



# J

# ATM Case Study Code

## J.1 ATM Case Study Implementation

This appendix contains the complete working implementation of the ATM system that we designed in the “Software Engineering Case Study” sections found at the ends of Chapters 1–8 and 10. The implementation comprises 670 lines of Java code. We consider the classes in the order in which we identified them in Section 3.10:

- ATM
- Screen
- Keypad
- CashDispenser
- DepositSlot
- Account
- BankDatabase
- Transaction
- BalanceInquiry
- Withdrawal
- Deposit

We apply the guidelines discussed in Section 8.19 and Section 10.9 to code these classes based on how we modeled them in the UML class diagrams of Fig. 10.21 and Fig. 10.22. To develop the bodies of class methods, we refer to the activity diagrams presented in Section 5.11 and the communication and sequence diagrams presented in Section 7.14. Note that 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 com-

plete the program logic and add attributes and behaviors as necessary to construct the ATM system specified by the requirements document in Section 2.9.

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 are 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 only simulate 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.

## J.2 Class ATM

Class ATM (Fig. J.1) represents the ATM as a whole. Lines 6–12 implement the class's attributes. We determine all but one of these attributes from the UML class diagrams of Fig. 10.21 and Fig. 10.22. Note that we implement the UML `Boolean` attribute `userAuthenticated` in Fig. 10.22 as a `boolean` attribute in Java (line 6). Line 7 declares an attribute not found in our UML design—an `int` attribute `currentAccountNumber` that keeps track of the account number of the current authenticated user. We will soon see how the class uses this attribute. Lines 8–12 declare reference-type attributes corresponding to the ATM class's associations modeled in the class diagram of Fig. 10.21. 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 {
6     private boolean userAuthenticated; // whether user is authenticated
7     private int currentAccountNumber; // current user's account number
8     private Screen screen; // ATM's screen
9     private Keypad keypad; // ATM's keypad
10    private CashDispenser cashDispenser; // ATM's cash dispenser
11    private DepositSlot depositSlot; // ATM's deposit slot
12    private BankDatabase bankDatabase; // account information database
13
14    // constants corresponding to main menu options
15    private static final int BALANCE_INQUIRY = 1;
16    private static final int WITHDRAWAL = 2;
17    private static final int DEPOSIT = 3;
18    private static final int EXIT = 4;
19
20    // no-argument ATM constructor initializes instance variables
21    public ATM()
22    {
23        userAuthenticated = false; // user is not authenticated to start
24        currentAccountNumber = 0; // no current account number to start
25        screen = new Screen(); // create screen

```

Fig. J.1 | Class ATM represents the ATM. (Part I of 4.)

```

26     keypad = new Keypad(); // create keypad
27     cashDispenser = new CashDispenser(); // create cash dispenser
28     depositSlot = new DepositSlot(); // create deposit slot
29     bankDatabase = new BankDatabase(); // create acct info database
30 } // end no-argument ATM constructor
31
32 // start ATM
33 public void run()
34 {
35     // welcome and authenticate user; perform transactions
36     while ( true )
37     {
38         // loop while user is not yet authenticated
39         while ( !userAuthenticated )
40         {
41             screen.displayMessageLine( "\nWelcome!" );
42             authenticateUser(); // authenticate user
43         } // end while
44
45         performTransactions(); // user is now authenticated
46         userAuthenticated = false; // reset before next ATM session
47         currentAccountNumber = 0; // reset before next ATM session
48         screen.displayMessageLine( "\nThank you! Goodbye!" );
49     } // end while
50 } // end method run
51
52 // attempts to authenticate user against database
53 private void authenticateUser()
54 {
55     screen.displayMessage( "\nPlease enter your account number: " );
56     int accountNumber = keypad.getInput(); // input account number
57     screen.displayMessage( "\nEnter your PIN: " ); // prompt for PIN
58     int pin = keypad.getInput(); // input PIN
59
60     // set userAuthenticated to boolean value returned by database
61     userAuthenticated =
62         bankDatabase.authenticateUser( accountNumber, pin );
63
64     // check whether authentication succeeded
65     if ( userAuthenticated )
66     {
67         currentAccountNumber = accountNumber; // save user's account #
68     } // end if
69     else
70         screen.displayMessageLine(
71             "Invalid account number or PIN. Please try again." );
72 } // end method authenticateUser
73
74 // display the main menu and perform transactions
75 private void performTransactions()
76 {
77     // local variable to store transaction currently being processed
78     Transaction currentTransaction = null;

```

**Fig. J.1** | Class ATM represents the ATM. (Part 2 of 4.)

