Chapter 13
ATM Case Study Part 2: Implementing an Object-Oriented Design
Java How to Program, 8/e

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 (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.
  - Programmers use navigability arrows to determine which objects need references to other objects.
  - Associations that have navigability arrows at both ends or have none at all indicate **bidirectional navigability**—navigation can proceed in either direction across the association.

![Class diagram with visibility markers](image-url)
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.
```

(C) 2010 Pearson Education, Inc. All rights reserved.
Fig. 13.4  |  Java code for class \texttt{Withdrawal} based on Figs. 13.1–13.2.

```java
public class Withdrawal {

    // attributes
    private int accountNumber; // account to withdraw funds from
    private double amount; // amount to withdraw

    // no-argument constructor
    public Withdrawal() {
    }

    // end no-argument \texttt{Withdrawal} constructor
}

// end class \texttt{Withdrawal}
```

(C) 2010 Pearson Education, Inc. All rights reserved.

Fig. 13.5  |  Java code for class \texttt{Withdrawal} based on Figs. 13.1–13.2.

```java
public class Withdrawal {

    // attributes
    private int accountNumber; // account to withdraw funds from
    private double amount; // amount to withdraw

    // references to associated objects
    private Screen screen; // ATM's screen
    private Keypad keypad; // ATM's keypad
    private CashDispenser cashDispenser; // ATM's cash dispenser
    private BankDatabase bankDatabase; // account info database

    // no-argument constructor
    public Withdrawal() {
    }

    // end no-argument \texttt{Withdrawal} constructor
}

// end class \texttt{Withdrawal}
```

(C) 2010 Pearson Education, Inc. All rights reserved.
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.
Fig. 13.7 | Attributes and operations of BalanceInquiry, Withdrawal and Deposit.

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

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

```java
1 // Class Withdrawal represents an ATM withdrawal transaction
2 public class Withdrawal extends Transaction
3 {
4     // end class Withdrawal
```
### 13.4 ATM Case Study Implementation

- Consider the classes in the order in which we identified them in Section 12.3—**ATM**, **Screen**, **Keypad**, **CashDispenser**, **Deposit-Slot**, **Account**, **BankDatabase**, **Transaction**, **BalanceInquiry**, **Withdrawal** and **Deposit**.

- Apply the guidelines discussed in Sections 13.2–13.3 to code these classes based on how we modeled them in the UML class diagrams of Figs. 13.9 and 13.10.

---

```java
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     // no-argument constructor
10    public Withdrawal() {
11         // end no-argument Withdrawal constructor
12     }
13     // method overriding execute
14     @Override
15     public void execute() {
16         // end method execute
17     }
18     // end class Withdrawal
19 }
```

*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.

```java
// ATM.java
public class ATM {
    private boolean userAuthenticated; // whether user is authenticated
    private int currentAccountNumber; // current user's account number
    private Screen screen; // ATM's screen
    private Keypad keypad; // ATM's keypad
    private CashDispenser cashDispenser; // ATM's cash dispenser
    private DepositSlot depositSlot; // ATM's deposit slot
    private BankDatabase bankDatabase; // account information database

    // constants corresponding to main menu options
    private static final int BALANCE_INQUIRY = 1;
    private static final int WITHDRAWAL = 2;
    private static final int DEPOSIT = 3;
    private static final int EXIT = 4;

    // no-argument ATM constructor initializes instance variables
    public ATM() {
        userAuthenticated = false; // user is not authenticated to start
    }
}
```

Fig. 13.13 | Class ATM represents the ATM (Part 1 of 7.)
```java
24. currentAccountNumber = 0; // no current account number to start
25. keypad = new Keypad(); // create keypad
26. cashDispenser = new CashDispenser(); // create cash dispenser
27. depositSlot = new DepositSlot(); // create deposit slot
28. bankDatabase = new BankDatabase(); // create acct info database
29. } // end no-argument ATM constructor
30. 
31. // start ATM
32. public void run()
33. {
34.   // welcome and authenticate user; perform transactions
35.   while ( true )
36.   {
37.     // loop while user is not yet authenticated
38.     while ( !userAuthenticated )
39.     {
40.       screen.displayMessageLine( "Welcome!" );
41.       authenticateUser(); // authenticate user
42.     } // end while
43.     performTransactions(); // user is now authenticated
44.     userAuthenticated = false; // reset before next ATM session
45.   }
46. }
```

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

---

