Lesson 10
ATM Case Study Part 2: Implementing an Object-Oriented Design

Assoc. Prof. Marenglen Biba

OBJECTIVES

In this chapter you'll:

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

13.1 Introduction

- Section 13.2 shows how to convert class diagrams to Java code.
- Section 13.3 tunes the design with inheritance and polymorphism.
- Section 13.4 presents a full Java code implementation of the ATM software.

13.2 Starting to Program the Classes of the ATM System

- ***Visibility***
- 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`.
  - 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 class normally should be `private`.

13.2 Starting to Program the Classes of the ATM System (cont.)

- 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.

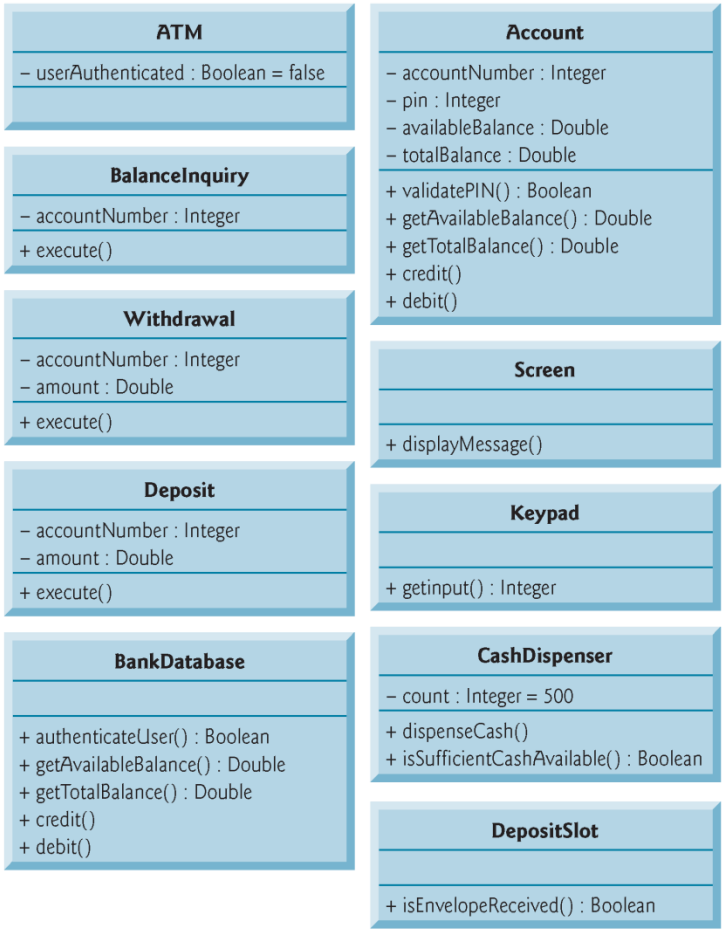

Fig. 13.1 | Class diagram with visibility markers.

13.2 Starting to Program the Classes of the ATM System (cont.)

- ***Navigability***
- The class diagram in Fig. 13.2 further refines the relationships among classes in the ATM system by adding navigability arrows to the association lines.
- Navigability arrows
  - represented as arrows in the class diagram
  - indicate in the direction which an association can be traversed.
- Programmers use navigability arrows to determine which objects need references to other objects.
- Associations that have navigability arrows at both ends indicate bidirectional navigability — navigation can proceed in either direction across the association.