```

79
80     boolean userExited = false; // user has not chosen to exit
81
82     // loop while user has not chosen option to exit system
83     while ( !userExited )
84     {
85         // show main menu and get user selection
86         int mainMenuSelection = displayMainMenu();
87
88         // decide how to proceed based on user's menu selection
89         switch ( mainMenuSelection )
90         {
91             // user chose to perform one of three transaction types
92             case BALANCE_INQUIRY:
93             case WITHDRAWAL:
94             case DEPOSIT:
95
96                 // initialize as new object of chosen type
97                 currentTransaction =
98                     createTransaction( mainMenuSelection );
99
100                currentTransaction.execute(); // execute transaction
101                break;
102            case EXIT: // user chose to terminate session
103                screen.displayMessageLine( "\nExiting the system..." );
104                userExited = true; // this ATM session should end
105                break;
106            default: // user did not enter an integer from 1-4
107                screen.displayMessageLine(
108                    "\nYou did not enter a valid selection. Try again." );
109                break;
110        } // end switch
111    } // end while
112 } // end method performTransactions
113
114 // display the main menu and return an input selection
115 private int displayMainMenu()
116 {
117     screen.displayMessageLine( "\nMain menu:" );
118     screen.displayMessageLine( "1 - View my balance" );
119     screen.displayMessageLine( "2 - Withdraw cash" );
120     screen.displayMessageLine( "3 - Deposit funds" );
121     screen.displayMessageLine( "4 - Exit\n" );
122     screen.displayMessage( "Enter a choice: " );
123     return keypad.getInput(); // return user's selection
124 } // end method displayMainMenu
125
126 // return object of specified Transaction subclass
127 private Transaction createTransaction( int type )
128 {
129     Transaction temp = null; // temporary Transaction variable
130

```

**Fig. J.1** | Class ATM represents the ATM. (Part 3 of 4.)

```

131 // determine which type of Transaction to create
132 switch ( type )
133 {
134     case BALANCE_INQUIRY: // create new BalanceInquiry transaction
135         temp = new BalanceInquiry(
136             currentAccountNumber, screen, bankDatabase );
137         break;
138     case WITHDRAWAL: // create new Withdrawal transaction
139         temp = new Withdrawal( currentAccountNumber, screen,
140             bankDatabase, keypad, cashDispenser );
141         break;
142     case DEPOSIT: // create new Deposit transaction
143         temp = new Deposit( currentAccountNumber, screen,
144             bankDatabase, keypad, depositSlot );
145         break;
146     } // end switch
147
148     return temp; // return the newly created object
149 } // end method createTransaction
150 } // end class ATM

```

**Fig. J.1** | Class ATM represents the ATM. (Part 4 of 4.)

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

The class diagram of Fig. 10.22 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 33–50) uses an infinite loop (lines 36–49) to repeatedly welcome a user, attempt to authenticate the user and, if authentication succeeds, allow the user to perform transactions. After an 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 does not have the ability to turn off the ATM completely.

Inside method `run`’s infinite loop, lines 39–43 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 41 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 13–16 of Fig. J.2) displays a message to the user, but this method also outputs a newline after displaying the message. We have added this method during implementation to give class `Screen`'s clients more control over the placement of displayed messages. Line 42 invokes class `ATM`'s private utility method `authenticateUser` (declared in lines 53–72) to attempt to authenticate the user.

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

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

Method `performTransactions` (lines 75–112) carries out an `ATM` session for an authenticated user. Line 78 declares a local `Transaction` variable to which we assign a `BalanceInquiry`, `Withdrawal` or `Deposit` object representing the `ATM` transaction currently being processed. Note that we use a `Transaction` variable here to allow us to take advantage of polymorphism. Also note that we name this variable after the role name included in the class diagram of Fig. 3.21—`currentTransaction`. Line 80 declares another local variable—a `boolean` called `userExited` that keeps track of whether the user has chosen to exit. This variable controls a `while` loop (lines 83–111) that allows the user to execute an unlimited number of transactions before choosing to exit. Within this loop, line 86 displays the main menu and obtains the user's menu selection by calling an `ATM` utility method `displayMainMenu` (declared in lines 115–124). This method displays the main menu by invoking methods of the `ATM`'s screen and returns a menu selection obtained from the user through the `ATM`'s keypad. Line 86 stores the user's selection returned by `displayMainMenu` in local variable `mainMenuSelection`.

After obtaining a main menu selection, method `performTransactions` uses a `switch` statement (lines 89–110) to respond to the selection appropriately. If `mainMenuSelection` is equal to any of the three integer constants representing transaction types (i.e., if the user chose to perform a transaction), lines 97–98 call utility method `createTransaction`

(declared in lines 127–149) to return a newly instantiated object of the type that corresponds to the selected transaction. Variable `currentTransaction` is assigned the reference returned by `createTransaction`, then line 100 invokes method `execute` of this transaction to execute it. We will discuss Transaction method `execute` and the three Transaction subclasses shortly. Note that 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.

Method `createTransaction` (lines 127–149) uses a `switch` statement (lines 132–146) to instantiate a new Transaction subclass object of the type indicated by the parameter `type`. Recall that method `performTransactions` passes `mainMenuSelection` to this method only when `mainMenuSelection` contains a value corresponding to one of the three transaction types. Therefore `type` equals either `BALANCE_INQUIRY`, `WITHDRAWAL` or `DEPOSIT`. Each case in the `switch` statement instantiates a new object by calling the appropriate Transaction subclass constructor. Note that 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 Section J.9–Section J.12.

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

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

### J.3 Class Screen

Class `Screen` (Fig. J.2) 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 {
6     // display a message without a carriage return
7     public void displayMessage( String message )
8     {
9         System.out.print( message );
10    } // end method displayMessage
11
12    // display a message with a carriage return
13    public void displayMessageLine( String message )
14    {
15        System.out.println( message );
16    } // end method displayMessageLine
17
18    // displays a dollar amount
19    public void displayDollarAmount( double amount )
20    {
21        System.out.printf( "$%,.2f", amount );
22    } // end method displayDollarAmount
23 } // end class Screen

