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

26

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. We now begin implementing our object-oriented design in C++. In Section 26.2, we show how to convert class diagrams to C++ code. In Section 26.3, we tune the design with inheritance and polymorphism. Then we present a full C++ code implementation of the ATM software in Section 26.4. The code is carefully commented and the discussions of the implementation are thorough and precise. Studying this application provides the opportunity for you to see a more substantial application of the kind you're likely to encounter in industry.

26.2 Starting to Program the Classes of the ATM System

[*Note:* **This section can be studied after Chapter 9.**]

Visibility

We now apply access specifiers to the members of our classes. Access specifiers `public` and `private` 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 operations of our classes should be `public` and which should be `private`.

Previously, we observed that data members normally should be `private` and that member functions invoked by clients of a given class should be `public`. Member functions that are called only by other member functions of the class as "utility functions," however, normally should be `private`. The UML employs **visibility markers** for modeling the visibility of attributes and operations. Public visibility is indicated by placing a plus sign (+) before an operation or an attribute; 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 communica-

Fig. 26.1 | Class diagram with visibility markers.

tion and sequence diagrams (see Section 25.8). When implementing a system designed using the UML, you use navigability arrows to help determine which objects need references or pointers 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.

Like the class diagram of Fig. 25.10, 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 from Section 25.4 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,

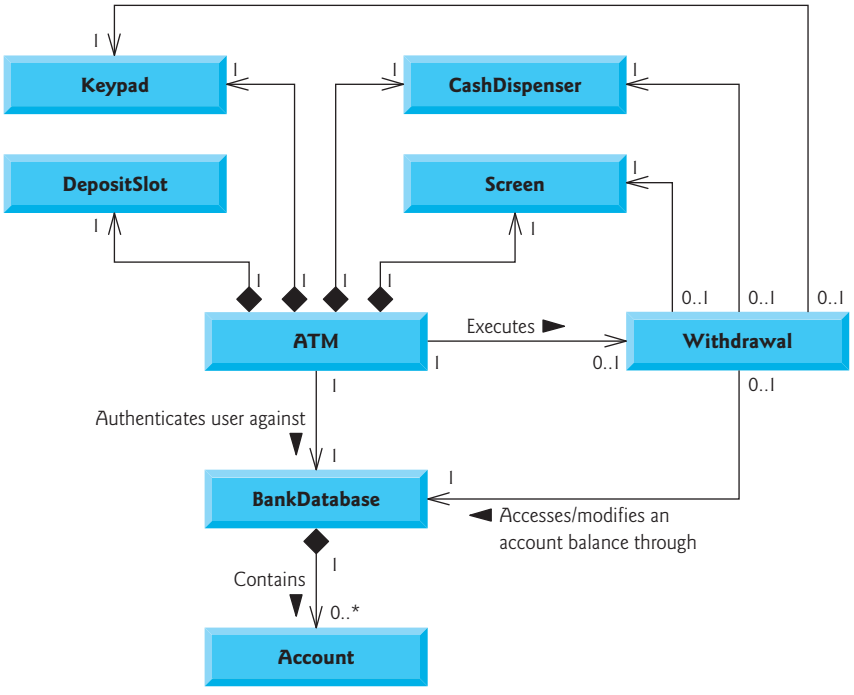

Fig. 26.2 | Class diagram with navigability arrows.

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 `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 are now ready to begin implementing the ATM system. We first convert the classes in the diagrams of Fig. 26.1 and Fig. 26.2 into C++ header files. This code will represent the "skeleton" of the system. In Section 26.3, we modify the header files to incorporate the object-oriented concept of inheritance. In Section 26.4, we present the complete working C++ code for our model.

As an example, we begin to develop the header file 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 the following five guidelines for each class:

1. Use the name in the first compartment of a class in a class diagram to define the class in a header file (Fig. 26.3). Use `#ifndef`, `#define` and `#endif` preprocessor

directives to prevent the header from being included more than once in a program.

```
1  // Fig. 26.3: Withdrawal.h
2  // Definition of class Withdrawal that represents a withdrawal transaction
3  #ifndef WITHDRAWAL_H
4  #define WITHDRAWAL_H
5
6  class Withdrawal
7  {
8  }; // end class Withdrawal
9
10 #endif // WITHDRAWAL_H
```

Fig. 26.3 | Definition of class `Withdrawal` enclosed in preprocessor wrappers.

