26

ATM Case Study, Part 2: Implementing an OO Design in C#

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 C# the UML-based object-oriented design of the ATM software.
- Study a detailed code walkthrough of the ATM software system that explains the implementation issues.

Outline

26.1 Introduction

In Chapter 25, we developed an object-oriented design for our ATM system. In this chapter, we take a deeper look at the details of programming with classes. We now begin implementing our object-oriented design by converting class diagrams to C# code. In the final case study section (Section 26.3), we modify the code to incorporate the object-oriented concepts of inheritance and polymorphism. We present the full C# code implementation in Section 26.4.

26.2 Starting to Program the Classes of the ATM System

Visibility

We now apply access modifiers to the members of our classes. In Chapter 4, we introduced access modifiers `public` and `private`. Access modifiers determine the **visibility**, or accessibility, of an object's attributes and operations to other objects. Before we can begin implementing our design, we must consider which attributes and methods of our classes should be `public` and which should be `private`.

In Chapter 4, we observed that attributes normally should be `private` and that methods invoked by clients of a class should be `public`. Methods that are called only by other methods of the class as "utility functions," however, should be `private`. The UML employs **visibility markers** for modeling the visibility of attributes and operations. Public visibility is indicated by placing a plus sign (+) before an operation or an attribute; a minus sign (–) indicates private visibility. Figure 26.1 shows our updated class diagram with visibility markers included. [*Note:* We do not include any operation parameters in Fig. 26.1. This is perfectly normal. Adding visibility markers does not affect the parameters already modeled in the class diagrams of Figs. 25.18–25.21.]

Navigability

Before we begin implementing our design in C#, we introduce an additional UML notation. The class diagram in Fig. 26.2 further refines the relationships among classes in the ATM system by adding navigability arrows to the association lines. **Navigability arrows** (represented as arrows with stick arrowheads in the class diagram) indicate in which direction an association can be traversed and are based on the collaborations modeled in communication and sequence diagrams (see Section 25.7). When implementing a system designed using the

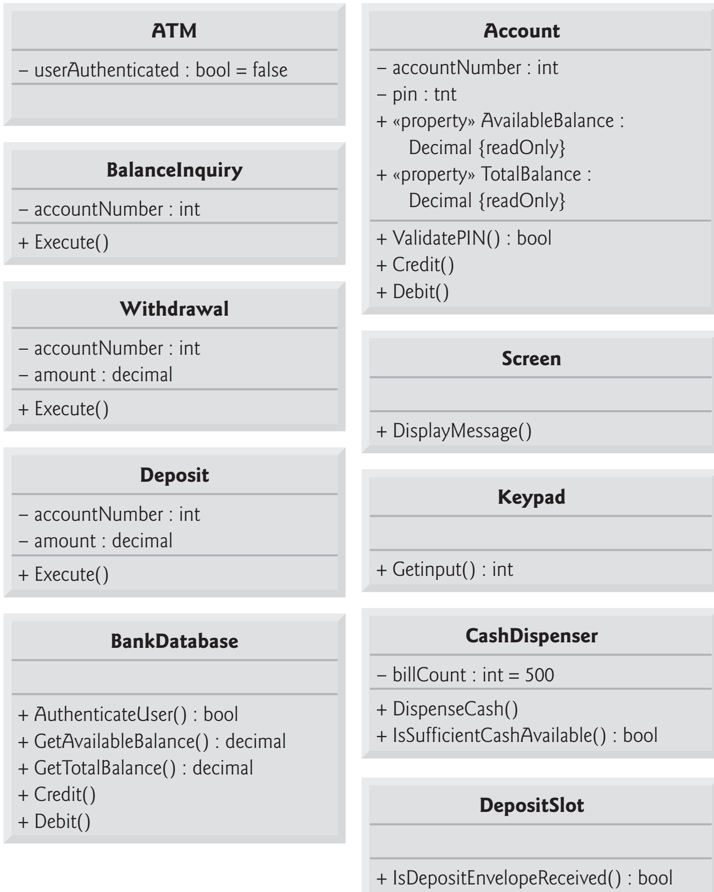

Fig. 26.1 | Class diagram with visibility markers.

UML, programmers use navigability arrows to help determine which objects need references to other objects. For example, the navigability arrow pointing from class `ATM` to class `BankDatabase` indicates that we can navigate from the former to the latter, thereby enabling the `ATM` to invoke the `BankDatabase`'s operations. However, since Fig. 26.2 does not contain a navigability arrow pointing from class `BankDatabase` to class `ATM`, the `BankDatabase` cannot access the `ATM`'s operations. Associations in a class diagram that have navigability arrows at both ends or do not have navigability arrows at all indicate **bidirectional navigability**—navigation can proceed in either direction across the association.

The class diagram of Fig. 26.2 omits classes `BalanceInquiry` and `Deposit` to keep the diagram simple. The navigability of the associations in which these classes participate closely parallels the navigability of class `Withdrawal`'s associations. Recall that `BalanceInquiry` has an association with class `Screen`. We can navigate from class `BalanceInquiry` to class `Screen` along this association, but we cannot navigate from class `Screen` to class `BalanceInquiry`. Thus, if we were to model class `BalanceInquiry` in Fig. 26.2, we would place a navigability arrow at class `Screen`'s end of this association. Also recall that class

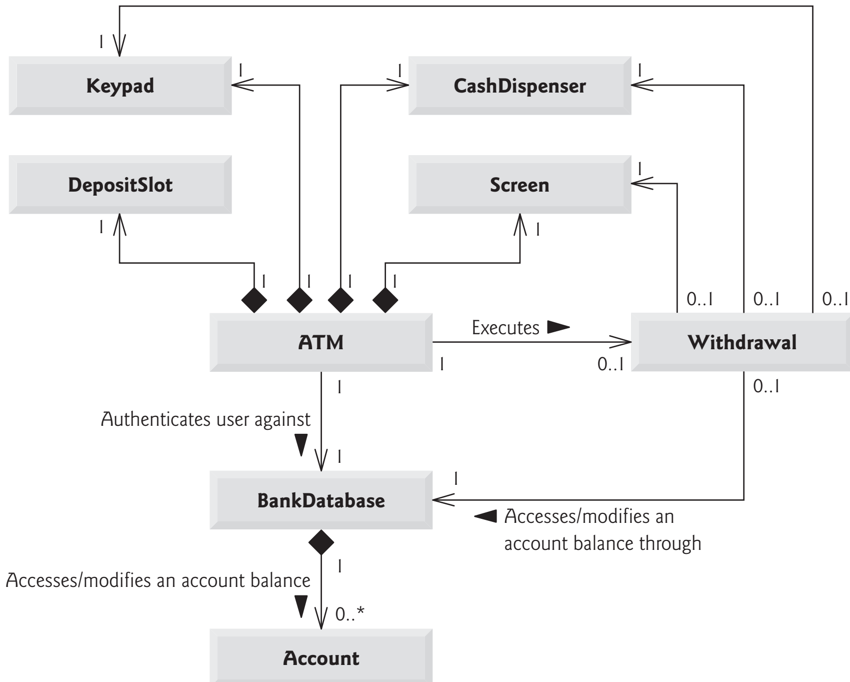

Fig. 26.2 | Class diagram with navigability arrows.

`Deposit` associates with classes `Screen`, `Keypad` and `DepositSlot`. We can navigate from class `Deposit` to each of these classes, but not vice versa. We therefore would place navigability arrows at the `Screen`, `Keypad` and `DepositSlot` ends of these associations. [*Note:* We model these additional classes and associations in our final class diagram in Section 26.3, after we have simplified the structure of our system by incorporating the object-oriented concept of inheritance.]

Implementing the ATM System from Its UML Design

We're now ready to begin implementing the ATM system. We first convert the classes in the diagrams of Fig. 26.1 and 26.2 into C# code. This code will represent the "skeleton" of the system. In Section 26.3, we modify the code to incorporate the object-oriented concept of inheritance. In Section 26.4, we present the complete working C# code that implements our object-oriented design.

As an example, we begin to develop the code for class `Withdrawal` from our design of class `Withdrawal` in Fig. 26.1. We use this figure to determine the attributes and operations of the class. We use the UML model in Fig. 26.2 to determine the associations among classes. We follow these four guidelines for each class:

1. Use the name located in the first compartment of a class in a class diagram to declare the class as a `public` class with an empty parameterless constructor—we include this constructor simply as a placeholder to remind us that most classes will need one or more constructors. In Section 26.4.10, when we complete a working

version of this class, we add any necessary arguments and code to the body of the constructor. Class `Withdrawal` initially yields the code in Fig. 26.3.

```
// Fig. 26.3: Withdrawal.cs
// Class Withdrawal represents an ATM withdrawal transaction
public class Withdrawal
{
   // parameterless constructor
   public Withdrawal()
   {
      // constructor body code
   }
}
```

Fig. 26.3 | Initial C# code for class `Withdrawal` based on Figs. 26.1 and 26.2.

2. Use the attributes located in the class's second compartment to declare the instance variables. The `private` attributes `accountNumber` and `amount` of class `Withdrawal` yield the code in Fig. 26.4.

```
// Fig. 26.4: Withdrawal.cs
// Class Withdrawal represents an ATM withdrawal transaction
public class Withdrawal
{
   // attributes
   private int accountNumber; // account to withdraw funds from
   private decimal amount; // amount to withdraw from account

   // parameterless constructor
   public Withdrawal()
   {
      // constructor body code
   }
}
```

Fig. 26.4 | Incorporating `private` variables for class `Withdrawal` based on Figs. 26.1 and 26.2.

3. Use the associations described in the class diagram to declare references to other objects. According to Fig. 26.2, `Withdrawal` can access one object of class `Screen`, one object of class `Keypad`, one object of class `CashDispenser` and one object of class `BankDatabase`. Class `Withdrawal` must maintain references to these objects to send messages to them, so lines 10–13 of Fig. 26.5 declare the appropriate references as `private` instance variables. In the implementation of class `Withdrawal` in Section 26.4.10, a constructor initializes these instance variables with references to the actual objects.

4. Use the operations located in the third compartment of Fig. 26.1 to declare the shells of the methods. If we have not yet specified a return type for an operation, we declare the method with return type `void`. Refer to the class diagrams of Figs. 25.18–25.21 to declare any necessary parameters. Adding the `public` oper-

ation `Execute` (which has an empty parameter list) in class `Withdrawal` yields the code in lines 23–26 of Fig. 26.6. [*Note:* We code the bodies of the methods when we implement the complete ATM system.]