```

**Fig. J.2** | Class `Screen` represents the screen of the ATM.

Method `displayMessage` (lines 7–10) takes a `String` as an argument and prints it to the console using `System.out.print`. The cursor stays on the same line, making this method appropriate for displaying prompts to the user. Method `displayMessageLine` (lines 13–16) does the same using `System.out.println`, which outputs a newline to move the cursor to the next line. Finally, method `displayDollarAmount` (lines 19–22) outputs a properly formatted dollar amount (e.g., \$1,234.56). Line 21 uses method `System.out.printf` to output a `double` value formatted with commas to increase readability and two decimal places. See Chapter 28, *Formatted Output*, for more information about formatting output with `printf`.

## J.4 Class Keypad

Class `Keypad` (Fig. J.3) represents the keypad of the ATM and is responsible for receiving all user input. Recall that we are 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.

Line 3 of class `Keypad` imports class `Scanner` for use in class `Keypad`. Line 7 declares `Scanner` variable `input` as an instance variable. Line 12 in the constructor creates a new `Scanner` object that reads input from the standard input stream (`System.in`) and assigns the object’s reference to variable `input`. Method `getInput` (declared in lines 16–19) invokes `Scanner` method `nextInt` (line 18) to return the next integer input by the user. [*Note:* Method `nextInt` can throw an `InputMismatchException` if the user enters non-integer

```

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 {
7     private Scanner input; // reads data from the command line
8
9     // no-argument constructor initializes the Scanner
10    public Keypad()
11    {
12        input = new Scanner( System.in );
13    } // end no-argument Keypad constructor
14
15    // return an integer value entered by user
16    public int getInput()
17    {
18        return input.nextInt(); // we assume that user enters an integer
19    } // end method getInput
20 } // end class Keypad

```

**Fig. J.3** | Class Keypad represents the ATM's keypad.

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 13, Exception Handling, 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 particular criteria (i.e., a number corresponding to a valid menu option), the client must perform the appropriate error checking.

## J.5 Class CashDispenser

Class `CashDispenser` (Fig. J.4) represents the cash dispenser of the ATM. Line 7 declares constant `INITIAL_COUNT`, which indicates the initial count of bills in the cash dispenser when the ATM starts (i.e., 500). Line 8 implements attribute `count` (modeled in Fig. 10.22), which keeps track of the number of bills remaining in the `CashDispenser` at any time. The constructor (lines 11–14) sets `count` to the initial count. Class `CashDispenser` has two public methods—`dispenseCash` (lines 17–21) and `isSufficientCashAvailable` (lines 24–32). The class trusts that a client (i.e., `Withdrawal`) calls `dispenseCash` only 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.

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

```

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

```

**Fig. J.4** | Class CashDispenser represents the ATM's cash dispenser.

enough bills). For example, if a user wishes to withdraw \$80 (i.e., `billsRequired` is 4), but only three bills remain (i.e., `count` is 3), the method returns `false`.

Method `dispenseCash` (lines 17–21) simulates cash dispensing. If our system were hooked up to a real hardware cash dispenser, this method would interact with the hardware device to physically dispense cash. Our simulated version of the method simply decreases the count of bills remaining by the number required to dispense the specified amount (line 20). Note that it is 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`.

## J.6 Class DepositSlot

Class `DepositSlot` (Fig. J.5) represents the deposit slot of the ATM. Like the version of class `CashDispenser` presented here, this version of class `DepositSlot` merely simulates the functionality of a real hardware deposit slot. `DepositSlot` has no attributes and only one method—`isEnvelopeReceived` (lines 8–11)—that indicates whether a deposit envelope was received.

```

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

```

**Fig. J.5** | Class `DepositSlot` represents the ATM's deposit slot.

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

## J.7 Class Account

Class `Account` (Fig. J.6) represents a bank account. Each `Account` has four attributes (modeled in Fig. 10.22)—`accountNumber`, `pin`, `availableBalance` and `totalBalance`. Lines 6–9 implement these attributes as private fields. Variable `availableBalance` represents the amount of funds available for withdrawal. Variable `totalBalance` represents the amount of funds available, plus the amount of deposited funds still pending confirmation or clearance.

Class `Account` has a constructor (lines 12–19) that takes an account number, the PIN established for the account, the initial available balance and the initial total balance as arguments. Lines 15–18 assign these values to the class's attributes (i.e., fields).

Method `validatePIN` (lines 22–28) determines whether a user-specified PIN (i.e., parameter `userPIN`) matches the PIN associated with the account (i.e., attribute `pin`). Recall that we modeled this method's parameter `userPIN` in the UML class diagram of Fig. 6.23. If the two PINs match, the method returns `true` (line 25); otherwise, it returns `false` (line 27).