```java
47. currentAccountNumber = 0; // reset before next ATM session
48. screen.displayMessageLine( "Thank 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( "Please enter your account number: " );
56.   int accountNumber = keypad.getInput(); // input account number
57.   screen.displayMessage( "Enter your PIN: " ); // prompt for PIN
58.   int pin = keypad.getInput(); // input PIN
59. 
60.   // set userAuthenticated to boolean value returned by database
61.   userAuthenticated = bankDatabase.authenticateUser( accountNumber, pin );
62. 
63.   // check whether authentication succeeded
64.   if ( userAuthenticated )
65.     {
66.       currentAccountNumber = accountNumber; // save user's account #
67.     } // end if
```

Fig. 13.13 | Class ATM represents the ATM. (Part 3 of 7.)
else
    screen.displayMessageLine("Invalid account number or PIN. Please try again.");
// end method authenticateUser

// display the main menu and perform transactions
private void performTransactions()
{
    // Local variable to store transaction currently being processed
    Transaction currentTransaction = null;
    boolean userExited = false; // user has not chosen to exit
    // loop while user has not chosen option to exit system
    while ( !userExited )
    {
        // show main menu and get user selection
        int mainMenuSelection = displayMainMenu();
        // decide how to proceed based on user's menu selection
        switch ( mainMenuSelection )
        {
            // user chose to perform one of three transaction types
            case BALANCE_INQUIRY:
                case WITHDRAWAL:
                case DEPOSIT:
                {
                    // Initialize as new object of chosen type
                    currentTransaction = createTransaction( mainMenuSelection );
                    currentTransaction.execute(); // execute transaction
                }
                break;
            case EXIT: // user chose to terminate session
                screen.displayMessageLine( "\nExiting the system...\n" );
                userExited = true; // this ATM session should end
                break;
            default: // user did not enter an integer from 1-4
                screen.displayMessageLine( "\nYou did not enter a valid selection. Try again.\n" );
                break;
        }
    }
} // end method performTransactions

Fig. 13.13 | Class ATM represents the ATM. (Part 4 of 7.)
private int displayMainMenu()
{
    screen.displayMessageLine("Main menu:");
    screen.displayMessageLine("1 - View my balance");
    screen.displayMessageLine("2 - Withdraw cash");
    screen.displayMessageLine("3 - Deposit funds");
    screen.displayMessageLine("4 - Exit");
    screen.displayMessageLine("Enter a choice:");
    return keypad.getlnput() // return user's selection
}

// return object of specified Transaction subclass
private Transaction createTransaction( int type )
{
    Transaction temp = null; // temporary Transaction variable
    switch ( type )
    {
        case BALANCE_INQUIRY: // create new BalanceInquiry transaction
            temp = new BalanceInquiry(
                currentAccountNumber, screen, bankDatabase );
            break;

        case WITHDRAWAL: // create new Withdrawal transaction
            temp = new Withdrawal( currentAccountNumber, screen, bankDatabase, keypad, cashDispenser );
            break;

        case DEPOSIT: // create new Deposit transaction
            temp = new Deposit( currentAccountNumber, screen, bankDatabase, keypad, depositSlot );
            break;

    } // end switch
    return temp; // return the newly created object
} // end method createTransaction

} // end class ATM
public class Screen {
    // display a message without a carriage return
    public void displayMessage(String message) {
        System.out.print(message);
    }
    // end method displayMessage

    // display a message with a carriage return
    public void displayMessageLine(String message) {
        System.out.println(message);
    }
    // end method displayMessageLine

    // displays a dollar amount
    public void displayDollarAmount(double amount) {
        System.out.printf("%.2f", amount);
    }
    // end method displayDollarAmount

    // Screen.java
    // Represents the screen of the ATM
    // (C) 2010 Pearson Education, Inc. All rights reserved.

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