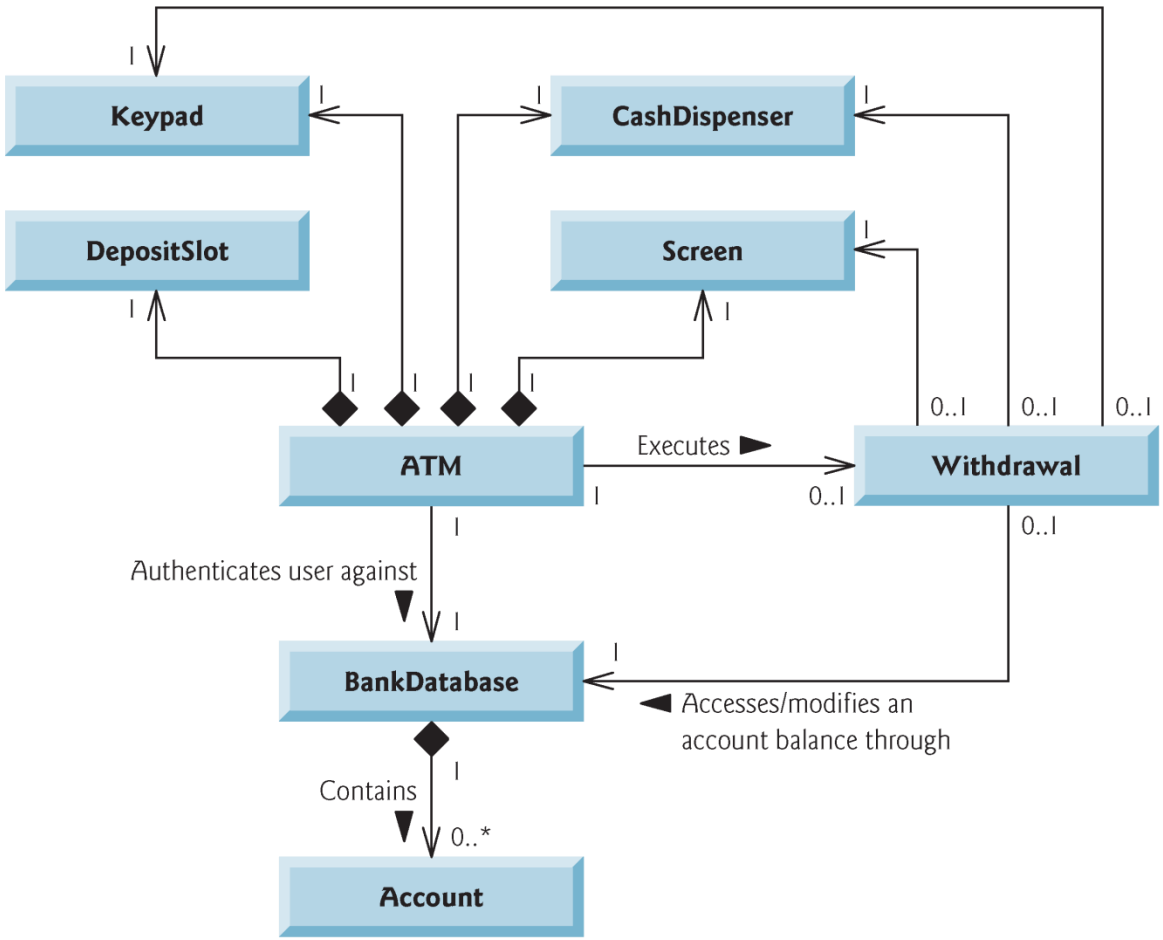

Fig. 13.2 | Class diagram with navigability arrows.

13.2 Starting to Program the Classes of the ATM System (cont.)

- ***Implementing the ATM System from Its UML Design***
- We are now ready to begin implementing the ATM system.
- Convert the classes in the diagrams of Fig. 13.1 and Fig. 13.2 into Java code.
- The code will represent the "skeleton" of the system.

13.2 Starting to Program the Classes of the ATM System (cont.)

- 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 (Fig. 13.3).
  - 2. Use the attributes located in the second compartment to declare the instance variables (Fig. 13.4).
  - 3. Use the associations described in the class diagram to declare the references to other objects (Fig. 13.5).
  - 4. Use the operations located in the third compartment of Fig. 13.1 to declare the shells of the methods (Fig. 13.6). If we have not yet specified a return type for an operation, we declare the method with return type `void`. Refer to the class diagrams of Figs. 12.17–12.21 to declare any necessary parameters.

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

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

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

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

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

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

```
1  // Class Withdrawal represents an ATM withdrawal transaction
2  public class Withdrawal
3  {
4     // attributes
5     private int accountNumber; // account to withdraw funds from
6     private double amount; // amount to withdraw
7
8     // references to associated objects
9     private Screen screen; // ATM's screen
10    private Keypad keypad; // ATM's keypad
11    private CashDispenser cashDispenser; // ATM's cash dispenser
12    private BankDatabase bankDatabase; // account info database
13
14    // no-argument constructor
15    public Withdrawal()
16    {
17    } // end no-argument Withdrawal constructor
18
19    // operations
20    public void execute()
21    {
22    } // end method execute
23 } // end class Withdrawal
```

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

13.3 Incorporating Inheritance and Polymorphism into the ATM System

- To apply inheritance, look for commonality among classes in the system.
- Create an inheritance hierarchy to model similar (yet not identical) classes in a more elegant and efficient manner.
- Modify class diagram to incorporate the new inheritance relationships.
- Translate updated design into Java code.