Software Engineering Observation 26.1

Many UML modeling tools can convert UML-based designs into C# code, considerably speeding up the implementation process.

```
// Fig. 26.5: Withdrawal.cs
// Class Withdrawal represents an ATM withdrawal transaction
public class Withdrawal
{
   // attributes
   private int accountNumber; // account to withdraw funds from
   private decimal 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-information database

   // parameterless constructor
   public Withdrawal()
   {
      // constructor body code
   }
}
```

Fig. 26.5 | Incorporating `private` reference handles for the associations of class `Withdrawal` based on Figs. 26.1 and 26.2.

```
1  // Fig. 26.6: Withdrawal.cs
2  // Class Withdrawal represents an ATM withdrawal transaction
3  public class Withdrawal
4  {
5     // attributes
6     private int accountNumber; // account to withdraw funds from
7     private decimal amount; // amount to withdraw
8
9     // references to associated objects
10    private Screen screen; // ATM's screen
11    private Keypad keypad; // ATM's keypad
12    private CashDispenser cashDispenser; // ATM's cash dispenser
13    private BankDatabase bankDatabase; // account-information database
14
15    // parameterless constructor
16    public Withdrawal()
17    {
```

Fig. 26.6 | C# code incorporating method `Execute` in class `Withdrawal` based on Figs. 26.1 and 26.2. (Part 1 of 2.)

```
        // constructor body code
    }

    // operations
    // perform transaction
    public void Execute()
    {
        // Execute method body code
    }
}
```

Fig. 26.6 | C# code incorporating method `Execute` in class `Withdrawal` based on Figs. 26.1 and 26.2. (Part 2 of 2.)

This concludes our discussion of the basics of generating class files from UML diagrams. In the next section, we demonstrate how to modify the code in Fig. 26.6 to incorporate the object-oriented concepts of inheritance and polymorphism, which we presented in Chapters 11 and 12, respectively.

Self-Review Exercises

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

26.2 In Fig. 26.2, the association between the `ATM` and the `Screen` indicates:
a) that we can navigate from the `Screen` to the `ATM`.
b) that we can navigate from the `ATM` to the `Screen`.
c) Both a and b; the association is bidirectional.
d) None of the above.

26.3 Write C# code to begin implementing the design for class `Account`.

26.3 Incorporating Inheritance and Polymorphism into the ATM System

We now revisit our ATM system design to see how it might benefit from inheritance and polymorphism. To apply inheritance, we first look for commonality among classes in the system. We create an inheritance hierarchy to model similar classes in an elegant and efficient manner that enables us to process objects of these classes polymorphically. We then modify our class diagram to incorporate the new inheritance relationships. Finally, we demonstrate how the inheritance aspects of our updated design are translated into C# code.

In Section 25.3, we encountered the problem of representing a financial transaction in the system. Rather than create one class to represent all transaction types, we created three distinct transaction classes—`BalanceInquiry`, `Withdrawal` and `Deposit`—to represent the transactions that the ATM system can perform. The class diagram of Fig. 26.7 shows the attributes and operations of these classes. They have one private attribute (`accountNumber`) and one public operation (`Execute`) in common. Each class requires attribute `accountNumber` to specify the account to which the transaction applies. Each class contains operation `Execute`, which the `ATM` invokes to perform the transaction. Clearly, `BalanceInquiry`, `Withdrawal` and `Deposit` represent *types of* transactions.

Figure 26.7 reveals commonality among the transaction classes, so using inheritance to factor out the common features seems appropriate for designing these classes. We place the common functionality in base class `Transaction` and derive classes `BalanceInquiry`, `Withdrawal` and `Deposit` from `Transaction` (Fig. 26.8).

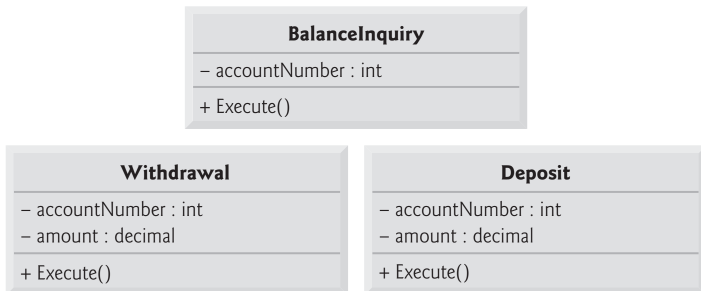

Fig. 26.7 | Attributes and operations of classes `BalanceInquiry`, `Withdrawal` and `Deposit`.

The UML specifies a relationship called a **generalization** to model inheritance. Figure 26.8 is the class diagram that models the inheritance relationship between base class `Transaction` and its three derived classes. The arrows with triangular hollow arrowheads indicate that classes `BalanceInquiry`, `Withdrawal` and `Deposit` are derived from class `Transaction` by inheritance. Class `Transaction` is said to be a generalization of its derived classes. The derived classes are said to be **specializations** of class `Transaction`.

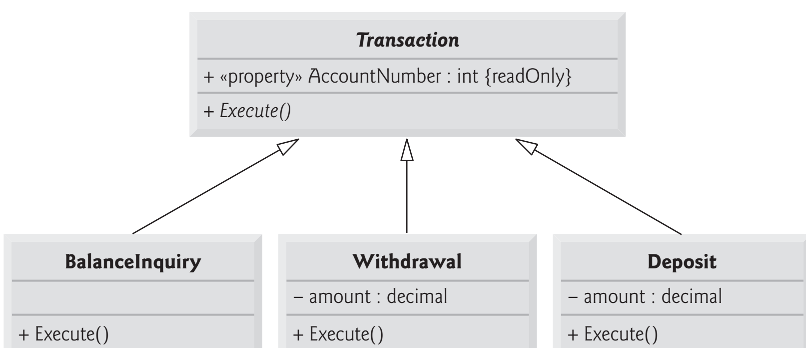

Fig. 26.8 | Class diagram modeling the generalization (i.e., inheritance) relationship between the base class `Transaction` and its derived classes `BalanceInquiry`, `Withdrawal` and `Deposit`.

As Fig. 26.7 shows, classes `BalanceInquiry`, `Withdrawal` and `Deposit` share private `int` attribute `accountNumber`. We'd like to factor out this common attribute and place it in the base class `Transaction`. However, recall that a base class's private attributes are not accessible in derived classes. The derived classes of `Transaction` require access to attribute `accountNumber` so that they can specify which `Account` to process in the `BankDatabase`. A derived class can access the public and protected members of its base class. However, the derived classes in this case do not need to modify attribute accountNumber—they need only to access its value. For this reason, we have chosen to replace private attribute accountNumber in our model with the public read-only property AccountNumber. Since this is a read-only property, it provides only a get accessor to access the account number. Each derived class inherits this property, enabling the derived class to access its account number as needed to execute a transaction. We no longer list accountNumber in the second compartment of each derived class, because the three derived classes inherit property AccountNumber from Transaction.

According to Fig. 26.7, classes BalanceInquiry, Withdrawal and Deposit also share operation Execute, so base class Transaction should contain public operation Execute. However, it does not make sense to implement Execute in class Transaction, because the functionality that this operation provides depends on the specific type of the actual transaction. We therefore declare Execute as an **abstract operation** in base class Transaction—it will become an abstract method in the C# implementation. This makes Transaction an abstract class and forces any class derived from Transaction that must be a concrete class (i.e., BalanceInquiry, Withdrawal and Deposit) to implement the operation Execute to make the derived class concrete. The UML requires that we place abstract class names and abstract operations in italics. Thus, in Fig. 26.8, Transaction and Execute appear in italics for the Transaction class; Execute is not italicized in derived classes BalanceInquiry, Withdrawal and Deposit. Each derived class overrides base class Transaction's Execute operation with an appropriate concrete implementation. Fig. 26.8 includes operation Execute in the third compartment of classes BalanceInquiry, Withdrawal and Deposit, because each class has a different concrete implementation of the overridden operation.

A derived class can inherit interface and implementation from a base class. Compared to a hierarchy designed for implementation inheritance, one designed for interface inheritance tends to have its functionality lower in the hierarchy—a base class signifies one or more operations that should be defined by each class in the hierarchy, but the individual derived classes provide their own implementations of the operation(s). The inheritance hierarchy designed for the ATM system takes advantage of this type of inheritance, which provides the ATM with an elegant way to execute all transactions "in the general" (i.e., polymorphically). Each class derived from Transaction inherits some implementation details (e.g., property AccountNumber), but the primary benefit of incorporating inheritance into our system is that the derived classes share a common interface (e.g., abstract operation Execute). The ATM can aim a Transaction reference at any transaction, and when the ATM invokes the operation Execute through this reference, the version of Execute specific to that transaction runs (polymorphically) automatically (due to polymorphism). For example, suppose a user chooses to perform a balance inquiry. The ATM aims a Transaction reference at a new object of class BalanceInquiry, which the C# compiler allows because a BalanceInquiry *is a* Transaction. When the ATM uses this reference to invoke Execute, BalanceInquiry's version of Execute is called (polymorphically).

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

An abstract class like Transaction is one for which the programmer never intends to (and, in fact, cannot) instantiate objects. An abstract class simply declares common attributes and behaviors for its derived classes in an inheritance hierarchy. Class Transaction defines the concept of what it means to be a transaction that has an account number and can be executed. You may wonder why we bother to include abstract operation Execute in class Transaction if Execute lacks a concrete implementation. Conceptually, we include this operation because it is the defining behavior of all transactions—executing. Technically, we must include operation Execute in base class Transaction so that the ATM (or any other class) can invoke each derived class's overridden version of this operation polymorphically via a Transaction reference.

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

Figure 26.9 presents an updated class diagram of our model that incorporates inheritance and introduces abstract base class Transaction. We model an association between class ATM and class Transaction to show that the ATM, at any given moment, either is executing a transaction or is not (i.e., zero or one objects of type Transaction exist in the system at a time). Because a Withdrawal is a type of Transaction, we no longer draw an association line directly between class ATM and class Withdrawal—derived class Withdrawal inherits base class Transaction's association with class ATM. Derived classes BalanceInquiry and Deposit also inherit this association, which replaces the previously omitted associations between classes BalanceInquiry and Deposit, and class ATM. Note again the use of triangular hollow arrowheads to indicate the specializations (i.e., derived classes) of class Transaction, as indicated in Fig. 26.8.