```java
1 // CashDispenser.java
2 // Represents the cash dispenser of the ATM
3 public class CashDispenser
4 {
5   // the default initial number of bills in the cash dispenser
6   private final static int INITIAL_COUNT = 500;
7   private int count; // number of $20 bills remaining
8
9 // no-argument CashDispenser constructor initializes count to default
10 public CashDispenser()
11 {
12   count = INITIAL_COUNT; // set count attribute to default
13 } // end CashDispenser constructor
14
15 // simulates dispensing of specified amount of cash
16 public void dispenseCash( int amount )
17 {
18   int billsRequired = amount / 20; // number of $20 bills required
19   count = billsRequired; // update the count of bills
20 } // end method dispenseCash
21

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

(C) 2010 Pearson Education, Inc. All rights reserved.
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.
// DepositSlot.java
// Represents the deposit slot of the ATM
public class DepositSlot {
  // indicates whether envelope was received (always returns true,
  // because this is only a software simulation of a real deposit slot)
  public boolean isEnvelopeReceived() {
    return true; // deposit envelope was received
  }
}

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

// Account.java
// Represents a bank account
public class Account {
  
  private int accountNumber; // account number
  private int pin; // PIN for authentication
  private double availableBalance; // funds available for withdrawal
  private double totalBalance; // funds available + pending deposits

  // Account constructor initializes attributes
  public Account( int theAccountNumber, int thePIN,
                   double theAvailableBalance, double theTotalBalance )
  {
    accountNumber = theAccountNumber;
    pin = thePIN;
    availableBalance = theAvailableBalance;
    totalBalance = theTotalBalance;
  }
}

Fig. 13.18  |  Class Account represents a bank account. (Part 1 of 3.)
if ( userPIN == pin )
   return true;
else
   return false;
} // end method validatePIN

// returns available balance
default double getAvailableBalance()
{
   return availableBalance;
} // end method getAvailableBalance

// returns the total balance
default double getTotalBalance()
{
   return totalBalance;
} // end method getTotalBalance

// credits an amount to the account
default void credit( double amount )
{
   totalBalance += amount; // add to total balance
} // end method credit

// debits an amount from the account
default void debit( double amount )
{
   availableBalance -= amount; // subtract from available balance
   totalBalance -= amount; // subtract from total balance
} // end method debit

// returns account number
default int getAccountNumber()
{
   return accountNumber;
} // end method getAccountNumber

// Class Account represents a bank account. (Part 2 of 3.)

// Class Account represents a bank account. (Part 3 of 3.)
// BankDatabase.java
// Represents the bank account information database
public class BankDatabase
{
    private Account[] accounts; // array of Accounts
    public BankDatabase()
    {
        accounts = new Account[2]; // just 2 accounts for testing
        accounts[0] = new Account(12345, 54321, 1000.0, 200.0);
        accounts[1] = new Account(98765, 67890, 200.0, 200.0);
        // end no-argument BankDatabase constructor

        // retrieve Account object containing specified account number
        private Account getAccount( int accountNumber )
        {
            for ( Account currentAccount : accounts )
            {
                // return current account if match found
                if ( currentAccount.getAccountNumber() == accountNumber )
                    return currentAccount;
            }
            // end for

            return null; // if no matching account was found, return null
        } // end method getAccount