13.3 Incorporating Inheritance and Polymorphism into the ATM System (cont.)

- Problem of representing a financial transaction in the system.
- Created three individual transaction classes —`BalanceInquiry`, `Withdrawal` and `Deposit`—to represent the transactions that the ATM system can perform.
- Figure 13.7 shows the attributes and operations of classes `BalanceInquiry`, `Withdrawal` and `Deposit`.
  - Each has 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.
- `BalanceInquiry`, `Withdrawal` and `Deposit` represent *types of transactions*.

13.3 Incorporating Inheritance and Polymorphism into the ATM System (cont.)

- Figure 13.7 reveals commonality among the transaction classes.
- Use inheritance to factor out the common features.
- Place the common functionality in a superclass, `Transaction`, that classes `BalanceInquiry`, `Withdrawal-` and `Deposit` extend.

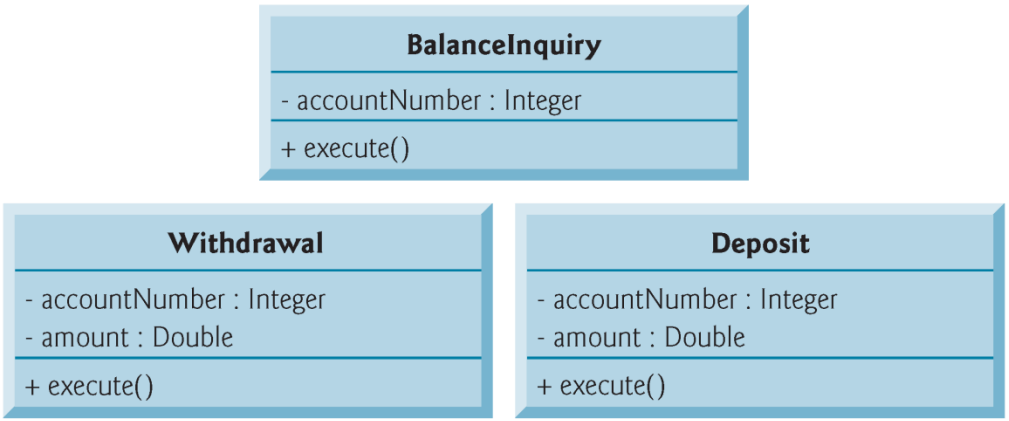

Fig. 13.7 | Attributes and operations of `BalanceInquiry`, `Withdrawal` and `Deposit`.

13.3 Incorporating Inheritance and Polymorphism into the ATM System (cont.)

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

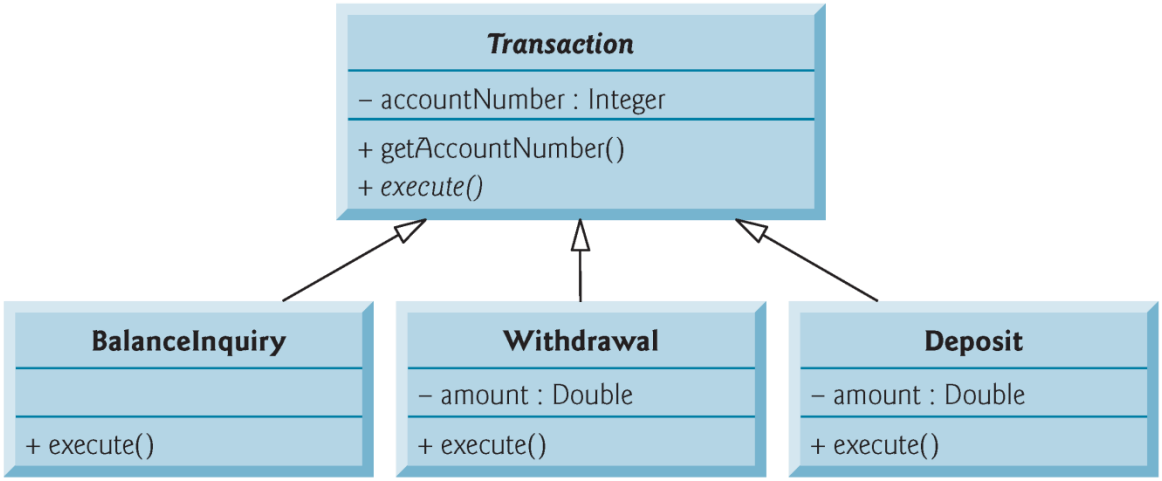

Fig. 13.8 | Class diagram modeling generalization of superclass `Transaction` and subclasses `BalanceInquiry`, `Withdrawal` and `Deposit`. Note that abstract class names (e.g., `Transaction`) and method names (e.g., `execute` in class `Transaction`) appear in italics.