We also add an association between Transaction and BankDatabase (Fig. 26.9). All Transactions require a reference to the BankDatabase so that they can access and modify account information. Each Transaction derived class inherits this reference, so we no longer model the association between Withdrawal and BankDatabase. The association between class Transaction and the BankDatabase replaces the previously omitted associations between classes BalanceInquiry and Deposit, and the BankDatabase.

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

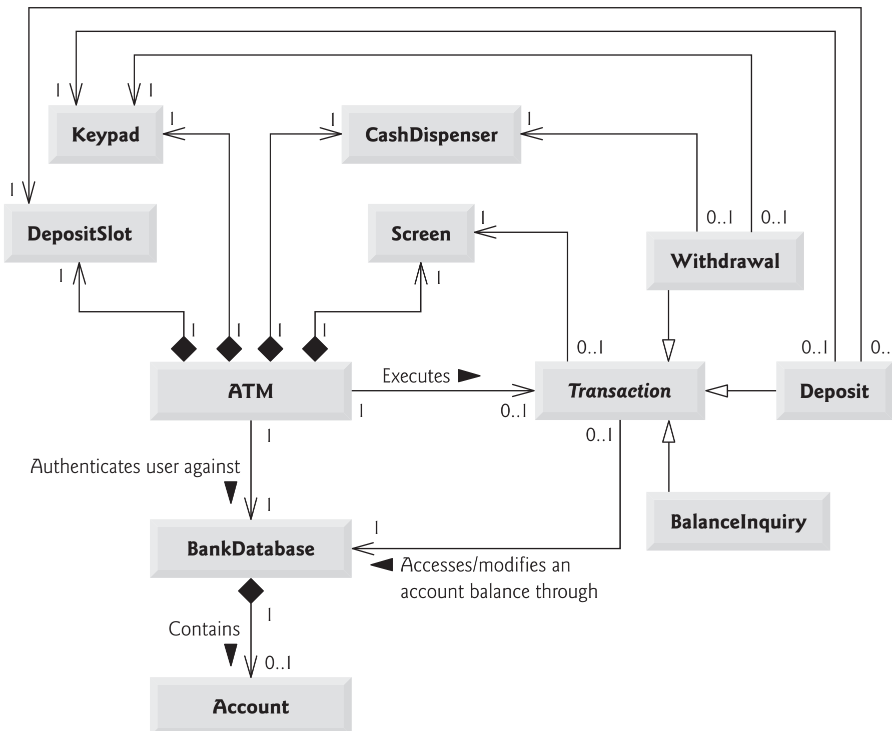

Fig. 26.9 | Class diagram of the ATM system (incorporating inheritance). Abstract class name `Transaction` appears in italics.

Our class diagram incorporating inheritance (Fig. 26.9) also models classes `Deposit` and `BalanceInquiry`. We show associations between `Deposit` and both the `DepositSlot` and the `Keypad`. Class `BalanceInquiry` takes part in only those associations inherited from class `Transaction`—a `BalanceInquiry` interacts only with the `BankDatabase` and the `Screen`.

The modified class diagram in Fig. 26.10 includes abstract base class `Transaction`. This abbreviated diagram does not show inheritance relationships (these appear in Fig. 26.9), but instead shows the attributes and operations after we have employed inheritance in our system. Abstract class name `Transaction` and abstract operation name `Execute` in class `Transaction` appear in italics. To save space, we do not include those attributes shown by associations in Fig. 26.9—we do, however, include them in the C# implementation. We also omit all operation parameters—incorporating inheritance does not affect the parameters already modeled in Figs. 25.18–25.21.

Software Engineering Observation 26.2

A complete class diagram shows all the associations among classes, and all the attributes and operations for each class. When the number of class attributes, operations and associations is substantial (as in Figs. 26.9 and 26.10), a good practice that promotes readability is to divide this information between two class diagrams—one focusing on associations and the other on attributes and operations.

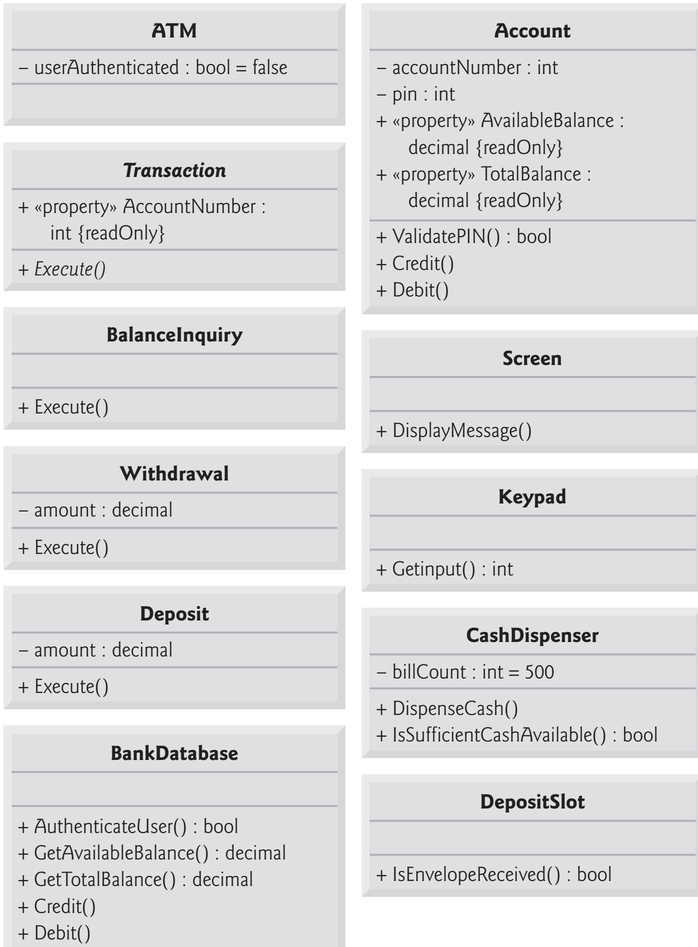

Fig. 26.10 | Class diagram after incorporating inheritance into the system.

Implementing the ATM System Design Incorporating Inheritance

In the previous section, we began implementing the ATM system design in C#. We now incorporate inheritance, using class `Withdrawal` as an example.

1. If a class `A` is a generalization of class `B`, then class `B` is derived from (and is a specialization of) class `A`. For example, abstract base class `Transaction` is a generalization of class `Withdrawal`. Thus, class `Withdrawal` is derived from (and is a specialization of) class `Transaction`. Figure 26.11contains the shell of class `Withdrawal`, in which the class definition indicates the inheritance relationship between `Withdrawal` and `Transaction` (line 3).

2. If class `A` is an abstract class and class `B` is derived from class `A`, then class `B` must implement the abstract operations of class `A` if class `B` is to be a concrete class. For example, class `Transaction` contains abstract operation `Execute`, so class

```
// Fig. 26.11: Withdrawal.cs
// Class Withdrawal represents an ATM withdrawal transaction.
public class Withdrawal : Transaction
{
   // code for members of class Withdrawal
}
```

Fig. 26.11 | C# code for shell of class `Withdrawal`.

`Withdrawal` must implement this operation if we want to instantiate `Withdrawal` objects. Figure 26.12 contains the portions of the C# code for class `Withdrawal` that can be inferred from Figs. 26.9 and 26.10. Class `Withdrawal` inherits property `AccountNumber` from base class `Transaction`, so `Withdrawal` does not declare this property. Class `Withdrawal` also inherits references to the `Screen` and the `BankDatabase` from class `Transaction`, so we do not include these references. Figure 26.10 specifies attribute `amount` and operation `Execute` for class `Withdrawal`. Line 6 of Fig. 26.12 declares an instance variable for attribute `amount`. Lines 17–20 declare the shell of a method for operation `Execute`. Recall that derived class `Withdrawal` must provide a concrete implementation of the `abstract` method `Execute` from base class `Transaction`. The `keypad` and `cash-Dispenser` references (lines 7–8) are instance variables whose need is apparent from class `Withdrawal`'s associations in Fig. 26.9—in this class's C# implementation (Section 26.4.10), a constructor initializes these references to actual objects.

We discuss the polymorphic processing of `Transactions` in Section 26.4.1 of the ATM implementation. Class ATM performs the actual polymorphic call to method `Execute` at line 99 of Fig. 26.26.

```
1  // Fig. 26.12: Withdrawal.cs
2  // Class Withdrawal represents an ATM withdrawal transaction.
3  public class Withdrawal : Transaction
4  {
5     // attributes
6     private decimal amount; // amount to withdraw
7     private Keypad keypad; // reference to keypad
8     private CashDispenser cashDispenser; // reference to cash dispenser
9
10    // parameterless constructor
11    public Withdrawal()
12    {
13       // constructor body code
14    }
15
16    // method that overrides Execute
17    public override void Execute()
18    {
19       // Execute method body code
20    }
21 }
```

Fig. 26.12 | C# code for class `Withdrawal` based on Figs. 26.9 and 26.10.

Self-Review Exercises

26.4 The UML uses an arrow with a __________ to indicate a generalization relationship.

a) solid filled arrowhead
b) triangular hollow arrowhead
c) diamond-shaped hollow arrowhead
d) stick arrowhead

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

26.6 Write C# code to begin implementing the design for class `Transaction` specified in Figures 26.9 and 26.10. Be sure to include `private` references based on class `Transaction`'s associations. Also, be sure to include properties with `public get` accessors for any of the `private` instance variables that the derived classes must access to perform their tasks.

26.4 ATM Case Study Implementation

This section contains the ATM system's complete working implementation. We consider the 11 classes in the order in which we identified them in Section 25.3 (with the exception of `Transaction`, which was introduced in Section 26.3 as the base class of classes `BalanceInquiry`, `Withdrawal` and `Deposit`):

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

We apply the guidelines discussed in Sections 26.2–26.3 to code these classes based on how we modeled them in the UML class diagrams of Figs. 26.9–26.10. To develop the bodies of class methods, we refer to the activity diagrams presented in Section 25.5 and the communication and sequence diagrams presented in Section 25.6. 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 25.2.

