the identity of an ATM user. This method takes a user-specified account number and PIN as arguments and indicates whether they match the account number and PIN of an Account in the database. Line 31 calls method getAccount, which returns either an Account with userAccountNumber as its account number or null to indicate that userAccountNumber is invalid. If getAccount returns an Account object, line 35 returns the boolean value returned by that object's validatePIN method. BankDatabase's authenticateUser method does not perform the PIN comparison itself—rather, it forwards userPIN to the Account object's validatePIN method to do so. The value returned by Account method validatePIN indicates whether the user-specified PIN matches the PIN of the user's Account, so method authenticateUser simply returns this value to the class's client (i.e., ATM).

BankDatabase trusts the ATM to invoke method authenticateUser and receive a return value of true before allowing the user to perform transactions. BankDatabase also trusts that each Transaction object created by the ATM contains the valid account number of the current authenticated user and that this is the account number passed to the remaining BankDatabase methods as argument userAccountNumber. Methods getAvailableBalance (lines 43–45), getTotalBalance (lines 48–50), credit (lines 53–55) and debit (lines 58–60) therefore simply retrieve the user's Account object with utility method getAccount, then invoke the appropriate Account method on that object. We know that the calls to getAccount from these methods will never return null, because userAccountNumber must refer to an existing Account. Methods getAvailableBalance and getTotalBalance return the values returned by the corresponding Account methods. Also, credit and debit simply redirect parameter amount to the Account methods they invoke.

34.4.8 Class Transaction

Class Transaction (Fig. 34.20) is an abstract superclass that represents the notion of an ATM transaction. It contains the common features of subclasses BalanceInquiry, Withdrawal and Deposit. This class expands upon the "skeleton" code first developed in Section 34.3. Line 4 declares this class to be abstract. Lines 5–7 declare the class's private attributes. Recall from the class diagram of Fig. 34.10 that class Transaction con-

tains an attribute `accountNumber` (line 5) that indicates the account involved in the `Transaction`. We derive attributes `screen` (line 6) and `bankDatabase` (line 7) from class `Transaction`'s associations modeled in Fig. 34.9—all transactions require access to the ATM's screen and the bank's database.

```
1  // Transaction.java
2  // Abstract superclass Transaction represents an ATM transaction
3
4  public abstract class Transaction {
5     private int accountNumber; // indicates account involved
6     private Screen screen; // ATM's screen
7     private BankDatabase bankDatabase; // account info database
8
9     // Transaction constructor invoked by subclasses using super()
10    public Transaction(int userAccountNumber, Screen atmScreen,
11       BankDatabase atmBankDatabase) {
12
13       accountNumber = userAccountNumber;
14       screen = atmScreen;
15       bankDatabase = atmBankDatabase;
16    }
17
18    // return account number
19    public int getAccountNumber() {
20       return accountNumber;
21    }
22
23    // return reference to screen
24    public Screen getScreen() {
25       return screen;
26    }
27
28    // return reference to bank database
29    public BankDatabase getBankDatabase() {
30       return bankDatabase;
31    }
32
33    // perform the transaction (overridden by each subclass)
34    abstract public void execute();
35 }
```

Fig. 34.20 | Abstract superclass `Transaction` represents an ATM transaction.

Class `Transaction`'s constructor (lines 10–16) takes as arguments the current user's account number and references to the ATM's screen and the bank's database. Because `Transaction` is an *abstract* class, this constructor will be called only by the constructors of the `Transaction` subclasses.

The class has three `public` *get* methods—`getAccountNumber` (lines 19–21), `getScreen` (lines 24–26) and `getBankDatabase` (lines 29–31). These are inherited by `Transaction` subclasses and used to gain access to class `Transaction`'s `private` attributes.

Class `Transaction` also declares `abstract` method `execute` (line 34). It does not make sense to provide this method's implementation, because a generic transaction cannot

be executed. So, we declare this method `abstract` and force each `Transaction` subclass to provide a concrete implementation that executes that particular type of transaction.