Methods `getAvailableBalance` (lines 31–34) and `getTotalBalance` (lines 37–40) are *get* methods that return the values of `double` attributes `availableBalance` and `totalBalance`, respectively.

Method `credit` (lines 43–46) adds an amount of money (i.e., parameter `amount`) to an `Account` as part of a deposit transaction. Note that this method adds the amount only to attribute `totalBalance` (line 45). The money credited to an account during a deposit

```

1 // Account.java
2 // Represents a bank account
3
4 public class Account
5 {
6     private int accountNumber; // account number
7     private int pin; // PIN for authentication
8     private double availableBalance; // funds available for withdrawal
9     private double totalBalance; // funds available + pending deposits
10
11 // Account constructor initializes attributes
12 public Account( int theAccountNumber, int thePIN,
13     double theAvailableBalance, double theTotalBalance )
14 {
15     accountNumber = theAccountNumber;
16     pin = thePIN;
17     availableBalance = theAvailableBalance;
18     totalBalance = theTotalBalance;
19 } // end Account constructor
20
21 // determines whether a user-specified PIN matches PIN in Account
22 public boolean validatePIN( int userPIN )
23 {
24     if ( userPIN == pin )
25         return true;
26     else
27         return false;
28 } // end method validatePIN
29
30 // returns available balance
31 public double getAvailableBalance()
32 {
33     return availableBalance;
34 } // end getAvailableBalance
35
36 // returns the total balance
37 public double getTotalBalance()
38 {
39     return totalBalance;
40 } // end method getTotalBalance
41
42 // credits an amount to the account
43 public void credit( double amount )
44 {
45     totalBalance += amount; // add to total balance
46 } // end method credit
47
48 // debits an amount from the account
49 public void debit( double amount )
50 {
51     availableBalance -= amount; // subtract from available balance
52     totalBalance -= amount; // subtract from total balance
53 } // end method debit

```

**Fig. J.6** | Class Account represents a bank account. (Part I of 2.)

```

54
55 // returns account number
56 public int getAccountNumber()
57 {
58     return accountNumber;
59 } // end method getAccountNumber
60 } // end class Account

```

**Fig. J.6** | Class Account represents a bank account. (Part 2 of 2.)

does not become available immediately, so we modify only the total balance. We assume that the bank updates the available balance appropriately at a later time. Our implementation of class Account includes only methods required for carrying out ATM transactions. Therefore, we omit the methods that some other bank system would invoke to add to attribute `availableBalance` (to confirm a deposit) or subtract from attribute `totalBalance` (to reject a deposit).

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

Method `getAccountNumber` (lines 56–59) provides access to an Account’s `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.

## J.8 Class BankDatabase

Class `BankDatabase` (Fig. J.7) models the bank’s database with which the ATM interacts to access and modify a user’s account information. We determine one reference-type attribute for class `BankDatabase` based on its composition relationship with class `Account`. Recall from Fig. 10.21 that a `BankDatabase` is composed of zero or more objects of class `Account`. Line 6 implements attribute `accounts`—an array of `Account` objects—to implement this composition relationship. Class `BankDatabase` has a no-argument constructor (lines 9–14) that initializes `accounts` to contain a set of new `Account` objects. For the sake of testing the system, we declare `accounts` to hold just two array elements (line 11), which we instantiate as new `Account` objects with test data (lines 12–13). Note that 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 17–28) 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 20–25 traverse the `accounts` array. If the account number of `currentAc-`

count equals the value of parameter `accountNumber`, the method immediately returns the `currentAccount`. If no account has the given account number, then line 27 returns `null`.

```

1 // BankDatabase.java
2 // Represents the bank account information database
3
4 public class BankDatabase
5 {
6     private Account accounts[]; // array of Accounts
7
8     // no-argument BankDatabase constructor initializes accounts
9     public BankDatabase()
10    {
11        accounts = new Account[ 2 ]; // just 2 accounts for testing
12        accounts[ 0 ] = new Account( 12345, 54321, 1000.0, 1200.0 );
13        accounts[ 1 ] = new Account( 98765, 56789, 200.0, 200.0 );
14    } // end no-argument BankDatabase constructor
15
16    // retrieve Account object containing specified account number
17    private Account getAccount( int accountNumber )
18    {
19        // loop through accounts searching for matching account number
20        for ( Account currentAccount : accounts )
21        {
22            // return current account if match found
23            if ( currentAccount.getAccountNumber() == accountNumber )
24                return currentAccount;
25        } // end for
26
27        return null; // if no matching account was found, return null
28    } // end method getAccount
29
30    // determine whether user-specified account number and PIN match
31    // those of an account in the database
32    public boolean authenticateUser( int userAccountNumber, int userPIN )
33    {
34        // attempt to retrieve the account with the account number
35        Account userAccount = getAccount( userAccountNumber );
36
37        // if account exists, return result of Account method validatePIN
38        if ( userAccount != null )
39            return userAccount.validatePIN( userPIN );
40        else
41            return false; // account number not found, so return false
42    } // end method authenticateUser
43
44    // return available balance of Account with specified account number
45    public double getAvailableBalance( int userAccountNumber )
46    {
47        return getAccount( userAccountNumber ).getAvailableBalance();
48    } // end method getAvailableBalance