We conclude the discussion by presenting a test harness (`ATMCaseStudy` in Section 26.4.12) that creates an object of class `ATM` and starts it by calling its `Run` method. Recall that we are developing a first version of the ATM system that runs on a personal computer and uses the keyboard and monitor to approximate the ATM's keypad and screen. Also, we simulate the actions of the ATM's cash dispenser and deposit slot. We attempt to implement the system so that real hardware versions of these devices could be integrated without significant code changes. [*Note:* For the purpose of this simulation, we have provided two predefined accounts in class `BankDatabase`. The first account has the account number 12345 and the PIN 54321. The second account has the account number 98765 and the PIN 56789. You should use these accounts when testing the ATM.]

26.4.1 Class ATM

Class `ATM` (Fig. 26.13) represents the ATM as a whole. Lines 5–11 implement the class's attributes. We determine all but one of these attributes from the UML class diagrams of Figs. 26.9–26.10. Line 5 declares the `bool` attribute `userAuthenticated` from Fig. 26.10. Line 6 declares an attribute not found in our UML design—`int` attribute `currentAccountNumber`, which keeps track of the account number of the current authenticated user. Lines 7–11 declare reference-type instance variables corresponding to the `ATM` class's associations modeled in the class diagram of Fig. 26.9. 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).

Lines 14–20 declare an enumeration that corresponds to the four options in the ATM's main menu (i.e., balance inquiry, withdrawal, deposit and exit). Lines 23–32 declare class `ATM`'s constructor, which initializes the class's attributes. When an `ATM` object is first created, no user is authenticated, so line 25 initializes `userAuthenticated` to `false`. Line 26 initializes `currentAccountNumber` to `0` because there is no current user yet. Lines 27–30 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 31 creates a new `BankDatabase`. As you'll soon see, the `BankDatabase` creates two `Account` objects that can be used to test the ATM. [*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.]

```
1  // ATM.cs
2  // Represents an automated teller machine.
3  public class ATM
4  {
5     private bool userAuthenticated; // true if user is authenticated
6     private int currentAccountNumber; // user's account number
7     private Screen screen; // reference to ATM's screen
8     private Keypad keypad; // reference to ATM's keypad
9     private CashDispenser cashDispenser; // ref to ATM's cash dispenser
10    private DepositSlot depositSlot; // reference to ATM's deposit slot
11    private BankDatabase bankDatabase; // ref to account info database
12
13    // enumeration that represents main menu options
14    private enum MenuOption
15    {
```

Fig. 26.13 | Class `ATM` represents the ATM. (Part 1 of 4.)

```
      BALANCE_INQUIRY = 1,
      WITHDRAWAL = 2,
      DEPOSIT = 3,
      EXIT_ATM = 4
   }

   // parameterless constructor initializes instance variables
   public ATM()
   {
      userAuthenticated = false; // user is not authenticated to start
      currentAccountNumber = 0; // no current account number to start
      screen = new Screen(); // create screen
      keypad = new Keypad(); // create keypad
      cashDispenser = new CashDispenser(); // create cash dispenser
      depositSlot = new DepositSlot(); // create deposit slot
      bankDatabase = new BankDatabase(); // create account info database
   }

   // start ATM
   public void Run()
   {
      // welcome and authenticate users; perform transactions
      while (true) // infinite loop
      {
         // loop while user is not yet authenticated
         while (!userAuthenticated)
         {
            screen.DisplayMessageLine("\nWelcome!");
            AuthenticateUser(); // authenticate user
         }

         PerformTransactions(); // for authenticated user
         userAuthenticated = false; // reset before next ATM session
         currentAccountNumber = 0; // reset before next ATM session
         screen.DisplayMessageLine("\nThank you! Goodbye!");
      }
   }

   // attempt to authenticate user against database
   private void AuthenticateUser()
   {
      // prompt for account number and input it from user
      screen.DisplayMessage("\nPlease enter your account number: ");
      int accountNumber = keypad.GetInput();

      // prompt for PIN and input it from user
      screen.DisplayMessage("\nEnter your PIN: ");
      int pin = keypad.GetInput();

      // set userAuthenticated to boolean value returned by database
      userAuthenticated =
         bankDatabase.AuthenticateUser(accountNumber, pin);

```

Fig. 26.13 | Class ATM represents the ATM. (Part 2 of 4.)

```
      // check whether authentication succeeded
      if  (userAuthenticated)
         currentAccountNumber = accountNumber; // save user's account #
      else
         screen.DisplayMessageLine(
            "Invalid account number or PIN. Please try again.");
   }

   // display the main menu and perform transactions
   private void PerformTransactions()
   {
      Transaction currentTransaction; // transaction being processed
      bool userExited = false; // user has not chosen to exit

      // loop while user has not chosen exit option
      while (!userExited)
      {
         // show main menu and get user selection
         int mainMenuSelection = DisplayMainMenu();

         // decide how to proceed based on user's menu selection
         switch ((MenuOption) mainMenuSelection)
         {
            // user chooses to perform one of three transaction types
            case MenuOption.BALANCE_INQUIRY:
            case MenuOption.WITHDRAWAL:
            case MenuOption.DEPOSIT:
               // initialize as new object of chosen type
               currentTransaction =
                  CreateTransaction(mainMenuSelection);
               currentTransaction.Execute(); // execute transaction
               break;
            case MenuOption.EXIT_ATM: // user chose to terminate session
               screen.DisplayMessageLine("\nExiting the system...");
               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.");
               break;
         }
      }
   }

   // display the main menu and return an input selection
   private int DisplayMainMenu()
   {
      screen.DisplayMessageLine("\nMain menu:");
      screen.DisplayMessageLine("1 - View my balance");
      screen.DisplayMessageLine("2 - Withdraw cash");
      screen.DisplayMessageLine("3 - Deposit funds");
      screen.DisplayMessageLine("4 - Exit\n");
      screen.DisplayMessage("Enter a choice: ");
```

Fig. 26.13 | Class ATM represents the ATM. (Part 3 of 4.)

```
122          return keypad.GetInput(); // return user's selection
123       }
124
125       // return object of specified Transaction derived class
126       private Transaction CreateTransaction(int type)
127       {
128          Transaction temp = null; // null Transaction reference
129
130          // determine which type of Transaction to create
131          switch ((MenuOption) type)
132          {
133             // create new BalanceInquiry transaction
134             case MenuOption.BALANCE_INQUIRY:
135                temp = new BalanceInquiry(currentAccountNumber,
136                   screen, bankDatabase);
137                break;
138             case MenuOption.WITHDRAWAL: // create new Withdrawal transaction
139                temp = new Withdrawal(currentAccountNumber, screen,
140                   bankDatabase, keypad, cashDispenser);
141                break;
142             case MenuOption.DEPOSIT: // create new Deposit transaction
143                temp = new Deposit(currentAccountNumber, screen,
144                   bankDatabase, keypad, depositSlot);
145                break;
146          }
147
148          return temp;
149       }
150    }
```

Fig. 26.13 | Class `ATM` represents the ATM. (Part 4 of 4.)

Implementing the Operation

The class diagram of Fig. 26.10 does not list any operations for class `ATM`. We now implement one operation (i.e., `public` method) in class `ATM` that allows an external client of the class (i.e., class `ATMCaseStudy`; Section 26.4.12) to tell the `ATM` to run. `ATM` method `Run` (Fig. 26.13, lines 35–52) uses an infinite loop (lines 38–51) 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 exits, the ATM resets itself, displays a goodbye message and restarts the process for the next user. 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 can exit the system, but cannot turn off the ATM completely.

Inside method `Run`'s infinite loop, lines 41–45 cause the ATM to repeatedly welcome and attempt to authenticate the user as long as the user has not been authenticated (i.e., the condition `!userAuthenticated` is `true`). Line 43 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 14–17 of Fig. 26.14) displays a message to the user, but this method also outputs a newline after displaying the message. We add this method during implementation to give class `Screen`'s clients more control over the placement of displayed messages. Line 44

(Fig. 26.13) invokes class ATM's private utility method AuthenticateUser (declared in lines 55–75) to attempt to authenticate the user.

Authenticating the User

We refer to the requirements document to determine the steps necessary to authenticate the user before allowing transactions to occur. Line 58 of method AuthenticateUser invokes method DisplayMessage of the ATM's screen to prompt the user to enter an account number. Line 59 invokes method GetInput of the ATM's keypad to obtain the user's input, then stores this integer in local variable accountNumber. Method AuthenticateUser next prompts the user to enter a PIN (line 62), and stores the PIN in local variable pin (line 63). Next, lines 66–67 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 bool value returned by this method—userAuthenticated becomes true if authentication succeeds (i.e., the accountNumber and pin match those of an existing Account in bankDatabase) and remains false otherwise. If userAuthenticated is true, line 71 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 73–74 call the screen's DisplayMessageLine method to indicate that an invalid account number and/or PIN was entered, so the user must try again. We set currentAccountNumber only after authenticating the user's account number and the associated PIN—if the database cannot authenticate the user, currentAccountNumber remains 0.

After method Run attempts to authenticate the user (line 44), if userAuthenticated is still false (line 41), the while loop body (lines 41–45) executes again. If userAuthenticated is now true, the loop terminates, and control continues with line 47, which calls class ATM's private utility method PerformTransactions.

Performing Transactions

Method PerformTransactions (lines 78–111) carries out an ATM session for an authenticated user. Line 80 declares local variable Transaction, to which we assign a BalanceInquiry, Withdrawal or Deposit object representing the ATM transaction currently being processed. We use a Transaction variable here to allow us to take advantage of polymorphism. Also, we name this variable after the role name included in the class diagram of Fig. 25.7—currentTransaction. Line 81 declares another local variable—a bool called userExited that keeps track of whether the user has chosen to exit. This variable controls a while loop (lines 84–110) that allows the user to execute an unlimited number of transactions before choosing to exit. Within this loop, line 87 displays the main menu and obtains the user's menu selection by calling ATM utility method DisplayMainMenu (declared in lines 114–123). 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 87 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 90–109) to respond to the selection appropriately. If mainMenuSelection is equal to the underlying value of any of the three enum members representing transaction types (i.e., if the user chose to perform a transaction), lines 97–98 call utility method

CreateTransaction (declared in lines 126–149) to return a newly instantiated object of the type that corresponds to the selected transaction. Variable currentTransaction is assigned the reference returned by method CreateTransaction, then line 99 invokes method Execute of this transaction to execute it. We discuss Transaction method Execute and the three Transaction derived classes shortly. We assign to the Transaction variable currentTransaction an object of one of the three Transaction derived classes so that we can execute transactions. For example, if the user chooses to perform a balance inquiry, (MenuOption) mainMenuSelection (line 90) matches the case label MenuOption.BALANCE_INQUIRY, and CreateTransaction returns a BalanceInquiry object (lines 97–98). Thus, currentTransaction refers to a BalanceInquiry and invoking currentTransaction.Execute() (line 99) results in BalanceInquiry's version of Execute being called polymorphically.