34.4.9 Class `BalanceInquiry`

Class `BalanceInquiry` (Fig. 34.21) extends `Transaction` and represents a balance-inquiry ATM transaction. `BalanceInquiry` does not have any attributes of its own, but it inherits `Transaction` attributes `accountNumber`, `screen` and `bankDatabase`, which are accessible through `Transaction`'s `public` *get* methods. The `BalanceInquiry` constructor takes arguments corresponding to these attributes and simply forwards them to `Transaction`'s constructor using `super` (line 9).

```
1  // BalanceInquiry.java
2  // Represents a balance inquiry ATM transaction
3
4  public class BalanceInquiry extends Transaction {
5     // BalanceInquiry constructor
6     public BalanceInquiry(int userAccountNumber, Screen atmScreen,
7        BankDatabase atmBankDatabase) {
8
9        super(userAccountNumber, atmScreen, atmBankDatabase);
10    }
11
12    // performs the transaction
13    @Override
14    public void execute() {
15       // get references to bank database and screen
16       BankDatabase bankDatabase = getBankDatabase();
17       Screen screen = getScreen();
18
19       // get the available balance for the account involved
20       double availableBalance =
21          bankDatabase.getAvailableBalance(getAccountNumber());
22
23       // get the total balance for the account involved
24       double totalBalance =
25          bankDatabase.getTotalBalance(getAccountNumber());
26
27       // display the balance information on the screen
28       screen.displayMessageLine("\nBalance Information:");
29       screen.displayMessage(" - Available balance: ");
30       screen.displayDollarAmount(availableBalance);
31       screen.displayMessage("\n - Total balance:     ");
32       screen.displayDollarAmount(totalBalance);
33       screen.displayMessageLine("");
34    }
35 }
```

Fig. 34.21 | Class `BalanceInquiry` represents a balance-inquiry ATM transaction.

Class `BalanceInquiry` overrides `Transaction`'s abstract method `execute` to provide a concrete implementation (lines 13–34) that performs the steps involved in a balance

inquiry. Lines 16–17 get references to the bank database and the ATM's screen by invoking methods inherited from superclass `Transaction`. Lines 20–21 retrieve the available balance of the account involved by invoking method `getAvailableBalance` of `bankDatabase`. Line 21 uses inherited method `getAccountNumber` to get the account number of the current user, which it then passes to `getAvailableBalance`. Lines 24–25 retrieve the total balance of the current user's account. Lines 28–33 display the balance information on the ATM's screen. Recall that `displayDollarAmount` takes a `double` argument and outputs it to the screen formatted as a dollar amount. For example, if a user's `availableBalance` is `1000.5`, line 30 outputs `$1,000.50`. Line 33 inserts a blank line of output to separate the balance information from subsequent output (i.e., the main menu repeated by class `ATM` after executing the `BalanceInquiry`).

34.4.10 Class `Withdrawal`

Class `Withdrawal` (Fig. 34.22) extends `Transaction` and represents a withdrawal ATM transaction. This class expands upon the "skeleton" code for this class developed in Fig. 34.12. Recall from the class diagram of Fig. 34.10 that class `Withdrawal` has one attribute, `amount`, which line 5 implements as an `int` field. Figure 34.9 models associations between class `Withdrawal` and classes `Keypad` and `CashDispenser`, for which lines 6–7 implement reference-type attributes `keypad` and `cashDispenser`, respectively. Line 10 declares a constant corresponding to the cancel menu option. We'll soon discuss how the class uses this constant.

```
1  // Withdrawal.java
2  // Represents a withdrawal ATM transaction
3
4  public class Withdrawal extends Transaction {
5     private int amount; // amount to withdraw
6     private Keypad keypad; // reference to keypad
7     private CashDispenser cashDispenser; // reference to cash dispenser
8
9     // constant corresponding to menu option to cancel
10    private final static int CANCELED = 6;
11
12    // Withdrawal constructor
13    public Withdrawal(int userAccountNumber, Screen atmScreen,
14       BankDatabase atmBankDatabase, Keypad atmKeypad,
15       CashDispenser atmCashDispenser) {
16
17       // initialize superclass variables
18       super(userAccountNumber, atmScreen, atmBankDatabase);
19
20       // initialize references to keypad and cash dispenser
21       keypad = atmKeypad;
22       cashDispenser = atmCashDispenser;
23    }
24
```