13.3 Incorporating Inheritance and Polymorphism into the ATM System (cont.)

- Polymorphism provides the `ATM` with an elegant way to execute all transactions "in the general."
- The polymorphic approach also makes the system easily extensible.
- To create a new transaction type, just create an additional `Transaction` subclass that overrides the `execute` method with a version of the method appropriate for executing the new transaction type.

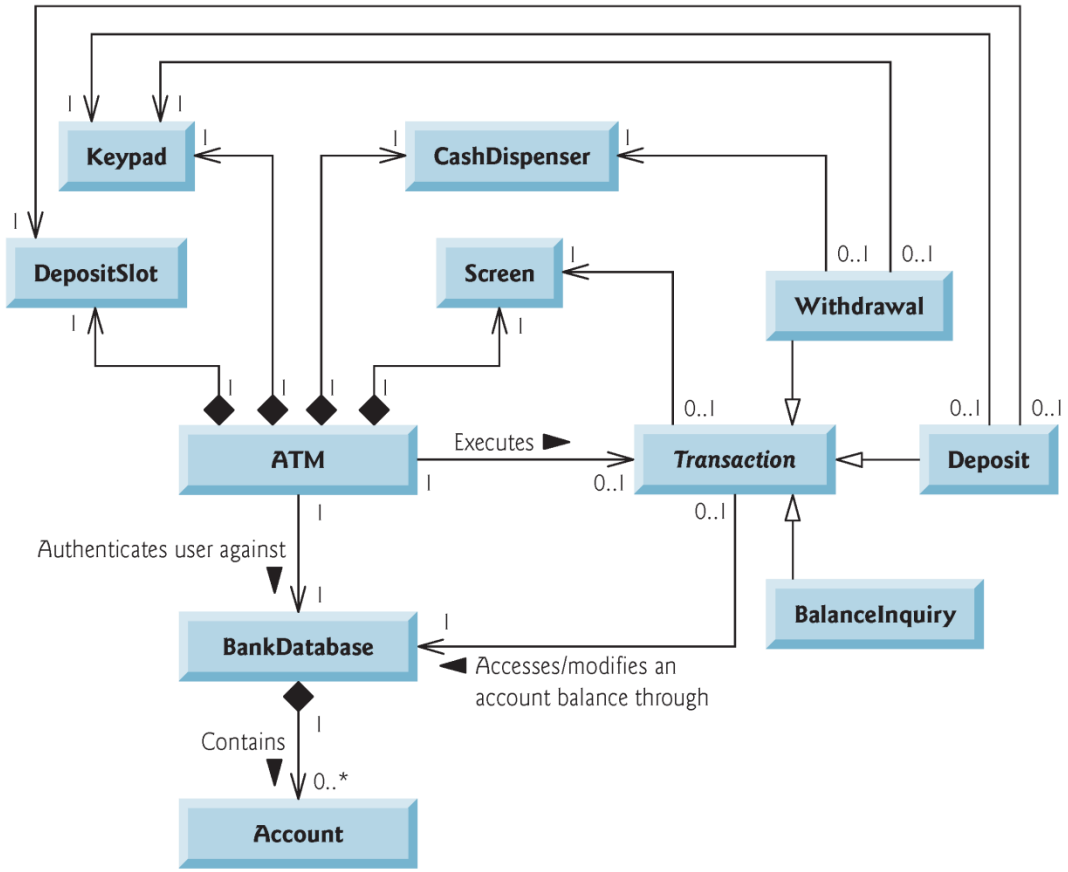

Fig. 13.9 | Class diagram of the ATM system (incorporating inheritance). Note that the abstract class name `Transaction` appears in italics.