Creating Transactions

Method CreateTransaction (lines 126–149) uses a switch statement (lines 131–146) to instantiate a new Transaction derived class object of the type indicated by the parameter type. Recall that method PerformTransactions passes mainMenuSelection to method CreateTransaction only when mainMenuSelection contains a value corresponding to one of the three transaction types. So parameter type (line 126) receives one of the values MenuOption.BALANCE_INQUIRY, MenuOption.WITHDRAWAL or MenuOption.DEPOSIT. Each case in the switch statement instantiates a new object by calling the appropriate Transaction derived class constructor. Each constructor has a unique parameter list, based on the specific data required to initialize the derived class object. A BalanceInquiry (lines 135–136) 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 (lines 139–140) requires references to the ATM's keypad and cashDispenser, and a Deposit (lines 143–144) requires references to the ATM's keypad and depositSlot. We discuss the transaction classes in detail in Sections 26.4.8–26.4.11.

After executing a transaction (line 99 in method PerformTransactions), userExited remains false, and the while loop in lines 84–110 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 103 sets userExited to true, causing the condition in line 84 of the while loop (!userExited) to become false. This while is the final statement of method PerformTransactions, so control returns to line 47 of the calling method Run. If the user enters an invalid main menu selection (i.e., not an integer in the range 1–4), lines 106–107 display an appropriate error message, userExited remains false (as set in line 81) and the user returns to the main menu to try again.

When method PerformTransactions returns control to method Run, the user has chosen to exit the system, so lines 48–49 reset the ATM's attributes userAuthenticated and currentAccountNumber to false and 0, respectively, to prepare for the next ATM user. Line 50 displays a goodbye message to the current user before the ATM welcomes the next user.

26.4.2 Class Screen

Class Screen (Fig. 26.14) represents the screen of the ATM and encapsulates all aspects of displaying output to the user. Class Screen simulates a real ATM's screen with the computer monitor and outputs text messages using standard console output methods `Console.Write` and `Console.WriteLine`. In the design portion of this case study, we endowed class `Screen` with one operation—`DisplayMessage`. For greater flexibility in displaying messages to the `Screen`, we now declare three `Screen` methods—`DisplayMessage`, `DisplayMessageLine` and `DisplayDollarAmount`.

```
1  // Screen.cs
2  // Represents the screen of the ATM
3  using System;
4
5  public class Screen
6  {
7     // displays a message without a terminating carriage return
8     public void DisplayMessage(string message)
9     {
10       Console.Write(message);
11    }
12
13    // display a message with a terminating carriage return
14    public void DisplayMessageLine(string message)
15    {
16       Console.WriteLine(message);
17    }
18
19    // display a dollar amount
20    public void DisplayDollarAmount(decimal amount)
21    {
22       Console.Write("{0:C}", amount);
23    }
24 }
```

Fig. 26.14 | Class `Screen` represents the screen of the ATM.

Method `DisplayMessage` (lines 8–11) takes a `string` as an argument and prints it to the screen using `Console.Write`. The cursor stays on the same line, making this method appropriate for displaying prompts to the user. Method `DisplayMessageLine` (lines 14–17) does the same using `Console.WriteLine`, which outputs a newline to move the cursor to the next line. Finally, method `DisplayDollarAmount` (lines 20–23) outputs a properly formatted dollar amount (e.g., `$1,234.56`). Line 22 uses method `Console.Write` to output a `decimal` value formatted as currency with a dollar sign, two decimal places and commas to increase the readability of large dollar amounts.

26.4.3 Class Keypad

Class `Keypad` (Fig. 26.15) 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 method `Console.ReadLine` to obtain keyboard input from the user. A computer keyboard contains many keys not found on the ATM's keypad. 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.cs
2 // Represents the keypad of the ATM.
3 using System;
4
5 public class Keypad
6 {
7    // return an integer value entered by user
8    public int GetInput()
9    {
10      return int.Parse(Console.ReadLine());
11   }
12 }
```

Fig. 26.15 | Class `Keypad` represents the ATM's keypad.

Method `GetInput` (lines 8–11) invokes converts the input returned by `Console.ReadLine` (line 10) to an `int` value. [*Note:* Method `ToInt32` can throw a `FormatException` if the user enters non-integer input. Because the real ATM's keypad permits only integer input, we simply assume that no exceptions will occur. See Chapter 13 for information on catching and processing exceptions.] Recall that `ReadLine` obtains all the input used by the ATM. Class `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.

26.4.4 Class CashDispenser

Class `CashDispenser` (Fig. 26.16) represents the cash dispenser of the ATM. Line 6 declares constant `INITIAL_COUNT`, which indicates the number of $20 bills in the cash dispenser when the ATM starts (i.e., `500`). Line 7 implements attribute `billCount` (modeled in Fig. 26.10), which keeps track of the number of bills remaining in the `CashDispenser` at any time. The constructor (lines 10–13) sets `billCount` to the initial count. [*Note:* We assume that the process of adding more bills to the `CashDispenser` and updating the `billCount` occur outside the ATM system.] Class `CashDispenser` has two `public` methods—`DispenseCash` (lines 16–21) and `IsSufficientCashAvailable` (lines 24–31). The class trusts that a client (i.e., `Withdrawal`) calls method `DispenseCash` only after establishing that sufficient cash is available by calling method `IsSufficientCashAvailable`. Thus, `DispenseCash` simulates dispensing the requested amount of cash without checking whether sufficient cash is available.

```
1 // CashDispenser.cs
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 const int INITIAL_COUNT = 500;
7    private int billCount; // number of $20 bills remaining
```

Fig. 26.16 | Class `CashDispenser` represents the ATM's cash dispenser. (Part 1 of 2.)

```
8
9      // parameterless constructor initializes billCount to INITIAL_COUNT
10     public CashDispenser()
11     {
12        billCount = INITIAL_COUNT; // set billCount to INITIAL_COUNT
13     }
14
15     // simulates dispensing the specified amount of cash
16     public void DispenseCash(decimal amount)
17     {
18        // number of $20 bills required
19        int billsRequired = ((int) amount) / 20;
20        billCount -= billsRequired;
21     }
22
23     // indicates whether cash dispenser can dispense desired amount
24     public bool IsSufficientCashAvailable(decimal amount)
25     {
26        // number of $20 bills required
27        int billsRequired = ((int) amount) / 20;
28
29        // return whether there are enough bills available
30        return (billCount >= billsRequired);
31     }
32  }
```

Fig. 26.16 | Class `CashDispenser` represents the ATM's cash dispenser. (Part 2 of 2.)

Method `IsSufficientCashAvailable` (lines 24–31) has a parameter `amount` that specifies the amount of cash in question. Line 27 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 convert `amount` to an integer value and divide it by `20` to obtain the number of `billsRequired`. Line 30 returns `true` if the `CashDispenser`'s `billCount` is greater than or equal to `billsRequired` (i.e., enough bills are available) and `false` otherwise (i.e., not enough bills). For example, if a user wishes to withdraw $80 (i.e., `billsRequired` is `4`), but only three bills remain (i.e., `billCount` is `3`), the method returns `false`.

Method `DispenseCash` (lines 16–21) 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 the cash. Our simulated version of the method simply decreases the `billCount` of bills remaining by the number required to dispense the specified `amount` (line 20). It is the responsibility of the client of the class (i.e., `Withdrawal`) to inform the user that cash has been dispensed—`CashDispenser` does not interact directly with `Screen`.

26.4.5 Class `DepositSlot`

Class `DepositSlot` (Fig. 26.17) represents the deposit slot of the ATM. This class simulates the functionality of a real hardware deposit slot. `DepositSlot` has no attributes and only one method—`IsDepositEnvelopeReceived` (lines 7–10)—which indicates whether a deposit envelope was received.

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

Fig. 26.17 | 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 `IsDepositEnvelopeReceived` simply returns `true` immediately (line 9), because this is only a software simulation, so we assume that the user inserts an envelope within the required time frame. If an actual hardware deposit slot were connected to our system, method `IsDepositEnvelopeReceived` would 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 `IsDepositEnvelopeReceived` were to receive such a signal within two minutes, the method would return `true`. If two minutes were to elapse and the method still had not received a signal, then the method would return `false`.

26.4.6 Class Account

Class `Account` (Fig. 26.18) represents a bank account. Each `Account` has four attributes (modeled in Fig. 26.10)—`accountNumber`, `pin`, `availableBalance` and `totalBalance`. Lines 5–8 implement these attributes as `private` instance variables. For each of the instance variables `accountNumber`, `availableBalance` and `totalBalance`, we provide a property with the same name as the attribute, but starting with a capital letter. For example, property `AccountNumber` corresponds to the `accountNumber` attribute modeled in Fig. 26.10. Clients of this class do not need to modify the `accountNumber` instance variable, so `AccountNumber` is declared as a read-only property (i.e., it provides only a `get` accessor).

```
1  // Account.cs
2  // Class Account represents a bank account.
3  public class Account
4  {
5     private int accountNumber; // account number
6     private int pin; // PIN for authentication
7     private decimal availableBalance; // available withdrawal amount
8     private decimal totalBalance; // funds available + pending deposit
9
10    // four-parameter constructor initializes attributes
11    public Account(int theAccountNumber, int thePIN,
12       decimal theAvailableBalance, decimal theTotalBalance)
13    {
```

Fig. 26.18 | Class `Account` represents a bank account. (Part 1 of 2.)