Fig. 34.22 | Class `Withdrawal` represents a withdrawal ATM transaction. (Part 1 of 3.)

```
25      // perform transaction
26      @Override
27      public void execute() {
28         boolean cashDispensed = false; // cash was not dispensed yet
29         double availableBalance; // amount available for withdrawal
30
31         // get references to bank database and screen
32         BankDatabase bankDatabase = getBankDatabase();
33         Screen screen = getScreen();
34
35         // loop until cash is dispensed or the user cancels
36         do {
37            // obtain a chosen withdrawal amount from the user
38            amount = displayMenuOfAmounts();
39
40            // check whether user chose a withdrawal amount or canceled
41            if (amount != CANCELED) {
42               // get available balance of account involved
43               availableBalance =
44                  bankDatabase.getAvailableBalance(getAccountNumber());
45
46               // check whether the user has enough money in the account
47               if (amount <= availableBalance) {
48                  // check whether the cash dispenser has enough money
49                  if (cashDispenser.isSufficientCashAvailable(amount)) {
50                     // update the account involved to reflect the withdrawal
51                     bankDatabase.debit(getAccountNumber(), amount);
52
53                     cashDispenser.dispenseCash(amount); // dispense cash
54                     cashDispensed = true; // cash was dispensed
55
56                     // instruct user to take cash
57                     screen.displayMessageLine("\nYour cash has been" +
58                        " dispensed. Please take your cash now.");
59                  }
60                  else { // cash dispenser does not have enough cash
61                     screen.displayMessageLine(
62                        "\nInsufficient cash available in the ATM." +
63                        "\n\nPlease choose a smaller amount.");
64                  }
65               }
66               else { // not enough money available in user's account
67                  screen.displayMessageLine(
68                     "\nInsufficient funds in your account." +
69                     "\n\nPlease choose a smaller amount.");
70               }
71            }
72            else { // user chose cancel menu option
73               screen.displayMessageLine("\nCanceling transaction...");
74               return; // return to main menu because user canceled
75            }
76         } while (!cashDispensed);
77      }
```

Fig. 34.22 | Class `Withdrawal` represents a withdrawal ATM transaction. (Part 2 of 3.)

```
78
79      // display a menu of withdrawal amounts and the option to cancel;
80      // return the chosen amount or 0 if the user chooses to cancel
81      private int displayMenuOfAmounts() {
82         int userChoice = 0; // local variable to store return value
83
84         Screen screen = getScreen(); // get screen reference
85
86         // array of amounts to correspond to menu numbers
87         int[] amounts = {0, 20, 40, 60, 100, 200};
88
89         // loop while no valid choice has been made
90         while (userChoice == 0) {
91            // display the withdrawal menu
92            screen.displayMessageLine("\nWithdrawal Menu:");
93            screen.displayMessageLine("1 - $20");
94            screen.displayMessageLine("2 - $40");
95            screen.displayMessageLine("3 - $60");
96            screen.displayMessageLine("4 - $100");
97            screen.displayMessageLine("5 - $200");
98            screen.displayMessageLine("6 - Cancel transaction");
99            screen.displayMessage("\nChoose a withdrawal amount: ");
100
101           int input = keypad.getInput(); // get user input through keypad
102
103           // determine how to proceed based on the input value
104           switch (input) {
105              case 1: // if the user chose a withdrawal amount
106              case 2: // (i.e., chose option 1, 2, 3, 4 or 5), return the
107              case 3: // corresponding amount from amounts array
108              case 4:
109              case 5:
110                 userChoice = amounts[input]; // save user's choice
111                 break;
112              case CANCELED: // the user chose to cancel
113                 userChoice = CANCELED; // save user's choice
114                 break;
115              default: // the user did not enter a value from 1-6
116                 screen.displayMessageLine(
117                    "\nInvalid selection. Try again.");
118           }
119        }
120
121        return userChoice; // return withdrawal amount or CANCELED
122     }
123  }
```

Fig. 34.22 | Class `Withdrawal` represents a withdrawal ATM transaction. (Part 3 of 3.)

Class `Withdrawal`'s constructor (lines 13–23) has five parameters. It uses `super` to pass parameters `userAccountNumber`, `atmScreen` and `atmBankDatabase` to superclass `Transaction`'s constructor to set the attributes that `Withdrawal` inherits from `Transaction`. The constructor also takes references `atmKeypad` and `atmCashDispenser` as parameters and assigns them to reference-type attributes `keypad` and `cashDispenser`.