```

**Fig. J.7** | Class `BankDatabase` represents the bank's account information database. (Part I of 2.)

```

49
50 // return total balance of Account with specified account number
51 public double getTotalBalance( int userAccountNumber )
52 {
53     return getAccount( userAccountNumber ).getTotalBalance();
54 } // end method getTotalBalance
55
56 // credit an amount to Account with specified account number
57 public void credit( int userAccountNumber, double amount )
58 {
59     getAccount( userAccountNumber ).credit( amount );
60 } // end method credit
61
62 // debit an amount from of Account with specified account number
63 public void debit( int userAccountNumber, double amount )
64 {
65     getAccount( userAccountNumber ).debit( amount );
66 } // end method debit
67 } // end class BankDatabase

```

**Fig. J.7** | Class `BankDatabase` represents the bank’s account information database. (Part 2 of 2.)

Method `authenticateUser` (lines 32–42) proves or disproves the identity of an ATM user. This method takes a user-specified account number and user-specified PIN as arguments and indicates whether they match the account number and PIN of an `Account` in the database. Line 35 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 39 returns the `boolean` value returned by that object’s `validatePIN` method. Note that `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 client of the class (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 45–48), `getTotalBalance` (lines 51–54), `credit` (lines 57–60) and `debit` (lines 63–66) therefore simply retrieve the user’s `Account` object with utility method `getAccount`, then invoke the appropriate `Account` method on that object. We know that the calls to `getAccount` within these methods will never return `null`, because `userAccountNumber` must refer to an existing `Account`. Note that `getAvailableBalance` and `getTotalBalance` return the values returned by the corresponding `Account` methods. Also note that `credit` and `debit` simply redirect parameter `amount` to the `Account` methods they invoke.

```

1 // Transaction.java
2 // Abstract superclass Transaction represents an ATM transaction
3
4 public abstract class Transaction
5 {
6     private int accountNumber; // indicates account involved
7     private Screen screen; // ATM's screen
8     private BankDatabase bankDatabase; // account info database
9
10    // Transaction constructor invoked by subclasses using super()
11    public Transaction( int userAccountNumber, Screen atmScreen,
12        BankDatabase atmBankDatabase )
13    {
14        accountNumber = userAccountNumber;
15        screen = atmScreen;
16        bankDatabase = atmBankDatabase;
17    } // end Transaction constructor
18
19    // return account number
20    public int getAccountNumber()
21    {
22        return accountNumber;
23    } // end method getAccountNumber
24
25    // return reference to screen
26    public Screen getScreen()
27    {
28        return screen;
29    } // end method getScreen
30
31    // return reference to bank database
32    public BankDatabase getBankDatabase()
33    {
34        return bankDatabase;
35    } // end method getBankDatabase
36
37    // perform the transaction (overridden by each subclass)
38    abstract public void execute();
39 } // end class Transaction

```

**Fig. J.8** | Abstract superclass Transaction represents an ATM transaction.

## J.9 Class Transaction

Class Transaction (Fig. J.8) 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 10.9. Line 4 declares this class to be abstract. Lines 6–8 declare the class’s private attributes. Recall from the class diagram of Fig. 10.22 that class Transaction contains an attribute accountNumber (line 6) that indicates the account involved in the Transaction. We derive attributes screen (line 7) and bankDatabase (line 8) from class Transaction’s associations modeled in Fig. 10.21—all transactions require access to the ATM’s screen and the bank’s database.

Class `Transaction` has a constructor (lines 11–17) that takes the current user’s account number and references to the ATM’s screen and the bank’s database as arguments. Because `Transaction` is an abstract class, this constructor will never be called directly to instantiate `Transaction` objects. Instead, the constructors of the `Transaction` subclasses will use `super` to invoke this constructor.

Class `Transaction` has three public *get* methods—`getAccountNumber` (lines 20–23), `getScreen` (lines 26–29) and `getBankDatabase` (lines 32–35). `Transaction` subclasses inherit these methods from `Transaction` and use them to gain access to class `Transaction`’s private attributes.

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

## J.10 Class `BalanceInquiry`

Class `BalanceInquiry` (Fig. J.9) extends `Transaction` and represents a balance inquiry ATM transaction. `BalanceInquiry` does not have any attributes of its own, but it inherits `Transaction` attributes `accountNumber`, `screen` and `bankDatabase`, which are accessible through `Transaction`’s public *get* methods. The `BalanceInquiry` constructor takes arguments corresponding to these attributes and simply forwards them to `Transaction`’s constructor using `super` (line 10).

Class `BalanceInquiry` overrides `Transaction`’s abstract method `execute` to provide a concrete implementation (lines 14–35) that performs the steps involved in a balance inquiry. Lines 17–18 get references to the bank database and the ATM’s screen by invoking methods inherited from superclass `Transaction`. Lines 21–22 retrieve the available balance of the account involved by invoking method `getAvailableBalance` of `bankDatabase`. Note that line 22 uses inherited method `getAccountNumber` to get the account number of the current user, which it then passes to `getAvailableBalance`. Lines 25–26 retrieve the total balance of the current user’s account. Lines 29–34 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 31 outputs \$1,000.50. Note that line 34 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`).