```
         accountNumber = theAccountNumber;
         pin = thePIN;
         availableBalance = theAvailableBalance;
         totalBalance = theTotalBalance;
      }

      // read-only property that gets the account number
      public int AccountNumber
      {
         get
         {
            return accountNumber;
         }
      }

      // read-only property that gets the available balance
      public decimal AvailableBalance
      {
         get
         {
            return availableBalance;
         }
      }

      // read-only property that gets the total balance
      public decimal TotalBalance
      {
         get
         {
            return totalBalance;
         }
      }

      // determines whether a user-specified PIN matches PIN in Account
      public bool ValidatePIN(int userPIN)
      {
         return (userPIN == pin);
      }

      // credits the account (funds have not yet cleared)
      public void Credit(decimal amount)
      {
         totalBalance += amount; // add to total balance
      }

      // debits the account
      public void Debit(decimal amount)
      {
         availableBalance -= amount; // subtract from available balance
         totalBalance -= amount; // subtract from total balance
      }
   }
```

Fig. 26.18 | Class `Account` represents a bank account. (Part 2 of 2.)

Class `Account` has a constructor (lines 11–18) that takes an account number, the PIN established for the account, the initial available balance and the initial total balance as arguments. Lines 14–17 assign these values to the class's attributes (i.e., instance variables). `Account` objects would normally be created externally to the ATM system. However, in this simulation, the `Account` objects are created in the `BankDatabase` class (Fig. 26.19).

`public` Read-Only Properties of Class `Account`

Read-only property `AccountNumber` (lines 21–27) provides access to an `Account`'s `accountNumber` instance variable. We include this property in our implementation so that a client of the class (e.g., `BankDatabase`) can identify a particular `Account`. For example, `BankDatabase` contains many `Account` objects, and it can access this property on each of its `Account` objects to locate the one with a specific account number.

Read-only properties `AvailableBalance` (lines 30–36) and `TotalBalance` (lines 39–45) allow clients to retrieve the values of `private decimal` instance variables `availableBalance` and `totalBalance`, respectively. Property `AvailableBalance` represents the amount of funds available for withdrawal. Property `TotalBalance` represents the amount of funds available, plus the amount of deposited funds pending confirmation of cash in deposit envelopes or clearance of checks in deposit envelopes.

`public` Methods of Class `Account`

Method `ValidatePIN` (lines 48–51) 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. 26.9. If the two PINs match, the method returns `true`; otherwise, it returns `false`.

Method `Credit` (lines 54–57) adds an amount of money (i.e., parameter `amount`) to an `Account` as part of a deposit transaction. This method adds the `amount` only to instance variable `totalBalance` (line 56). The money credited to an account during a deposit does not become available immediately, so we modify only the total balance. We assume that the bank updates the available balance appropriately at a later time, when the amount of cash in the deposit envelope has be verified and the checks in the deposit envelope have cleared. 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 instance variable `availableBalance` to confirm a deposit or to subtract from attribute `totalBalance` to reject a deposit.

Method `Debit` (lines 60–64) 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 instance variable `availableBalance` (line 62) and instance variable `totalBalance` (line 63), because a withdrawal affects both balances.

26.4.7 Class BankDatabase

Class `BankDatabase` (Fig. 26.19) models the bank 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. 26.9 that a `BankDatabase` is composed of zero or more objects of class `Account`. Line 5 declares attribute `accounts`—an array that will store `Account` objects—to implement this composition relationship. Class `BankDatabase` has a parameterless constructor (lines 8–15) that initializes `accounts` with new `Account` objects (lines 13–14). The `Account` constructor (Fig. 26.25, lines 11–18) has four parameters—the account number, the PIN assigned to the account, the initial available balance and the initial total balance.

```
1  // BankDatabase.cs
2  // Represents the bank account information database
3  public class BankDatabase
4  {
5     private Account[] accounts; // array of the bank's Accounts
6
7     // parameterless constructor initializes accounts
8     public BankDatabase()
9     {
10       // create two Account objects for testing and
11       // place them in the accounts array
12       accounts = new Account[2]; // create accounts array
13       accounts[0] = new Account(12345, 54321, 1000.00M, 1200.00M);
14       accounts[1] = new Account(98765, 56789, 200.00M, 200.00M);
15    }
16
17    // retrieve Account object containing specified account number
18    private Account GetAccount(int accountNumber)
19    {
20       // loop through accounts searching for matching account number
21       foreach (Account currentAccount in accounts)
22       {
23          if (currentAccount.AccountNumber == accountNumber)
24             return currentAccount;
25       }
26
27       // account not found
28       return null;
29    }
30
31    // determine whether user-specified account number and PIN match
32    // those of an account in the database
33    public bool AuthenticateUser(int userAccountNumber, int userPIN)
34    {
35       // attempt to retrieve the account with the account number
36       Account userAccount = GetAccount(userAccountNumber);
37
38       // if account exists, return result of Account function ValidatePIN
39       if (userAccount != null)
40          return userAccount.ValidatePIN(userPIN); // true if match
41       else
42          return false; // account number not found, so return false
43    }
44
45    // return available balance of Account with specified account number
46    public decimal GetAvailableBalance(int userAccountNumber)
47    {
```

Fig. 26.19 | Class `BankDatabase` represents the bank's account information database. (Part 1 of 2.)

```
48          Account userAccount = GetAccount(userAccountNumber);
49          return userAccount.AvailableBalance;
50       }
51
52       // return total balance of Account with specified account number
53       public decimal GetTotalBalance(int userAccountNumber)
54       {
55          Account userAccount = GetAccount(userAccountNumber);
56          return userAccount.TotalBalance;
57       }
58
59       // credit the Account with specified account number
60       public void Credit(int userAccountNumber, decimal amount)
61       {
62          Account userAccount = GetAccount(userAccountNumber);
63          userAccount.Credit(amount);
64       }
65
66       // debit the Account with specified account number
67       public void Debit(int userAccountNumber, decimal amount)
68       {
69          Account userAccount = GetAccount(userAccountNumber);
70          userAccount.Debit(amount);
71       }
72    }
```

Fig. 26.19 | Class `BankDatabase` represents the bank's account information database. (Part 2 of 2.)

Recall that class `BankDatabase` serves as an intermediary between class `ATM` and the actual `Account` objects that contain users' account information. Thus, methods of class `BankDatabase` invoke the corresponding methods and properties of the `Account` object belonging to the current ATM user.

`private` *Utility Method* `GetAccount`

We include `private` utility method `GetAccount` (lines 18–29) to allow the `BankDatabase` to obtain a reference to a particular `Account` within the `accounts` array. To locate the user's `Account`, the `BankDatabase` compares the value returned by property `AccountNumber` for each element of `accounts` to a specified account number until it finds a match. Lines 21–25 traverse the `accounts` array. If `currentAccount`'s account number equals the value of parameter `accountNumber`, the method returns `currentAccount`. If no account has the given account number, then line 28 returns `null`.

`public` *Methods*

Method `AuthenticateUser` (lines 33–43) proves or disproves the identity of an ATM user. This method takes a user-specified account number and a user-specified PIN as arguments and indicates whether they match the account number and PIN of an `Account` in the database. Line 36 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 40 returns the `bool` value returned by that object's `ValidatePIN` method. `BankDatabase`'s `AuthenticateUser` method does not perform the PIN comparison itself—rather, it forwards `userPIN` to the `Account` object's `ValidatePIN` method to do so. The value returned by `Account` method `ValidatePIN` (line 40) indicates whether the user-specified PIN matches the PIN of the user's `Account`, so method `AuthenticateUser` simply returns this value (line 40) to the client of the class (i.e., `ATM`).

The `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 account number is passed to the remaining `BankDatabase` methods as argument `userAccountNumber`. Methods `GetAvailableBalance` (lines 46–50), `GetTotalBalance` (lines 53–57), `Credit` (lines 60–64) and `Debit` (lines 67–71) 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`. `GetAvailableBalance` and `GetTotalBalance` return the values returned by the corresponding `Account` properties. Also, methods `Credit` and `Debit` simply redirect parameter `amount` to the `Account` methods they invoke.

26.4.8 Class Transaction

Class `Transaction` (Fig. 26.20) is an `abstract` base class that represents the notion of an ATM transaction. It contains the common features of derived classes `BalanceInquiry`, `Withdrawal` and `Deposit`. This class expands on the "skeleton" code first developed in Section 26.2. Line 3 declares this class to be `abstract`. Lines 5–7 declare the class's `private` instance variables. Recall from the class diagram of Fig. 26.10 that class `Transaction` contains the property `AccountNumber` that indicates the account involved in the `Transaction`. Line 5 implements the instance variable `accountNumber` to maintain the `AccountNumber` property's data. We derive attributes `screen` (implemented as instance variable `userScreen` in line 6) and `bankDatabase` (implemented as instance variable `database` in line 7) from class `Transaction`'s associations, modeled in Fig. 26.9. All transactions require access to the ATM's screen and the bank's database.

Class `Transaction` has a constructor (lines 10–16) 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 (line 3), this constructor is never called directly to instantiate `Transaction` objects. Instead, this constructor is invoked by the constructors of the `Transaction` derived classes via constructor initializers.

Class `Transaction` has three `public` read-only properties—`AccountNumber` (lines 19–25), `UserScreen` (lines 28–34) and `Database` (lines 37–43). Derived classes of `Transaction` inherit these properties and use them to gain access to class `Transaction`'s `private` instance variables. We chose the names of the `UserScreen` and `Database` properties for clarity—we wanted to avoid property names that are the same as the class names `Screen` and `BankDatabase`, which can be confusing.

Class `Transaction` also declares `abstract` method `Execute` (line 46). It does not make sense to provide an implementation for this method in class `Transaction`, because a generic transaction cannot be executed. Thus, we declare this method to be `abstract`, forcing each `Transaction` concrete derived class to provide its own implementation that executes the particular type of transaction.