Class `Withdrawal` overrides `Transaction` method `execute` with a concrete implementation (lines 26–77) that performs the steps of a withdrawal. Line 28 declares and initializes a local `boolean` variable `cashDispensed`, which indicates whether cash has been dispensed (i.e., whether the transaction has completed successfully) and is initially `false`. Line 29 declares local `double` variable `availableBalance`, which will store the user's available balance during a withdrawal transaction. Lines 32–33 get references to the bank database and the ATM's screen by invoking methods inherited from superclass `Transaction`.

Lines 36–76 execute until cash is dispensed (i.e., until `cashDispensed` becomes `true`) or until the user chooses to cancel (in which case, the loop terminates). We use this loop to continuously return the user to the start of the transaction if an error occurs (i.e., the requested withdrawal amount is greater than the user's available balance or greater than the amount of cash in the cash dispenser). Line 38 displays a menu of withdrawal amounts and obtains a user selection by calling `private` utility method `displayMenuOfAmounts` (declared in lines 81–122). This method displays the menu of amounts and returns either an `int` withdrawal amount or an `int` constant `CANCELED` to indicate that the user has chosen to cancel the transaction.

Method `displayMenuOfAmounts` (lines 81–122) first declares local variable `userChoice` (initially `0`) to store the value that the method will return (line 82). Line 84 gets a reference to the screen by calling method `getScreen` inherited from superclass `Transaction`. Line 87 declares an integer array of withdrawal amounts that correspond to the amounts displayed in the withdrawal menu. We ignore the first element in the array (index `0`) because the menu has no option 0. Lines 90–119 repeat until `userChoice` takes on a value other than `0`. We'll see shortly that this occurs when the user makes a valid selection from the menu. Lines 92–99 display the withdrawal menu on the screen and prompt the user to enter a choice. Line 101 obtains integer `input` through the keypad. The `switch` statement at lines 104–118 determines how to proceed based on the user's input. If the user selects a number between `1` and `5`, line 110 sets `userChoice` to the value of the element in `amounts` at index `input`. For example, if the user enters `3` to withdraw $60, line 110 sets `userChoice` to the value of `amounts[3]` (i.e., `60`). Variable `userChoice` no longer equals `0`, so the loop terminates and line 121 returns `userChoice`. If the user selects the cancel menu option, lines 113–114 execute, setting `userChoice` to `CANCELED` and causing the method to return this value. If the user does not enter a valid menu selection, lines 116–117 display an error message and the user is returned to the withdrawal menu.

Line 41 in method `execute` determines whether the user has selected a withdrawal amount or chosen to cancel. If the user cancels, lines 73–74 execute and display an appropriate message to the user before returning control to the calling method (i.e., `ATM` method `performTransactions`). If the user has chosen a withdrawal amount, lines 43–44 retrieve the available balance of the current user's `Account` and store it in variable `availableBalance`. Next, line 47 determines whether the selected amount is less than or equal to the user's available balance. If it's not, lines 67–69 display an appropriate error message. Control then continues to the end of the `do...while`, and the loop repeats because `cashDispensed` is still `false`. If the user's balance is high enough, the `if` statement at line 49 determines whether the cash dispenser has enough money to satisfy the withdrawal request by invoking the `cashDispenser`'s `isSufficientCashAvailable` method. If this method returns `false`, lines 61–63 display an appropriate error message and the `do...while` repeats. If sufficient cash is available, then the requirements for the withdrawal are satis-

fied, and line 51 debits `amount` from the user's account in the database. Lines 53–54 then instruct the cash dispenser to dispense the cash to the user and set `cashDispensed` to `true`. Finally, lines 57–58 display a message to the user that cash has been dispensed. Because `cashDispensed` is now `true`, control continues after the `do...while`. No additional statements appear below the loop, so the method returns.