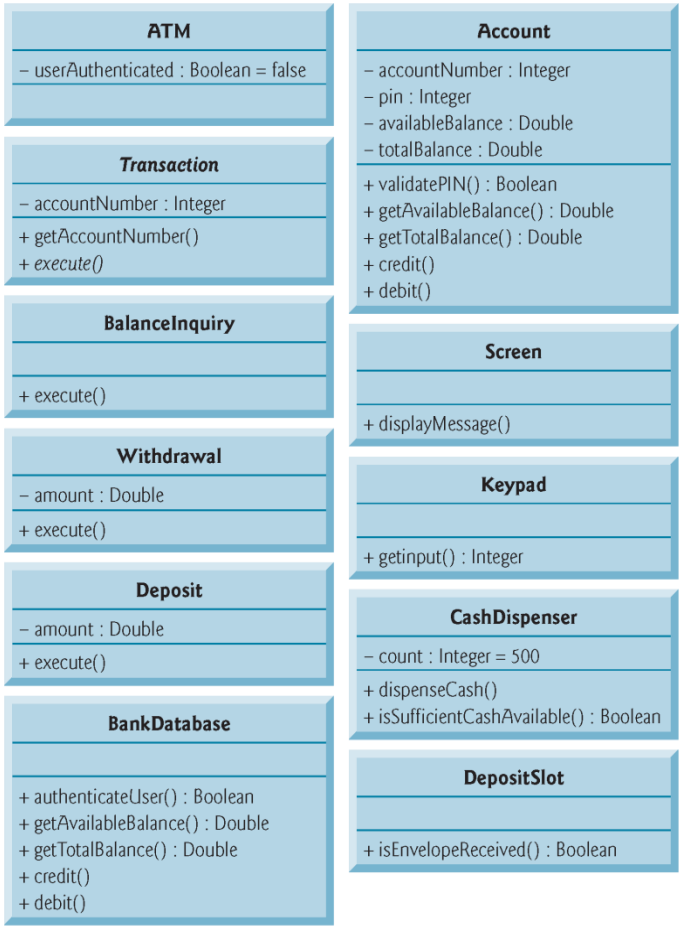

Fig. 13.10 | Class diagram with attributes and operations (incorporating inheritance). Note that the abstract class name `Transaction` and the abstract

13.3.1 Implementing the ATM System Design (Incorporating Inheritance)

- Figure 13.11 shows the declaration of class `Withdrawal`.

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

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

13.3.1 Implementing the ATM System Design (Incorporating Inheritance) (cont.)

- Figure 13.12 is the Java code for class `Withdrawal` from Fig. 13.9 and Fig. 13.10.

```
1  // Withdrawal.java
2  // Generated using the class diagrams in Fig. 13.9 and Fig. 13.10
3  public class Withdrawal extends Transaction
4  {
5     // attributes
6     private double amount; // amount to withdraw
7     private Keypad keypad; // reference to keypad
8     private CashDispenser cashDispenser; // reference to cash dispenser
9
10    // no-argument constructor
11    public Withdrawal()
12    {
13    } // end no-argument Withdrawal constructor
14
15    // method overriding execute
16    @Override
17    public void execute()
18    {
19    } // end method execute
20 } // end class Withdrawal
```

Fig. 13.12 | Java code for class `Withdrawal` based on Figs. 13.9 and 13.10.

13.5 ATM Case Study Implementation (cont.)

- Our ATM design does not specify all the program logic and may not specify all the attributes and operations required to complete the ATM implementation.
  - This is a normal part of the object-oriented design process.
- As we implement the system, we complete the program logic and add attributes and behaviors as necessary to construct the ATM system specified by the requirements document in Section 12.2.
- The Java application (`ATMCaseStudy`) starts the ATM and puts the other classes in the system in use.

Lab Session

- Implementation in NetBeans of the Case Study

13.5.1 Class ATM

- Class `ATM` (Fig. 13.13) represents the ATM as a whole.
- 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.
- Lines 8–12 declare reference-type attributes corresponding to the `ATM` class's associations modeled in Fig. 13.9.
  - These attributes allow the ATM to access its parts (i.e., its `Screen`, `Keypad`, `CashDispenser` and `DepositSlot`) and interact with the bank's account-information database (i.e., a `BankDatabase` object).

```
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
```

Fig. 13.13 | Class ATM represents the ATM. (Part 1 of 7.)

```
24          currentAccountNumber = 0; // no current account number to start
25          screen = new Screen(); // create screen
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
```

Fig. 13.13 | Class ATM represents the ATM. (Part 2 of 7.)

```
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
```

Fig. 13.13 | Class ATM represents the ATM. (Part 3 of 7.)

```
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;
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             {
```

Fig. 13.13 | Class ATM represents the ATM. (Part 4 of 7.)

```
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
```

Fig. 13.13 | Class ATM represents the ATM. (Part 5 of 7.)

```
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
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;
```

Fig. 13.13 | Class ATM represents the ATM. (Part 6 of 7.)

```
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. 13.13 | Class ATM represents the ATM. (Part 7 of 7.)

13.5.2 Class `Screen`

- Class `Screen` (Fig. 13.14) represents the screen of the ATM and encapsulates all aspects of displaying output to the user.
- 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 `displayDollar-Amount`.