## J.11 Class `Withdrawal`

Class `Withdrawal` (Fig. J.10) extends `Transaction` and represents a withdrawal ATM transaction. This class expands upon the “skeleton” code for this class developed in Fig. 10.24. Recall from the class diagram of Fig. 10.21 that class `Withdrawal` has one attribute, `amount`, which line 6 implements as an `int` field. Figure 10.21 models associations between class `Withdrawal` and classes `Keypad` and `CashDispenser`, for which lines 7–8 implement reference-type attributes `keypad` and `cashDispenser`, respectively. Line 11 declares a constant corresponding to the cancel menu option. We will soon discuss how the class uses this constant.

```

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

```

**Fig. J.9** | Class `BalanceInquiry` represents a balance inquiry ATM transaction.

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

Class `Withdrawal` overrides `Transaction`'s abstract method `execute` with a concrete implementation (lines 27–84) that performs the steps involved in a withdrawal. Line 29 declares and initializes a local `boolean` variable `cashDispensed`. This variable indicates whether cash has been dispensed (i.e., whether the transaction has completed successfully) and is initially `false`. Line 30 declares local `double` variable `availableBalance`, which will store the user's available balance during a withdrawal transaction. Lines 33–34 get references to the bank database and the ATM's screen by invoking methods inherited from superclass `Transaction`.

Lines 37–82 contain a `do...while` statement that executes its body until cash is dispensed (i.e., until `cashDispensed` becomes true) or until the user chooses to cancel (in which case, the loop terminates). We use this loop to continuously return the user to the start of the transaction if an error occurs (i.e., the requested withdrawal amount is greater than the user's available balance or greater than the amount of cash in the cash dispenser). Line 40 displays a menu of withdrawal amounts and obtains a user selection by calling private utility method `displayMenuOfAmounts` (declared in lines 88–132). 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.

```

1 // Withdrawal.java
2 // Represents a withdrawal ATM transaction
3
4 public class Withdrawal extends Transaction
5 {
6     private int amount; // amount to withdraw
7     private Keypad keypad; // reference to keypad
8     private CashDispenser cashDispenser; // reference to cash dispenser
9
10    // constant corresponding to menu option to cancel
11    private final static int CANCELED = 6;
12
13    // Withdrawal constructor
14    public Withdrawal( int userAccountNumber, Screen atmScreen,
15                    BankDatabase atmBankDatabase, Keypad atmKeypad,
16                    CashDispenser atmCashDispenser )
17    {
18        // initialize superclass variables
19        super( userAccountNumber, atmScreen, atmBankDatabase );
20
21        // initialize references to keypad and cash dispenser
22        keypad = atmKeypad;
23        cashDispenser = atmCashDispenser;
24    } // end Withdrawal constructor
25
26    // perform transaction
27    public void execute()
28    {
29        boolean cashDispensed = false; // cash was not dispensed yet
30        double availableBalance; // amount available for withdrawal
31
32        // get references to bank database and screen
33        BankDatabase bankDatabase = getBankDatabase();
34        Screen screen = getScreen();
35
36        // loop until cash is dispensed or the user cancels
37        do
38        {
39            // obtain a chosen withdrawal amount from the user
40            amount = displayMenuOfAmounts();
41

```

**Fig. J.10** | Class `Withdrawal` represents a withdrawal ATM transaction. (Part 1 of 3.)

```

42     // check whether user chose a withdrawal amount or canceled
43     if ( amount != CANCELED )
44     {
45         // get available balance of account involved
46         availableBalance =
47             bankDatabase.getAvailableBalance( getAccountNumber() );
48
49         // check whether the user has enough money in the account
50         if ( amount <= availableBalance )
51         {
52             // check whether the cash dispenser has enough money
53             if ( cashDispenser.isSufficientCashAvailable( amount ) )
54             {
55                 // update the account involved to reflect the withdrawal
56                 bankDatabase.debit( getAccountNumber(), amount );
57
58                 cashDispenser.dispenseCash( amount ); // dispense cash
59                 cashDispensed = true; // cash was dispensed
60
61                 // instruct user to take cash
62                 screen.displayMessageLine( "\nYour cash has been" +
63                     " dispensed. Please take your cash now." );
64             } // end if
65             else // cash dispenser does not have enough cash
66                 screen.displayMessageLine(
67                     "\nInsufficient cash available in the ATM." +
68                     "\n\nPlease choose a smaller amount." );
69         } // end if
70         else // not enough money available in user's account
71         {
72             screen.displayMessageLine(
73                 "\nInsufficient funds in your account." +
74                 "\n\nPlease choose a smaller amount." );
75         } // end else
76     } // end if
77     else // user chose cancel menu option
78     {
79         screen.displayMessageLine( "\nCanceling transaction.." );
80         return; // return to main menu because user canceled
81     } // end else
82 } while ( !cashDispensed );
83
84 } // end method execute
85
86 // display a menu of withdrawal amounts and the option to cancel;
87 // return the chosen amount or 0 if the user chooses to cancel
88 private int displayMenuOfAmounts()
89 {
90     int userChoice = 0; // local variable to store return value
91
92     Screen screen = getScreen(); // get screen reference
93