34.4.11 Class `Deposit`

Class `Deposit` (Fig. 34.23) extends `Transaction` and represents a deposit transaction. Recall from Fig. 34.10 that class `Deposit` has one attribute `amount`, which line 5 implements as an `int` field. Lines 6–7 create reference attributes `keypad` and `depositSlot` that implement the associations between class `Deposit` and classes `Keypad` and `DepositSlot` modeled in Fig. 34.9. Line 8 declares a constant `CANCELED` that corresponds to the value a user enters to cancel. We'll soon discuss how the class uses this constant.

```
1  // Deposit.java
2  // Represents a deposit ATM transaction
3
4  public class Deposit extends Transaction {
5     private double amount; // amount to deposit
6     private Keypad keypad; // reference to keypad
7     private DepositSlot depositSlot; // reference to deposit slot
8     private final static int CANCELED = 0; // constant for cancel option
9
10    // Deposit constructor
11    public Deposit(int userAccountNumber, Screen atmScreen,
12       BankDatabase atmBankDatabase, Keypad atmKeypad,
13       DepositSlot atmDepositSlot) {
14
15       // initialize superclass variables
16       super(userAccountNumber, atmScreen, atmBankDatabase);
17
18       // initialize references to keypad and deposit slot
19       keypad = atmKeypad;
20       depositSlot = atmDepositSlot;
21    }
22
23    // perform transaction
24    @Override
25    public void execute() {
26       BankDatabase bankDatabase = getBankDatabase(); // get reference
27       Screen screen = getScreen(); // get reference
28
29       amount = promptForDepositAmount(); // get deposit amount from user
30
31       // check whether user entered a deposit amount or canceled
32       if (amount != CANCELED) {
33          // request deposit envelope containing specified amount
34          screen.displayMessage(
35             "\nPlease insert a deposit envelope containing ");
```

Fig. 34.23 | Class `Deposit` represents a deposit ATM transaction. (Part 1 of 2.)

```
36            screen.displayDollarAmount(amount);
37            screen.displayMessageLine(".");
38
39            // receive deposit envelope
40            boolean envelopeReceived = depositSlot.isEnvelopeReceived();
41
42            // check whether deposit envelope was received
43            if (envelopeReceived) {
44               screen.displayMessageLine("\nYour envelope has been " +
45                  "received.\nNOTE: The money just deposited will not " +
46                  "be available until we verify the amount of any " +
47                  "enclosed cash and your checks clear.");
48
49               // credit account to reflect the deposit
50               bankDatabase.credit(getAccountNumber(), amount);
51            }
52            else { // deposit envelope not received
53               screen.displayMessageLine("\nYou did not insert an " +
54                  "envelope, so the ATM has canceled your transaction.");
55            }
56         }
57         else { // user canceled instead of entering amount
58            screen.displayMessageLine("\nCanceling transaction...");
59         }
60      }
61
62      // prompt user to enter a deposit amount in cents
63      private double promptForDepositAmount() {
64         Screen screen = getScreen(); // get reference to screen
65
66         // display the prompt
67         screen.displayMessage("\nPlease enter a deposit amount in " +
68            "CENTS (or 0 to cancel): ");
69         int input = keypad.getInput(); // receive input of deposit amount
70
71         // check whether the user canceled or entered a valid amount
72         if (input == CANCELED) {
73            return CANCELED;
74         }
75         else {
76            return (double) input / 100; // return dollar amount
77         }
78      }
79   }
```

Fig. 34.23 | Class `Deposit` represents a deposit ATM transaction. (Part 2 of 2.)

Like `Withdrawal`, class `Deposit`'s constructor (lines 11–21) passes three parameters to superclass `Transaction`'s constructor. The constructor also has parameters `atmKeypad` and `atmDepositSlot`, which it assigns to corresponding attributes.

Method `execute` (lines 24–60) overrides the `abstract` version in superclass `Transaction` with a concrete implementation that performs the steps required in a deposit transaction. Lines 26–27 get references to the database and the screen. Line 29 prompts the user

to enter a deposit amount by invoking `private` utility method `promptForDepositAmount` (declared in lines 63–78) and sets attribute `amount` to the value returned. Method `promptForDepositAmount` asks the user to enter a deposit amount as an integer number of cents (because the ATM's keypad does not contain a decimal point; this is consistent with many real ATMs) and returns the `double` value representing the dollar amount to be deposited.

Line 64 in method `promptForDepositAmount` gets a reference to the ATM's screen. Lines 67–68 display a message asking the user to input a deposit amount as a number of cents or "0" to cancel the transaction. Line 69 receives the user's input from the keypad. Lines 72–77 determine whether the user has entered a real deposit amount or chosen to cancel. If the latter, line 73 returns the constant `CANCELED`. Otherwise, line 76 returns the deposit amount after converting from the number of cents to a dollar amount by casting `input` to a `double`, then dividing by `100`. For example, if the user enters `125` as the number of cents, line 76 returns `125.0` divided by `100`, or `1.25`—125 cents is $1.25.