```
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. 13.14 | Class `Screen` represents the screen of the ATM.

13.5.3 Class Keypad

- Class `Keypad` (Fig. 13.15) represents the keypad of the ATM and is responsible for receiving all user input.
- 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  {
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. 13.15 | Class `Keypad` represents the ATM's keypad.

13.5.4 Class `CashDispenser`

- Class `CashDispenser` (Fig. 13.16) represents the cash dispenser of the ATM.
- Constant `INITIAL_COUNT` indicates the initial count of bills in the cash dispenser when the ATM starts (i.e., `500`).
- 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  {
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
```

Fig. 13.16 | Class `CashDispenser` represents the ATM's cash dispenser. (Part 1 of 2.)

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

13.5.5 Class `DepositSlot`

- Class `DepositSlot` (Fig. 13.17) represents the ATM's deposit slot.
- `DepositSlot` has no attributes and only one method—`isEnvelopeReceived` (lines 8–11)—which indicates whether a deposit envelope was received.

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

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

13.5.6 Class `Account`

- Class `Account` (Fig. 13.18) represents a bank account.
- Each `Account` has four attributes (modeled in Fig. 13.10)—`accountNumber`, `pin`, `availableBalance` and `totalBalance`.
- Variable `availableBalance` represents the amount of funds available for withdrawal.
- Variable `totalBalance` represents the amount of funds available, plus the amount of deposited funds still pending confirmation or clearance.

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

Fig. 13.18 | Class `Account` represents a bank account. (Part 1 of 3.)

```
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
```

Fig. 13.18 | Class `Account` represents a bank account. (Part 2 of 3.)

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

Fig. 13.18 | Class `Account` represents a bank account. (Part 3 of 3.)

13.5.7 Class `BankDatabase`

- Class `BankDatabase` (Fig. 13.19) models the bank's database with which the ATM interacts to access and modify a user's account information.
- We determine one reference-type attribute for class `BankDatabase` based on its composition relationship with class `Account`.

```
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       {
```

Fig. 13.19 | Class `BankDatabase` represents the bank's account information database. (Part 1 of 3.)

```
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
```

Fig. 13.19 | Class `BankDatabase` represents the bank's account information database. (Part 2 of 3.)

```
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
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 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. 13.19 | Class `BankDatabase` represents the bank's account information database. (Part 3 of 3.)

13.5.8 Class `Transaction`

- Class `Transaction` (Fig. 13.20) is an abstract superclass that represents the notion of an ATM transaction.
- It contains the common features of subclasses `BalanceInquiry`, `Withdrawal` and `Deposit`.
- The class has three `public` *get* methods—`getAccountNumber` (lines 20–23), `get-Screen` (lines 26–29) and `getBankDatabase` (lines 32–35).
  - These are inherited by `Transaction` subclasses and used to gain access to class `Transaction`'s `private` attributes.

```
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
```

Fig. 13.20 | Abstract superclass `Transaction` represents an ATM transaction. (Part 1 of 2.)

```
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. 13.20 | Abstract superclass `Transaction` represents an ATM transaction. (Part 2 of 2.)

13.5.9 Class `BalanceInquiry`

- Class `BalanceInquiry` (Fig. 13.21) extends `Transaction` and represents a balance-inquiry ATM transaction.
- `BalanceInquiry` does not have any attributes of its own, but it inherits `Transaction` attributes `accountNumber`, `screen` and `bankDatabase`, which are accessible through `Transaction`'s `public` *get* methods.

```
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    @Override
15    public void execute()
16    {
17       // get references to bank database and screen
18       BankDatabase bankDatabase = getBankDatabase();
19       Screen screen = getScreen();
20
21       // get the available balance for the account involved
22       double availableBalance =
23          bankDatabase.getAvailableBalance( getAccountNumber() );
```

Fig. 13.21 | Class `BalanceInquiry` represents a balance-inquiry ATM transaction. (Part 1 of 2.)