2. Use the attributes located in the class's second compartment to declare the data members. For example, the `private` attributes `accountNumber` and `amount` of class `Withdrawal` yield the code in Fig. 26.4.

```
1  // Fig. 26.4: Withdrawal.h
2  // Definition of class Withdrawal that represents a withdrawal transaction
3  #ifndef WITHDRAWAL_H
4  #define WITHDRAWAL_H
5
6  class Withdrawal
7  {
8  private:
9     // attributes
10    int accountNumber; // account to withdraw funds from
11    double amount; // amount to withdraw
12 }; // end class Withdrawal
13
14 #endif // WITHDRAWAL_H
```

Fig. 26.4 | Adding attributes to the `Withdrawal` class header file.

3. Use the associations described in the class diagram to declare references (or pointers, where appropriate) to other objects. For example, 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 handles on these objects to send messages to them, so lines 19–22 of Fig. 26.5 declare four references as `private` data members. In the implementation of class `Withdrawal` in Section 26.4, a constructor initializes these data members with references to actual objects. Lines 6–9 `#include` the header files containing the definitions of classes `Screen`, `Keypad`, `CashDispenser` and `BankDatabase` so that we can declare references to objects of these classes in lines 19–22.
4. It turns out that including the header files for classes `Screen`, `Keypad`, `CashDispenser` and `BankDatabase` in Fig. 26.5 does more than is necessary. Class `With-`

`drawal` contains *references* to objects of these classes—it does not contain actual objects—and the amount of information required by the compiler to create a reference differs from that which is required to create an object. Recall that creating an object requires that you provide the compiler with a definition of the class that

```
1  // Fig. 26.5: Withdrawal.h
2  // Definition of class Withdrawal that represents a withdrawal transaction
3  #ifndef WITHDRAWAL_H
4  #define WITHDRAWAL_H
5
6  #include "Screen.h" // include definition of class Screen
7  #include "Keypad.h" // include definition of class Keypad
8  #include "CashDispenser.h" // include definition of class CashDispenser
9  #include "BankDatabase.h" // include definition of class BankDatabase
10
11 class Withdrawal
12 {
13 private:
14    // attributes
15    int accountNumber; // account to withdraw funds from
16    double amount; // amount to withdraw
17
18    // references to associated objects
19    Screen &screen; // reference to ATM's screen
20    Keypad &keypad; // reference to ATM's keypad
21    CashDispenser &cashDispenser; // reference to ATM's cash dispenser
22    BankDatabase &bankDatabase; // reference to the account info database
23 }; // end class Withdrawal
24
25 #endif // WITHDRAWAL_H
```

Fig. 26.5 | Declaring references to objects associated with class `Withdrawal`.

introduces the name of the class as a new user-defined type and indicates the data members that determine how much memory is required to store the object. Declaring a *reference* (or pointer) to an object, however, requires only that the compiler knows that the object's class exists—it does not need to know the size of the object. Any reference (or pointer), regardless of the class of the object to which it refers, contains only the memory address of the actual object. The amount of memory required to store an address is a physical characteristic of the computer's hardware. The compiler thus knows the size of any reference (or pointer). As a result, including a class's full header file when declaring only a reference to an object of that class is unnecessary—we need to introduce the name of the class, but we do not need to provide the data layout of the object, because the compiler already knows the size of all references. C++ provides a statement called a **forward declaration** that signifies that a header file contains references or pointers to a class, but that the class definition lies outside the header file. We can replace the `#includes` in the `Withdrawal` class definition of Fig. 26.5 with forward declarations of classes `Screen`, `Keypad`, `CashDispenser` and `BankDatabase` (lines 6–9 in

Fig. 26.6). Rather than `#include` the entire header file for each of these classes, we place only a forward declaration of each class in the header file for class `Withdrawal`. If class `Withdrawal` contained actual objects instead of references (i.e., if the ampersands in lines 19–22 were omitted), then we'd need to `#include` the full header files.