        // determine whether user-specified account number and PIN match
        // those of an account in the database
        public boolean authenticateUser( int userAccountNumber, int userPIN )
        {
            Account userAccount = getAccount( userAccountNumber );
            // if account exists, return result of Account method validatePIN
            if ( userAccount != null )
            {
                return userAccount.validatePIN( userPIN );
            } // end if
            else
                return false; // account number not found, so return false
        } // end method authenticateUser

        // return current account if match found
        if ( currentAccount.getAccountNumber() == accountNumber )
            return currentAccount;
        } // end for

        return null; // if no matching account was found, return null
    } // end method getAccount

    // determine whether user-specified account number and PIN match
    // those of an account in the database
    public boolean authenticateUser( int userAccountNumber, int userPIN )
    {
        Account userAccount = getAccount( userAccountNumber );
        // if account exists, return result of Account method validatePIN
        if ( userAccount != null )
            return userAccount.validatePIN( userPIN );
        else
            return false; // account number not found, so return false
    } // end method authenticateUser

    // return current account if match found
    if ( currentAccount.getAccountNumber() == accountNumber )
        return currentAccount;
    } // end for

    return null; // if no matching account was found, return null
} // end method getAccount

// determine whether user-specified account number and PIN match
// those of an account in the database
public boolean authenticateUser( int userAccountNumber, int userPIN )
{
    Account userAccount = getAccount( userAccountNumber );
    // if account exists, return result of Account method validatePIN
    if ( userAccount != null )
    {
        return userAccount.validatePIN( userPIN );
    } // end if
    else
        return false; // account number not found, so return false
} // end method authenticateUser

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

(C) 2010 Pearson Education, Inc. All rights reserved.
• 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), getScreen (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.
```java
24.04.2012

// Transaction.java
// Abstract superclass Transaction represents an ATM transaction

public abstract class Transaction {
    private int accountNumber; // indicates account involved
    private Screen screen; // ATM's screen
    private BankDatabase bankDatabase; // account info database

    // Transaction constructor invoked by subclasses using super()
    public Transaction( int userAccountNumber, Screen atmScreen, 
                        BankDatabase atmBankDatabase )
        { accountNumber = userAccountNumber;
          screen = atmScreen;
          bankDatabase = atmBankDatabase;
        } // end Transaction constructor

    // return account number
    public int getAccountNumber()
    { return accountNumber;
    } // end method getAccountNumber

    // return reference to screen
    public Screen getScreen()
    { return screen;
    } // end method getScreen

    // return reference to bank database
    public BankDatabase getBankDatabase()
    { return bankDatabase;
    } // end method getBankDatabase

    // perform the transaction (overridden by each subclass)
    abstract public void execute();
} // end class Transaction
```

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

(C) 2010 Pearson Education, Inc. All rights reserved.

24.04.2012

```
24.04.2012
```

(C) 2010 Pearson Education, Inc. All rights reserved.
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.

```java
// BalanceInquiry.java
// Represents a balance-inquiry ATM transaction

public class BalanceInquiry extends Transaction {

    // BalanceInquiry constructor
    public BalanceInquiry( int userAccountNumber, Screen atmScreen,
                     BankDatabase atmBankDatabase ) {
        super( userAccountNumber, atmScreen, atmBankDatabase );
    }

    // performs the transaction
    @Override
    public void execute() {

        // get references to bank database and screen
        BankDatabase bankDatabase = getBankDatabase();
        Screen screen = getScreen();

        // get the available balance for the account involved
        double availableBalance =
             bankDatabase.getAvailableBalance( getAccountNumber() );

    }
}
```

Fig. 13.21 | Class BalanceInquiry represents a balance-inquiry ATM transaction. (Part 1 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.
public class Withdrawal extends Transaction {
    // amount to withdraw
    private int amount; // amount to withdraw
    private Keypad keypad; // reference to keypad
    private CashDispenser cashDispenser; // reference to cash dispenser

    // constant corresponding to menu option to cancel
    private final static int CANCELED = 6;