```
24
25         // get the total balance for the account involved
26         double totalBalance =
27            bankDatabase.getTotalBalance( getAccountNumber() );
28
29         // display the balance information on the screen
30         screen.displayMessageLine( "\nBalance Information:" );
31         screen.displayMessage( " - Available balance: " );
32         screen.displayDollarAmount( availableBalance );
33         screen.displayMessage( "\n - Total balance:     " );
34         screen.displayDollarAmount( totalBalance );
35         screen.displayMessageLine( "" );
36      } // end method execute
37   } // end class BalanceInquiry
```

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

13.5.10 Class `Withdrawal`

- Class `Withdrawal` (Fig. 13.22) extends `Transaction` and represents a withdrawal ATM transaction.
- Figure 13.9 models associations between class `Withdrawal` and classes `Keypad` and `CashDispenser`, for which lines 7–8 implement reference-type attributes `keypad` and `cashDispenser`, respectively.

```
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;
```

Fig. 13.22 | Class `Withdrawal` represents a withdrawal ATM transaction. (Part 1 of 7.)

```
23          cashDispenser = atmCashDispenser;
24       } // end Withdrawal constructor
25
26       // perform transaction
27       @Override
28       public void execute()
29       {
30          boolean cashDispensed = false; // cash was not dispensed yet
31          double availableBalance; // amount available for withdrawal
32
33          // get references to bank database and screen
34          BankDatabase bankDatabase = getBankDatabase();
35          Screen screen = getScreen();
36
37          // loop until cash is dispensed or the user cancels
38          do
39          {
40             // obtain a chosen withdrawal amount from the user
41             amount = displayMenuOfAmounts();
42
43             // check whether user chose a withdrawal amount or canceled
44             if ( amount != CANCELED )
45             {
```

Fig. 13.22 | Class `Withdrawal` represents a withdrawal ATM transaction. (Part 2 of 7.)

```
46              // get available balance of account involved
47              availableBalance =
48                 bankDatabase.getAvailableBalance( getAccountNumber() );
49
50              // check whether the user has enough money in the account
51              if ( amount <= availableBalance )
52              {
53                 // check whether the cash dispenser has enough money
54                 if ( cashDispenser.isSufficientCashAvailable( amount ) )
55                 {
56                    // update the account involved to reflect the withdrawal
57                    bankDatabase.debit( getAccountNumber(), amount );
58
59                    cashDispenser.dispenseCash( amount ); // dispense cash
60                    cashDispensed = true; // cash was dispensed
61
62                    // instruct user to take cash
63                    screen.displayMessageLine( "\nYour cash has been" +
64                       " dispensed. Please take your cash now." );
65                 } // end if
```

Fig. 13.22 | Class `Withdrawal` represents a withdrawal ATM transaction. (Part 3 of 7.)

```
66                  else // cash dispenser does not have enough cash
67                     screen.displayMessageLine(
68                        "\nInsufficient cash available in the ATM." +
69                        "\n\nPlease choose a smaller amount." );
70               } // end if
71               else // not enough money available in user's account
72               {
73                  screen.displayMessageLine(
74                     "\nInsufficient funds in your account." +
75                     "\n\nPlease choose a smaller amount." );
76               } // end else
77            } // end if
78            else // user chose cancel menu option
79            {
80               screen.displayMessageLine( "\nCanceling transaction..." );
81               return; // return to main menu because user canceled
82            } // end else
83         } while ( !cashDispensed );
84
85      } // end method execute
86
```

Fig. 13.22 | Class `Withdrawal` represents a withdrawal ATM transaction. (Part 4 of 7.)

```
87      // display a menu of withdrawal amounts and the option to cancel;
88      // return the chosen amount or 0 if the user chooses to cancel
89      private int displayMenuOfAmounts()
90      {
91         int userChoice = 0; // local variable to store return value
92
93         Screen screen = getScreen(); // get screen reference
94
95         // array of amounts to correspond to menu numbers
96         int[] amounts = { 0, 20, 40, 60, 100, 200 };
97
98         // loop while no valid choice has been made
99         while ( userChoice == 0 )
100        {
101           // display the menu
102           screen.displayMessageLine( "\nWithdrawal Menu:" );
103           screen.displayMessageLine( "1 - $20" );
104           screen.displayMessageLine( "2 - $40" );
105           screen.displayMessageLine( "3 - $60" );
106           screen.displayMessageLine( "4 - $100" );
107           screen.displayMessageLine( "5 - $200" );
108           screen.displayMessageLine( "6 - Cancel transaction" );
109           screen.displayMessage( "\nChoose a withdrawal amount: " );
```

Fig. 13.22 | Class `Withdrawal` represents a withdrawal ATM transaction. (Part 5 of 7.)