```

**Fig. J.10** | Class Withdrawal represents a withdrawal ATM transaction. (Part 2 of 3.)

```

94     // array of amounts to correspond to menu numbers
95     int amounts[] = { 0, 20, 40, 60, 100, 200 };
96
97     // loop while no valid choice has been made
98     while ( userChoice == 0 )
99     {
100        // display the menu
101        screen.displayMessageLine( "\nWithdrawal Menu:" );
102        screen.displayMessageLine( "1 - $20" );
103        screen.displayMessageLine( "2 - $40" );
104        screen.displayMessageLine( "3 - $60" );
105        screen.displayMessageLine( "4 - $100" );
106        screen.displayMessageLine( "5 - $200" );
107        screen.displayMessageLine( "6 - Cancel transaction" );
108        screen.displayMessage( "\nChoose a withdrawal amount: " );
109
110        int input = keypad.getInput(); // get user input through keypad
111
112        // determine how to proceed based on the input value
113        switch ( input )
114        {
115            case 1: // if the user chose a withdrawal amount
116            case 2: // (i.e., chose option 1, 2, 3, 4 or 5), return the
117            case 3: // corresponding amount from amounts array
118            case 4:
119            case 5:
120                userChoice = amounts[ input ]; // save user's choice
121                break;
122            case CANCELED: // the user chose to cancel
123                userChoice = CANCELED; // save user's choice
124                break;
125            default: // the user did not enter a value from 1-6
126                screen.displayMessageLine(
127                    "\nInvalid selection. Try again." );
128        } // end switch
129    } // end while
130
131    return userChoice; // return withdrawal amount or CANCELED
132 } // end method displayMenuOfAmounts
133 } // end class Withdrawal

```

**Fig. J.10** | Class `Withdrawal` represents a withdrawal ATM transaction. (Part 3 of 3.)

Method `displayMenuOfAmounts` (lines 88–132) first declares local variable `userChoice` (initially 0) to store the value that the method will return (line 90). Line 92 gets a reference to the screen by calling method `getScreen` inherited from superclass `Transaction`. Line 95 declares an integer array of withdrawal amounts that correspond to the amounts displayed in the withdrawal menu. We ignore the first element in the array (index 0) because the menu has no option 0. The `while` statement at lines 98–129 repeats until `userChoice` takes on a value other than 0. We will see shortly that this occurs when the user makes a valid selection from the menu. Lines 101–108 display the withdrawal menu on the screen and prompt the user to enter a choice. Line 110 obtains integer input through the keypad. The `switch` statement at lines 113–128 determines how to proceed

based on the user's input. If the user selects a number between 1 and 5, line 120 sets `userChoice` to the value of the element in `amounts` at index `input`. For example, if the user enters 3 to withdraw \$60, line 120 sets `userChoice` to the value of `amounts[ 3 ]` (i.e., 60). Line 120 terminates the `switch`. Variable `userChoice` no longer equals 0, so the `while` at lines 98–129 terminates and line 131 returns `userChoice`. If the user selects the cancel menu option, lines 123–124 execute, setting `userChoice` to `CANCELED` and causing the method to return this value. If the user does not enter a valid menu selection, lines 126–127 display an error message and the user is returned to the withdrawal menu.

The `if` statement at line 43 in method `execute` determines whether the user has selected a withdrawal amount or chosen to cancel. If the user cancels, lines 79–80 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 46–47 retrieve the available balance of the current user's `Account` and store it in variable `availableBalance`. Next, the `if` statement at line 50 determines whether the selected amount is less than or equal to the user's available balance. If it is not, lines 72–74 display an appropriate error message. Control then continues to the end of the `do...while`, and the loops repeats because `cashDispensed` is still `false`. If the user's balance is high enough, the `if` statement at line 53 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 66–68 display an appropriate error message and the `do...while` repeats. If sufficient cash is available, then the requirements for the withdrawal are satisfied, and line 56 debits amount from the user's account in the database. Lines 58–59 then instruct the cash dispenser to dispense the cash to the user and set `cashDispensed` to `true`. Finally, lines 62–63 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 control to class `ATM`.

## J.12 Class Deposit

Class `Deposit` (Fig. J.11) extends `Transaction` and represents a deposit ATM transaction. Recall from the class diagram of Fig. 10.22 that class `Deposit` has one attribute `amount`, which line 6 implements as an `int` field. Lines 7–8 create reference-type attributes `keypad` and `depositSlot` that implement the associations between class `Deposit` and classes `Keypad` and `DepositSlot` modeled in Fig. 10.21. Line 9 declares a constant `CANCELED` that corresponds to the value a user enters to cancel. We will soon discuss how the class uses this constant.