    // Withdrawal constructor
    public Withdrawal(int userAccountNumber, Screen atmScreen, BankDatabase atmBankDatabase, Keypad atmKeypad, CashDispenser atmCashDispenser)
    {
        // initialize superClass variables
        super(userAccountNumber, atmScreen, atmBankDatabase);

        // initialize references to keypad and cash dispenser
        keypad = atmKeypad;
    
    cashDispenser = atmCashDispenser;

    // end Withdrawal constructor

    // perform transaction
    @Override
    public void execute()
    {
        boolean cashDispensed = false; // cash was not dispensed yet
        double availableBalance; // amount available for withdrawal

        // get references to bank database and screen
        BankDatabase atmBankDatabase = getBankDatabase();
        Screen atmScreen = getScreen();

        // loop until cash is dispensed or the user cancels
        do
        {
            // obtain a chosen withdrawal amount from the user
            amount = displayMenuOfAmounts();

            // check whether user chose a withdrawal amount or canceled
            if (amount != CANCELED)
            {
                // display the amount to withdraw
                displayAmount(amount);

                // dispense the cash
                dispenseCash(amount, cashDispenser);

                cashDispensed = true;
                availableBalance = availableBalance + amount;

                // display available balance
                displayAvailableBalance(availableBalance);
            }
        }
        while (cashDispensed == false);
    
    // display menu of amount to withdraw
    private void displayMenuOfAmounts()
    {
        // display menu options
        displayMenuOptions();

        // get user's choice
        int choice = getChoice();

        // display the selected amount
        displaySelectedAmount(choice);
    
    // dispense the cash
    private void dispenseCash(int amount, CashDispenser cashDispenser)
    {
        // dispense the cash
        cashDispenser.dispenseCash(amount);

        // display transactions
        displayTransactions();
    
    // display available balance
    private void displayAvailableBalance(double availableBalance)
    {
        // display available balance
        displayAvailableBalance(availableBalance);

        // display the available balance
        displayAvailableBalance(availableBalance);
    
    // display menu options
    private void displayMenuOptions()
    {
        // display menu options
        displayMenuOptions();

        // display the menu options
        displayMenuOptions();
    
    // get user's choice
    private int getChoice()
    {
        // get user's choice
        int choice = getChoice();

        // display the selected option
        displaySelectedOption(choice);
    
    // display the selected option
    private void displaySelectedOption(int choice)
    {
        // display the selected option
        displaySelectedOption(choice);

        // display the selected option
        displaySelectedOption(choice);
// get available balance of account involved
availableBalance = bankDatabase.getAvailableBalance( getAccountNumber() );

// check whether the user has enough money in the account
if ( amount <= availableBalance )
{
  // check whether the cash dispenser has enough money
  if ( cashDispenser.isSufficientCashAvailable( amount ) )
  {
    // update the account involved to reflect the withdrawal
    bankDatabase.debit( getAccountNumber(), amount );
    cashDispenser.dispenseCash( amount ); // dispense cash
    cashDispensed = true; // cash was dispensed
  }

  // instruct user to take cash
  screen.displayMessageLine( \"Your cash has been \n\ndispensed. Please take your cash now.\" );
}

else // cash dispenser does not have enough cash
{
  screen.displayMessageLine( \"Insufficient cash available in the ATM.\n\nPlease choose a smaller amount.\" );
}

else // not enough money available in user's account
{
  screen.displayMessageLine( \"Insufficient funds in your account.\n\nPlease choose a smaller amount.\" );
}