```
// Transaction.cs5
// Abstract base class Transaction represents an ATM transaction.
public abstract class Transaction
{
   private int accountNumber; // account involved in the transaction
   private Screen userScreen; // reference to ATM's screen
   private BankDatabase database; // reference to account info database

   // three-parameter constructor invoked by derived classes
   public Transaction(int userAccount, Screen theScreen,
      BankDatabase theDatabase)
   {
      accountNumber = userAccount;
      userScreen = theScreen;
      database = theDatabase;
   }

   // read-only property that gets the account number
   public int AccountNumber
   {
      get
      {
         return accountNumber;
      }
   }

   // read-only property that gets the screen reference
   public Screen UserScreen
   {
      get
      {
         return userScreen;
      }
   }

   // read-only property that gets the bank database reference
   public BankDatabase Database
   {
      get
      {
         return database;
      }
   }

   // perform the transaction (overridden by each derived class)
   public abstract void Execute(); // no implementation here
}
```

Fig. 26.20 | `abstract` base class `Transaction` represents an ATM transaction.

26.4.9 Class `BalanceInquiry`

Class `BalanceInquiry` (Fig. 26.21) inherits from `Transaction` and represents an ATM balance inquiry transaction (line 3). `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 read-only properties. The BalanceInquiry constructor (lines 6–8) takes arguments corresponding to these attributes and forwards them to Transaction's constructor by invoking the constructor initializer with keyword base (line 8). The body of the constructor is empty.

Class BalanceInquiry overrides Transaction's abstract method Execute to provide a concrete implementation (lines 11–27) that performs the steps involved in a balance inquiry. Lines 14–15 obtain the specified Account's available balance by invoking the GetAvailableBalance method of the inherited property Database. Line 15 uses the inherited property AccountNumber to get the account number of the current user. Line 18 retrieves the specified Account's total balance. Lines 21–26 display the balance information on the ATM's screen using the inherited property UserScreen. Recall that DisplayDollarAmount takes a decimal argument and outputs it to the screen formatted as a dollar amount with a dollar sign. For example, if a user's available balance is 1000.50M, line 23 outputs $1,000.50. Line 26 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).

```
1  // BalanceInquiry.cs
2  // Represents a balance inquiry ATM transaction
3  public class BalanceInquiry : Transaction
4  {
5     // five-parameter constructor initializes base class variables
6     public BalanceInquiry(int userAccountNumber,
7        Screen atmScreen, BankDatabase atmBankDatabase)
8        : base(userAccountNumber, atmScreen, atmBankDatabase) {}
9
10    // performs transaction; overrides Transaction's abstract method
11    public override void Execute()
12    {
13       // get the available balance for the current user's Account
14       decimal availableBalance =
15          Database.GetAvailableBalance(AccountNumber);
16
17       // get the total balance for the current user's Account
18       decimal totalBalance = Database.GetTotalBalance(AccountNumber);
19
20       // display the balance information on the screen
21       UserScreen.DisplayMessageLine("\nBalance Information:");
22       UserScreen.DisplayMessage(" - Available balance: ");
23       UserScreen.DisplayDollarAmount(availableBalance);
24       UserScreen.DisplayMessage("\n - Total balance: ");
25       UserScreen.DisplayDollarAmount(totalBalance);
26       UserScreen.DisplayMessageLine("");
27    }
28 }
```

Fig. 26.21 | Class BalanceInquiry represents a balance inquiry ATM transaction.

26.4.10 Class Withdrawal

Class Withdrawal (Fig. 26.22) extends Transaction and represents an ATM withdrawal transaction. This class expands on the "skeleton" code for this class developed in

Fig. 26.11. Recall from the class diagram of Fig. 26.9 that class `Withdrawal` has one attribute, `amount`, which line 5 declares as a `decimal` instance variable. Figure 26.9 models associations between class `Withdrawal` and classes `Keypad` and `CashDispenser`, for which lines 6–7 implement reference attributes `keypad` and `cashDispenser`, respectively. Line 10 declares a constant corresponding to the cancel menu option.

```
1  // Withdrawal.cs
2  // Class Withdrawal represents an ATM withdrawal transaction.
3  public class Withdrawal : Transaction
4  {
5     private decimal amount; // amount to withdraw
6     private Keypad keypad; // reference to Keypad
7     private CashDispenser cashDispenser; // reference to cash dispenser
8
9     // constant that corresponds to menu option to cancel
10    private const int CANCELED = 6;
11
12    // five-parameter constructor
13    public Withdrawal(int userAccountNumber, Screen atmScreen,
14       BankDatabase atmBankDatabase, Keypad atmKeypad,
15       CashDispenser atmCashDispenser)
16       : base(userAccountNumber, atmScreen, atmBankDatabase)
17    {
18       // initialize references to keypad and cash dispenser
19       keypad = atmKeypad;
20       cashDispenser = atmCashDispenser;
21    }
22
23    // perform transaction, overrides Transaction's abstract method
24    public override void Execute()
25    {
26       bool cashDispensed = false; // cash was not dispensed yet
27
28       // transaction was not canceled yet
29       bool transactionCanceled = false;
30
31       // loop until cash is dispensed or the user cancels
32       do
33       {
34          // obtain the chosen withdrawal amount from the user
35          int selection = DisplayMenuOfAmounts();
36
37          // check whether user chose a withdrawal amount or canceled
38          if (selection != CANCELED)
39          {
40             // set amount to the selected dollar amount
41             amount = selection;
42
43             // get available balance of account involved
44             decimal availableBalance =
45                Database.GetAvailableBalance(AccountNumber);
46
```

Fig. 26.22 | Class `Withdrawal` represents an ATM withdrawal transaction. (Part 1 of 3.)

```
               // 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))
                  {
                     // debit the account to reflect the withdrawal
                     Database.Debit(AccountNumber, amount);

                     cashDispenser.DispenseCash(amount); // dispense cash
                     cashDispensed = true; // cash was dispensed

                     // instruct user to take cash
                     UserScreen.DisplayMessageLine(
                        "\nPlease take your cash from the cash dispenser.");
                  }
                  else // cash dispenser does not have enough cash
                     UserScreen.DisplayMessageLine(
                        "\nInsufficient cash available in the ATM." +
                        "\n\nPlease choose a smaller amount.");
               }
               else // not enough money available in user's account
                  UserScreen.DisplayMessageLine(
                     "\nInsufficient cash available in your account." +
                     "\n\nPlease choose a smaller amount.");
            }
            else
            {
               UserScreen.DisplayMessageLine("\nCanceling transaction...");
               transactionCanceled = true; // user canceled the transaction
            }
         } while ((!cashDispensed) && (!transactionCanceled));
      }

      // display a menu of withdrawal amounts and the option to cancel;
      // return the chosen amount or 6 if the user chooses to cancel
      private int DisplayMenuOfAmounts()
      {
         int userChoice = 0; // variable to store return value

         // 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
            UserScreen.DisplayMessageLine("\nWithdrawal options:");
            UserScreen.DisplayMessageLine("1 - $20");
            UserScreen.DisplayMessageLine("2 - $40");
            UserScreen.DisplayMessageLine("3 - $60");
            UserScreen.DisplayMessageLine("4 - $100");
            UserScreen.DisplayMessageLine("5 - $200");
```

Fig. 26.22 | Class `Withdrawal` represents an ATM withdrawal transaction. (Part 2 of 3.)

```
100          UserScreen.DisplayMessageLine("6 - Cancel transaction");
101          UserScreen.DisplayMessage(
102             "\nChoose a withdrawal option (1-6): ");
103
104          // get user input through keypad
105          int input = keypad.GetInput();
106
107          // determine how to proceed based on the input value
108          switch (input)
109          {
110             // if the user chose a withdrawal amount (i.e., option
111             // 1, 2, 3, 4, or 5), return the corresponding amount
112             // from the amounts array
113             case 1: case 2: case 3: case 4: case 5:
114                userChoice = amounts[ input ]; // save user's choice
115                break;
116             case CANCELED: // the user chose to cancel
117                userChoice = CANCELED; // save user's choice
118                break;
119             default:
120                UserScreen.DisplayMessageLine(
121                   "\nInvalid selection. Try again.");
122                break;
123          }
124       }
125
126       return userChoice;
127    }
128 }
```

Fig. 26.22 | Class `Withdrawal` represents an ATM withdrawal transaction. (Part 3 of 3.)

Class `Withdrawal`'s constructor (lines 13–21) has five parameters. It uses the constructor initializer to pass parameters `userAccountNumber`, `atmScreen` and `atmBankDatabase` to base class `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`, respectively.

Overriding `abstract` *Method* `Execute`

Class `Withdrawal` overrides `Transaction`'s `abstract` method `Execute` with a concrete implementation (lines 24–79) that performs the steps involved in a withdrawal. Line 26 declares and initializes a local `bool` variable `cashDispensed`. This variable indicates whether cash has been dispensed (i.e., whether the transaction has completed successfully) and is initially `false`. Line 29 declares and initializes to `false` a `bool` variable `transactionCanceled` to indicate that the transaction has not yet been canceled by the user.

Lines 32–78 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 (i.e., until `transactionCanceled` becomes `true`). 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 35 displays a menu of withdrawal amounts and obtains a user selection by calling `private` utility method `DisplayMenuOfAmounts` (declared in lines 83–127). 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.

Displaying Options With `private` Utility Method `DisplayMenuOfAmounts`

Method `DisplayMenuOfAmounts` (lines 83–127) first declares local variable `userChoice` (initially `0`) to store the value that the method will return (line 85). Line 88 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 91–124 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 94–102 display the withdrawal menu on the screen and prompt the user to enter a choice. Line 105 obtains integer `input` through the keypad. The `switch` statement at lines 108–123 determines how to proceed based on the user's input. If the user selects `1`, `2`, `3`, `4` or `5`, line 114 sets `userChoice` to the value of the element in the `amounts` array at index `input`. For example, if the user enters `3` to withdraw $60, line 114 sets `userChoice` to the value of `amounts[ 3 ]`—i.e., `60`. Variable `userChoice` no longer equals `0`, so the `while` at lines 91–124 terminates, and line 126 returns `userChoice`. If the user selects the cancel menu option, line 117 executes, setting `userChoice` to `CANCELED` and causing the method to return this value. If the user does not enter a valid menu selection, lines 120–121 display an error message, and the user is returned to the withdrawal menu.

The `if` statement at line 38 in method `Execute` determines whether the user has selected a withdrawal amount or chosen to cancel. If the user cancels, line 75 displays an appropriate message to the user before control is returned to the calling method—`ATM` method `PerformTransactions`. If the user has chosen a withdrawal amount, line 41 assigns local variable `selection` to instance variable `amount`. Lines 44–45 retrieve the available balance of the current user's `Account` and store it in a local `decimal` variable `availableBalance`. Next, the `if` statement at line 48 determines whether the selected amount is less than or equal to the user's available balance. If it is not, lines 69–71 display an error message. Control then continues to the end of the `do…while` statement, and the loop repeats because both `cashDispensed` and `transactionCanceled` are still `false`. If the user's balance is high enough, the `if` statement at line 51 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 64–66 display an error message, and the `do…while` statement repeats. If sufficient cash is available, the requirements for the withdrawal are satisfied, and line 54 debits the user's account in the database by `amount`. Lines 56–57 then instruct the cash dispenser to dispense the cash to the user and set `cashDispensed` to `true`. Finally, lines 60–61 display a message to the user to take the dispensed cash. Because `cashDispensed` is now `true`, control continues after the `do…while` statement. No additional statements appear below the loop, so the method returns control to class `ATM`.

26.4.11 Class `Deposit`

Class `Deposit` (Fig. 26.23) inherits from `Transaction` and represents an ATM deposit transaction. Recall from the class diagram of Fig. 26.10 that class `Deposit` has one attribute, `amount`, which line 5 declares as a `decimal` instance variable. Lines 6–7 create reference attributes `keypad` and `depositSlot` that implement the associations between class `Deposit` and classes `Keypad` and `DepositSlot`, modeled in Fig. 26.9. Line 10 declares a constant `CANCELED` that corresponds to the value a user enters to cancel a deposit transaction.