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

Method `execute` (lines 25–65) overrides abstract method `execute` in superclass `Transaction` with a concrete implementation that performs the steps required in a deposit transaction. Lines 27–28 get references to the database and the screen. Line 30 prompts the user to enter a deposit amount by invoking private utility method `promptForDepositAmount` (declared in lines 68–84) and sets attribute `amount` to the value

```

1 // Deposit.java
2 // Represents a deposit ATM transaction
3
4 public class Deposit extends Transaction
5 {
6     private double amount; // amount to deposit
7     private Keypad keypad; // reference to keypad
8     private DepositSlot depositSlot; // reference to deposit slot
9     private final static int CANCELED = 0; // constant for cancel option
10
11     // Deposit constructor
12     public Deposit( int userAccountNumber, Screen atmScreen,
13         BankDatabase atmBankDatabase, Keypad atmKeypad,
14         DepositSlot atmDepositSlot )
15     {
16         // initialize superclass variables
17         super( userAccountNumber, atmScreen, atmBankDatabase );
18
19         // initialize references to keypad and deposit slot
20         keypad = atmKeypad;
21         depositSlot = atmDepositSlot;
22     } // end Deposit constructor
23
24     // perform transaction
25     public void execute()
26     {
27         BankDatabase bankDatabase = getBankDatabase(); // get reference
28         Screen screen = getScreen(); // get reference
29
30         amount = promptForDepositAmount(); // get deposit amount from user
31
32         // check whether user entered a deposit amount or canceled
33         if ( amount != CANCELED )
34         {
35             // request deposit envelope containing specified amount
36             screen.displayMessage(
37                 "\nPlease insert a deposit envelope containing " );
38             screen.displayDollarAmount( amount );
39             screen.displayMessageLine( "." );
40
41             // receive deposit envelope
42             boolean envelopeReceived = depositSlot.isEnvelopeReceived();
43
44             // check whether deposit envelope was received
45             if ( envelopeReceived )
46             {
47                 screen.displayMessageLine( "\nYour envelope has been " +
48                     "received.\nNOTE: The money just deposited will not " +
49                     "be available until we verify the amount of any " +
50                     "enclosed cash and your checks clear." );
51

```

**Fig. J.11** | Class `Deposit` represents a deposit ATM transaction. (Part 1 of 2.)

```

52         // credit account to reflect the deposit
53         bankDatabase.credit( getAccountNumber(), amount );
54     } // end if
55     else // deposit envelope not received
56     {
57         screen.displayMessageLine( "\nYou did not insert an " +
58             "envelope, so the ATM has canceled your transaction." );
59     } // end else
60 } // end if
61 else // user canceled instead of entering amount
62 {
63     screen.displayMessageLine( "\nCanceling transaction..." );
64 } // end else
65 } // end method execute
66
67 // prompt user to enter a deposit amount in cents
68 private double promptForDepositAmount()
69 {
70     Screen screen = getScreen(); // get reference to screen
71
72     // display the prompt
73     screen.displayMessage( "\nPlease enter a deposit amount in " +
74         "CENTS (or 0 to cancel): " );
75     int input = keypad.getInput(); // receive input of deposit amount
76
77     // check whether the user canceled or entered a valid amount
78     if ( input == CANCELED )
79         return CANCELED;
80     else
81     {
82         return ( double ) input / 100; // return dollar amount
83     } // end else
84 } // end method promptForDepositAmount
85 } // end class Deposit

```

**Fig. J.11** | Class `Deposit` represents a deposit ATM transaction. (Part 2 of 2.)

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 70 in method `promptForDepositAmount` gets a reference to the ATM's screen. Lines 73–74 display a message on the screen asking the user to input a deposit amount as a number of cents or “0” to cancel the transaction. Line 75 receives the user's input from the keypad. The `if` statement at lines 78–83 determines whether the user has entered a real deposit amount or chosen to cancel. If the user chooses to cancel, line 79 returns the constant `CANCELED`. Otherwise, line 82 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 82 returns 125.0 divided by 100, or 1.25—125 cents is \$1.25.

The `if` statement at lines 33–64 in method `execute` determines whether the user has chosen to cancel the transaction instead of entering a deposit amount. If the user cancels, line 63 displays an appropriate message, and the method returns. If the user enters a deposit amount, lines 36–39 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 42 sets a local `boolean` variable to the value returned by `depositSlot`'s `isEnvelopeReceived` method, indicating whether a deposit envelope has been received. Recall that we coded method `isEnvelopeReceived` (lines 8–11 of Fig. J.5) to always return `true`, because we are 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 47–53 execute if the deposit slot receives an envelope. Lines 47–50 display an appropriate message to the user. Line 53 then credits the deposit amount to the user's account in the database. Lines 57–58 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.

### J.13 Class `ATMCaseStudy`

Class `ATMCaseStudy` (Fig. J.12) is a simple class that allows us to start, or “turn on,” the ATM and test the implementation of our ATM system model. Class `ATMCaseStudy`'s `main` method (lines 7–11) does nothing more than instantiate a new ATM object named `theATM` (line 9) and invoke its `run` method (line 10) to start the ATM.

### J.14 Wrap-Up

Congratulations on completing the entire software engineering ATM case study! We hope you found this experience to be valuable and that it reinforced many of the concepts that you learned in Chapters 1–10. We would sincerely appreciate your comments, criticisms and suggestions. You can reach us at [deitel@deitel.com](mailto:deitel@deitel.com). We will respond promptly.

```

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

```

Fig. J.12 | `ATMCaseStudy.java` starts the ATM.