```
110
111          int input = keypad.getInput(); // get user input through keypad
112
113          // determine how to proceed based on the input value
114          switch ( input )
115          {
116             case 1: // if the user chose a withdrawal amount
117             case 2: // (i.e., chose option 1, 2, 3, 4 or 5), return the
118             case 3: // corresponding amount from amounts array
119             case 4:
120             case 5:
121                userChoice = amounts[ input ]; // save user's choice
122                break;
123             case CANCELED: // the user chose to cancel
124                userChoice = CANCELED; // save user's choice
125                break;
126             default: // the user did not enter a value from 1-6
127                screen.displayMessageLine(
128                   "\nInvalid selection. Try again." );
129          } // end switch
130       } // end while
131
```

Fig. 13.22 | Class `Withdrawal` represents a withdrawal ATM transaction. (Part 6 of 7.)

```
132         return userChoice; // return withdrawal amount or CANCELED
133      } // end method displayMenuOfAmounts
134   } // end class Withdrawal
```

Fig. 13.22 | Class `Withdrawal` represents a withdrawal ATM transaction. (Part 7 of 7.)

13.5.11 Class `Deposit`

- Class `Deposit` (Fig. 13.23) extends `Transaction` and represents a deposit transaction.
- 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. 13.9.
- Line 9 declares a constant `CANCELED` that corresponds to the value a user enters to cancel.

```
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
```

Fig. 13.23 | Class `Deposit` represents a deposit ATM transaction. (Part 1 of 4.)

```
24      // perform transaction
25      @Override
26      public void execute()
27      {
28         BankDatabase bankDatabase = getBankDatabase(); // get reference
29         Screen screen = getScreen(); // get reference
30
31         amount = promptForDepositAmount(); // get deposit amount from user
32
33         // check whether user entered a deposit amount or canceled
34         if ( amount != CANCELED )
35         {
36            // request deposit envelope containing specified amount
37            screen.displayMessage(
38               "\nPlease insert a deposit envelope containing " );
39            screen.displayDollarAmount( amount );
40            screen.displayMessageLine( "." );
41
42            // receive deposit envelope
43            boolean envelopeReceived = depositSlot.isEnvelopeReceived();
44
45            // check whether deposit envelope was received
46            if ( envelopeReceived )
47            {
48               screen.displayMessageLine( "\nYour envelope has been " +
```

Fig. 13.23 | Class `Deposit` represents a deposit ATM transaction. (Part 2 of 4.)

```
49                "received.\nNOTE: The money just deposited will not " +
50                "be available until we verify the amount of any " +
51                "enclosed cash and your checks clear." );
52
53             // credit account to reflect the deposit
54             bankDatabase.credit( getAccountNumber(), amount );
55          } // end if
56          else // deposit envelope not received
57          {
58             screen.displayMessageLine( "\nYou did not insert an " +
59                "envelope, so the ATM has canceled your transaction." );
60          } // end else
61       } // end if
62       else // user canceled instead of entering amount
63       {
64          screen.displayMessageLine( "\nCanceling transaction..." );
65       } // end else
66    } // end method execute
67
68    // prompt user to enter a deposit amount in cents
69    private double promptForDepositAmount()
70    {
71       Screen screen = getScreen(); // get reference to screen
72
```

Fig. 13.23 | Class `Deposit` represents a deposit ATM transaction. (Part 3 of 4.)

```
73         // display the prompt
74         screen.displayMessage( "\nPlease enter a deposit amount in " +
75            "CENTS (or 0 to cancel): " );
76         int input = keypad.getInput(); // receive input of deposit amount
77
78         // check whether the user canceled or entered a valid amount
79         if ( input == CANCELED )
80            return CANCELED;
81         else
82         {
83            return ( double ) input / 100; // return dollar amount
84         } // end else
85      } // end method promptForDepositAmount
86   } // end class Deposit
```

Fig. 13.23 | Class `Deposit` represents a deposit ATM transaction. (Part 4 of 4.)

13.5.12 Class ATMCaseStudy

- Class `ATMCaseStudy` (Fig. 13.24) is a simple class that allows us to start, or "turn on," the ATM and test the implementation of our ATM system model.

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

Fig. 13.24 | `ATMCaseStudy.java` starts the ATM.

End of class

- Readings
  - Chapter 12 and 13