} // end if

} // end method execute
// display a menu of withdrawal amounts and the option to cancel;
private int displayMenuOfAmounts()
{
    int userChoice = 0; // local variable to store return value
    Screen screen = getScreen(); // get screen reference
    // array of amounts to correspond to menu numbers
    int[] amounts = { 0, 20, 40, 60, 100, 200 }; // loop while no valid choice has been made
    while ( userChoice == 0 )
    {
        // display the menu
        screen.displayMessageLine( "\nWithdrawal Menu: " );
        screen.displayMessageLine( "1 - $20" );
        screen.displayMessageLine( "2 - $40" );
        screen.displayMessageLine( "3 - $60" );
        screen.displayMessageLine( "4 - $100" );
        screen.displayMessageLine( "5 - $200" );
        screen.displayMessageLine( "6 - Cancel transaction" );
        screen.displayMessageLine( "\nChoose a withdrawal amount: " );
    }
}

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

int input = keypad.getInput(); // get user input through keypad
// determine how to proceed based on the input value
switch ( input )
{
    case 1: // if the user chose a withdrawal amount
    case 2: // (i.e., chose option 1, 2, 3, 4 or 5), return the
    case 3: // corresponding amount from amounts array
    case 4:
    case 5:
        userChoice = amounts[ input ]; // save user's choice
        break;
    case CANCELED: // the user chose to cancel
        userChoice = CANCELED; // save user's choice
        break;
    default: // the user did not enter a value from 1-6
        screen.displayMessageLine( "\nInvalid selection. Try again." );
    } // end switch
} // end while

Fig. 13.22 | Class Withdrawal represents a withdrawal ATM transaction. (Part 6 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.

```java
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)
```java
24. // Deposit.java
25. // Represents a deposit ATM transaction
26. public class Deposit extends Transaction
27. {
28.     private double amount; // amount to deposit
29.     private Keypad keypad; // reference to keypad
30.     private DepositSlot depositSlot; // reference to deposit slot
31.     private final static int CANCELED = 0; // constant for cancel option
32. }
33. // Deposit constructor
34. public Deposit( int userAccountNumber, Screen atmScreen,
35.                 BankDatabase atmBankDatabase, Keypad atmKeypad,
36.                 DepositSlot atmDepositSlot )
37. {
38.     // initialize superclass variables
39.     super( userAccountNumber, atmScreen, atmBankDatabase );
40. }
41. // initialize references to keypad and deposit slot
42.     keypad = atmKeypad;
43.     depositSlot = atmDepositSlot;
44. } // end Deposit constructor

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.     amount = promptForDepositAmount(); // get deposit amount from user
31.     // check whether user entered a deposit amount or canceled
32.     if( amount != CANCELED )
33.         {
34.             // request deposit envelope containing specified amount
35.             screen.displayMessage( "\nPlease insert a deposit envelope containing " );
36.             screen.displayMessageLine( "." );
37.             // receive deposit envelope
38.             boolean envelopeReceived = depositSlot.isEnvelopeReceived();
39.             // check whether deposit envelope was received
40.             if( envelopeReceived )
41.                 {
42.                     screen.displayMessageLine( "Your envelope has been " +
```````````
"received.

NOTE: the money just deposited will not " +
"be available until we verify the amount of any " +
"enclosed cash and your checks clear."

// credit account to reflect the deposit
bankDatabase.credit( getAccountNumber(), amount );
}
else

// deposit envelope not received

{ 
    screen.displayMessageLine( "\nYou did not insert an " +
    "envelope, so the ATM has canceled your transaction." );
}

// end if
else

// user canceled instead of entering amount

{ 
    screen.displayMessageLine( "\nCanceling transaction..." );
}

// end else
}
// end method execute

// prompt user to enter a deposit amount in cents
private double promptForDepositAmount()

{ 
    Screen screen = getScreen(); // get reference to screen
Fig. 13.23 | Class Deposit represents a deposit ATM transaction. (Part 3 of 4.)

} // end if
else

{ 
    return ( double ) input / 100; // return dollar amount
}

} // end else

} // end method promptForDepositAmount

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

(C) 2010 Pearson Education, Inc. All rights reserved.
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.

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

**Fig. 13.24** ATMCaseStudy.java starts the ATM.