Line 32 in method `execute` determines whether the user has chosen to cancel the transaction instead of entering a deposit amount. If so, line 58 displays an appropriate message, and the method returns. If the user enters a deposit amount, lines 34–37 instruct the user to insert a deposit envelope with the correct amount. Recall that `Screen` method `displayDollarAmount` outputs a `double` formatted as a dollar amount.

Line 40 sets a local `boolean` variable to the value returned by `depositSlot`'s `isEnvelopeReceived` method, indicating whether a deposit envelope has been received. Recall that we coded method `isEnvelopeReceived` (Fig. 34.17) to always return `true`, because we're simulating the functionality of the deposit slot and assume that the user always inserts an envelope. However, we code method `execute` of class `Deposit` to test for the possibility that the user does not insert an envelope—good software engineering demands that programs account for *all* possible return values. Thus, class `Deposit` is prepared for future versions of `isEnvelopeReceived` that could return `false`. Lines 44–50 execute if the deposit slot receives an envelope. Lines 44–47 display an appropriate message to the user. Line 50 then credits the deposit amount to the user's account in the database. Lines 53–54 will execute if the deposit slot does not receive a deposit envelope. In this case, we display a message to the user stating that the ATM has canceled the transaction. The method then returns without modifying the user's account.

34.4.12 Class ATMCaseStudy

Class `ATMCaseStudy` (Fig. 34.24) is a simple class that allows us to start, or "turn on," the ATM and test the implementation of our ATM system model. Class `ATMCaseStudy`'s `main` method instantiates a new `ATM` object named `theATM` and invokes its `run` method to start the ATM.

```
1  // ATMCaseStudy.java
2  // Driver program for the ATM case study
3
4  public class ATMCaseStudy {
5     // main method creates and runs the ATM
6     public static void main(String[] args) {
```

Fig. 34.24 | `ATMCaseStudy.java` starts the ATM.

```
7          ATM theATM = new ATM();
8          theATM.run();
9       }
10   }
```

Fig. 34.24 | `ATMCaseStudy.java` starts the ATM.

34.5 Wrap-Up

In this chapter, you used inheritance to tune the design of the ATM software system, and you fully implemented the ATM in Java. Congratulations on completing the entire ATM case study! We hope you found this experience to be valuable and that it reinforced many of the object-oriented programming concepts that you've learned. In the next chapter, we present the Java Platform Module System—Java 9's most important new software-engineering technology.

Answers to Self-Review Exercises

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

34.2 b.

34.3 The design for class `Keypad` yields the code in Fig. 34.25. Recall that class `Keypad` has no attributes for the moment, but attributes may become apparent as we continue the implementation. Also, if we were designing a real ATM, method `getInput` would need to interact with the ATM's keypad hardware. We'll actually read input from the keyboard of a personal computer when we write the complete Java code in Section 34.4.

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

Fig. 34.25 | Java code for class `Keypad` based on Figs. 34.1–34.2.

34.4 b.

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

34.6 The design for class `Transaction` yields the code in Fig. 34.26. The bodies of the class constructor and methods are completed in Section 34.4. When fully implemented, methods `getScreen` and `getBankDatabase` will return superclass `Transaction`'s `private` reference attributes `screen` and `bankDatabase`, respectively. These methods allow the `Transaction` subclasses to access the ATM's screen and interact with the bank's database.

```
1  // Abstract class Transaction represents an ATM transaction
2  public abstract class Transaction {
3     // attributes
4     private int accountNumber; // indicates account involved
5     private Screen screen; // ATM's screen
6     private BankDatabase bankDatabase; // account info database
7
8     // no-argument constructor invoked by subclasses using super()
9     public Transaction() { }
10
11    // return account number
12    public int getAccountNumber() { }
13
14    // return reference to screen
15    public Screen getScreen() { }
16
17    // return reference to bank database
18    public BankDatabase getBankDatabase() { }
19
20    // abstract method overridden by subclasses
21    public abstract void execute();
22 }
```

Fig. 34.26 | Java code for class `Transaction` based on Figs. 34.9 and 34.10.