```
1  // Deposit.cs
2  // Represents a deposit ATM transaction.
3  public class Deposit : Transaction
4  {
5     private decimal amount; // amount to deposit
6     private Keypad keypad; // reference to the Keypad
7     private DepositSlot depositSlot; // reference to the deposit slot
8
9     // constant representing cancel option
10    private const int CANCELED = 0;
11
12    // five-parameter constructor initializes class's instance variables
13    public Deposit(int userAccountNumber, Screen atmScreen,
14       BankDatabase atmBankDatabase, Keypad atmKeypad,
15       DepositSlot atmDepositSlot)
16       : base(userAccountNumber, atmScreen, atmBankDatabase)
17    {
18       // initialize references to keypad and deposit slot
19       keypad = atmKeypad;
20       depositSlot = atmDepositSlot;
21    } -parameter constructor
22
23    // perform transaction; overrides Transaction's abstract method
24    public override void Execute()
25    {
26       amount = PromptForDepositAmount(); // get deposit amount from user
27
28       // check whether user entered a deposit amount or canceled
29       if (amount != CANCELED)
30       {
31          // request deposit envelope containing specified amount
32          UserScreen.DisplayMessage(
33             "\nPlease insert a deposit envelope containing ");
34          UserScreen.DisplayDollarAmount(amount);
35          UserScreen.DisplayMessageLine(" in the deposit slot.");
36
37          // retrieve deposit envelope
38          bool envelopeReceived = depositSlot.IsDepositEnvelopeReceived();
39
40          // check whether deposit envelope was received
41          if (envelopeReceived)
42          {
43             UserScreen.DisplayMessageLine(
44                "\nYour envelope has been received.\n" +
45                "The money just deposited will not be available " +
46                "until we \nverify the amount of any " +
47                "enclosed cash, and any enclosed checks clear.");
```

Fig. 26.23 | Class `Deposit` represents an ATM deposit transaction. (Part 1 of 2.)

```
48
49             // credit account to reflect the deposit
50             Database.Credit(AccountNumber, amount);
51          }
52          else
53             UserScreen.DisplayMessageLine(
54                "\nYou did not insert an envelope, so the ATM has " +
55                "canceled your transaction.");
56       }
57       else
58          UserScreen.DisplayMessageLine("\nCanceling transaction...");
59    }
60
61    // prompt user to enter a deposit amount to credit
62    private decimal PromptForDepositAmount()
63    {
64       // display the prompt and receive input
65       UserScreen.DisplayMessage(
66          "\nPlease input a deposit amount in CENTS (or 0 to cancel): ");
67       int input = keypad.GetInput();
68
69       // check whether the user canceled or entered a valid amount
70       if (input == CANCELED)
71          return CANCELED;
72       else
73          return input / 100.00M;
74    }
75 }
```

Fig. 26.23 | Class `Deposit` represents an ATM deposit transaction. (Part 2 of 2.)

Class `Deposit` contains a constructor (lines 13–21) that passes three parameters to base class `Transaction`'s constructor using a constructor initializer. The constructor also has parameters `atmKeypad` and `atmDepositSlot`, which it assigns to the corresponding reference instance variables (lines 19–20).

Overriding `abstract` *Method* `Execute`

Method `Execute` (lines 24–59) overrides `abstract` method `Execute` in base class `Transaction` with a concrete implementation that performs the steps required in a deposit transaction. Line 26 prompts the user to enter a deposit amount by invoking `private` utility method `PromptForDepositAmount` (declared in lines 62–74) and sets attribute `amount` to the value returned. Method `PromptForDepositAmount` asks the user to enter a deposit amount as an integer number of cents (because the ATM's keypad does not contain a decimal point; this is consistent with many real ATMs) and returns the `decimal` value representing the dollar amount to be deposited.

Getting Deposit Amount with `private` *Utility Method* `PromptForDepositAmount`

Lines 65–66 in method `PromptForDepositAmount` display a message asking the user to input a deposit amount as a number of cents or "0" to cancel the transaction. Line 67 receives the user's input from the keypad. The `if` statement at lines 70–73 determines whether the user has entered a deposit amount or chosen to cancel. If the user chooses to cancel, line 71 returns constant `CANCELED`. Otherwise, line 73 returns the deposit amount after converting the `int` number of cents to a dollar-and-cents amount by dividing by the `decimal` literal `100.00M`. For example, if the user enters `125` as the number of cents, line 73 returns `125` divided by `100.00M`, or `1.25`—125 cents is $1.25.

The `if` statement at lines 29–58 in method `Execute` determines whether the user has chosen to cancel the transaction instead of entering a deposit amount. If the user cancels, line 58 displays an appropriate message, and the method returns. If the user enters a deposit amount, lines 32–35 instruct the user to insert a deposit envelope with the correct amount. Recall that `Screen` method `DisplayDollarAmount` outputs a `decimal` value formatted as a dollar amount (including the dollar sign).

Line 38 sets a local `bool` variable to the value returned by `depositSlot`'s `IsDepositEnvelopeReceived` method, indicating whether a deposit envelope has been received. Recall that we coded method `IsDepositEnvelopeReceived` (lines 7–10 of Fig. 26.17) to always return `true`, because we are simulating the functionality of the deposit slot and assume that the user always inserts an envelope in a timely fashion (i.e., within the two-minute time limit). 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 `IsDepositEnvelopeReceived` that could return `false`. Lines 43–50 execute if the deposit slot receives an envelope. Lines 43–47 display an appropriate message to the user. Line 50 credits the user's account in the database with the deposit amount. Lines 53–55 execute if the deposit slot does not receive a deposit envelope. In this case, we display a message stating that the ATM has canceled the transaction. The method then returns without crediting the user's account.

26.4.12 Class ATMCaseStudy

Class `ATMCaseStudy` (Fig. 26.24) simply allows us to start, or "turn on," the ATM and test the implementation of our ATM system model. Class `ATMCaseStudy`'s `Main` method (lines 6–10) simply instantiates a new `ATM` object named `theATM` (line 8) and invokes its `Run` method (line 9) to start the ATM.

```
1  // ATMCaseStudy.cs
2  // App for testing the ATM case study.
3  public class ATMCaseStudy
4  {
5     // Main method is the app's entry point
6     public static void Main(string[] args)
7     {
8        ATM theATM = new ATM();
9        theATM.Run();
10    }
11 }
```

Fig. 26.24 | Class `ATMCaseStudy` starts the ATM.

26.5 Wrap-Up

In this chapter, you used inheritance to tune the design of the ATM software system, and you fully implemented the ATM in C#. Congratulations on completing the entire ATM case study! We hope you found this experience to be valuable and that it reinforced many of the object-oriented programming concepts that you've learned.

Answers to Self-Review Exercises

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

26.2 b.

26.3 The design for class `Account` yields the code in Fig. 26.25. We used `public` auto-implemented properties `AvailableBalance` and `TotalBalance` to store the data that methods `Credit` and `Debit`, will manipulate.

```
// Fig. 26.25: Account.cs
// Class Account represents a bank account.
public class Account
{
   private int accountNumber; // account number
   private int pin; // PIN for authentication

   // automatic read-only property AvailableBalance
   public decimal AvailableBalance { get; private set; }

   // automatic read-only property TotalBalance
   public decimal TotalBalance { get; private set; }

   // parameterless constructor
   public Account()
   {
      // constructor body code
   }

   // validates user PIN
   public bool ValidatePIN()
   {
      // ValidatePIN method body code
   }

   // credits the account
   public void Credit()
   {
      // Credit method body code
   }

   // debits the account
   public void Debit()
   {
```

Fig. 26.25 | C# code for class `Account` based on Figs. 26.1 and 26.2. (Part 1 of 2.)

```
         // Debit method body code
      }
   }
```

Fig. 26.25 | C# code for class `Account` based on Figs. 26.1 and 26.2. (Part 2 of 2.)

26.4 b.

26.5 False. The UML requires that we italicize abstract class names and operation names.

26.6 The design for class `Transaction` yields the code in Fig. 26.26. In the implementation, a constructor initializes `private` instance variables `userScreen` and `database` to actual objects, and read-only properties `UserScreen` and `Database` access these instance variables. These properties allow classes derived from `Transaction` to access the ATM's screen and interact with the bank's database. We chose the names of the `UserScreen` and `Database` properties for clarity—we wanted to avoid property names that are the same as the class names `Screen` and `BankDatabase`, which can be confusing.

```
// Fig. 26.26: Transaction.cs
// Abstract base class Transaction represents an ATM transaction.
public abstract class Transaction
{
   private int accountNumber; // indicates account involved
   private Screen userScreen; // ATM's screen
   private BankDatabase database; // account info database

   // parameterless constructor
   public Transaction()
   {
      // constructor body code
   }

   // read-only property that gets the account number
   public int AccountNumber
   {
      get
      {
         return accountNumber;
      }
   }

   // read-only property that gets the screen reference
   public Screen UserScreen
   {
      get
      {
         return userScreen;
      }
   }

```

Fig. 26.26 | C# code for class `Transaction` based on Figures 26.9 and 26.10. (Part 1 of 2.)

```
   // read-only property that gets the bank database reference
   public BankDatabase Database
   {
      get
      {
         return database;
      }
   }

   // perform the transaction (overridden by each derived class)
   public abstract void Execute();
}
```

Fig. 26.26 | C# code for class `Transaction` based on Figures 26.9 and 26.10. (Part 2 of 2.)