Using a forward declaration (where possible) instead of including a full header file helps avoid a preprocessor problem called a **circular include**. This problem occurs when the header file for a class `A` `#include`s the header file for a class `B` and vice versa. Some preprocessors are not be able to resolve such `#include` directives, causing a compilation error. If class `A`, for example, uses only a reference to an object of class `B`, then the `#include` in class `A`'s header file can be replaced by a forward declaration of class `B` to prevent the circular include.

```
1  // Fig. 26.6: Withdrawal.h
2  // Definition of class Withdrawal that represents a withdrawal transaction
3  #ifndef WITHDRAWAL_H
4  #define WITHDRAWAL_H
5
6  class Screen; // forward declaration of class Screen
7  class Keypad; // forward declaration of class Keypad
8  class CashDispenser; // forward declaration of class CashDispenser
9  class BankDatabase; // forward declaration of class BankDatabase
10
11 class Withdrawal
12 {
13 private:
14    // attributes
15    int accountNumber; // account to withdraw funds from
16    double amount; // amount to withdraw
17
18    // references to associated objects
19    Screen &screen; // reference to ATM's screen
20    Keypad &keypad; // reference to ATM's keypad
21    CashDispenser &cashDispenser; // reference to ATM's cash dispenser
22    BankDatabase &bankDatabase; // reference to the account info database
23 }; // end class Withdrawal
24
25 #endif // WITHDRAWAL_H
```

Fig. 26.6 | Using forward declarations in place of `#include` directives.

5. Use the operations located in the third compartment of Fig. 26.1 to write the function prototypes of the class's member functions. If we've not yet specified a return type for an operation, we declare the member function with return type `void`. Refer to the class diagrams of Figs. 6.21–6.24 to declare any necessary parameters. For example, adding the `public` operation `execute` in class `Withdrawal`, which has an empty parameter list, yields the prototype in line 15 of Fig. 26.7. [*Note:* We code the definitions of member functions in `.cpp` files when we implement the complete ATM system in Section 26.4.]

Software Engineering Observation 26.1

Several UML modeling tools can convert UML-based designs into C++ code, considerably speeding the implementation process. For more information on these "automatic" code generators, refer to our UML Resource Center at `www.deitel.com/UML/`.

This concludes our discussion of the basics of generating class header files from UML diagrams. In Section 26.3, we demonstrate how to modify the header files to incorporate the object-oriented concept of inheritance.

```
1  // Fig. 26.7: Withdrawal.h
2  // Definition of class Withdrawal that represents a withdrawal transaction
3  #ifndef WITHDRAWAL_H
4  #define WITHDRAWAL_H
5
6  class Screen; // forward declaration of class Screen
7  class Keypad; // forward declaration of class Keypad
8  class CashDispenser; // forward declaration of class CashDispenser
9  class BankDatabase; // forward declaration of class BankDatabase
10
11 class Withdrawal
12 {
13 public:
14    // operations
15    void execute(); // perform the transaction
16 private:
17    // attributes
18    int accountNumber; // account to withdraw funds from
19    double amount; // amount to withdraw
20
21    // references to associated objects
22    Screen &screen; // reference to ATM's screen
23    Keypad &keypad; // reference to ATM's keypad
24    CashDispenser &cashDispenser; // reference to ATM's cash dispenser
25    BankDatabase &bankDatabase; // reference to the account info database
26 }; // end class Withdrawal
27
28 #endif // WITHDRAWAL_H
```

Fig. 26.7 | Adding operations to the `Withdrawal` class header file.

Self-Review Exercises for Section 26.2

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 that:
a) we can navigate from the `Screen` to the `ATM`
b) we can navigate from the `ATM` to the `Screen`
c) Both a and b; the association is bidirectional
d) None of the above

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

26.3 Incorporating Inheritance into the ATM System

[*Note:* **This section can be studied after Chapter 12.**]
We now revisit our ATM system design to see how it might benefit from inheritance. To apply inheritance, we first look for *commonality* among classes in the system. We create an inheritance hierarchy to model similar (yet not identical) classes in a more efficient and elegant 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 our updated design is translated into C++ header files.

In Section 25.4, we encountered the problem of representing a financial transaction in the system. Rather than create one class to represent all transaction types, we decided to create three individual transaction classes—`BalanceInquiry`, `Withdrawal` and `Deposit`—to represent the transactions that the ATM system can perform. Figure 26.8 shows the attributes and operations of these classes, which have one attribute (`accountNumber`) and one operation (`execute`) in common. Each class requires attribute `accountNumber` to specify the account to which the transaction applies. Each class contains operation `execute`, which the `ATM` invokes to perform the transaction. Clearly, `BalanceInquiry`, `Withdrawal` and `Deposit` represent *types of* transactions. Figure 26.8 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. 26.9).

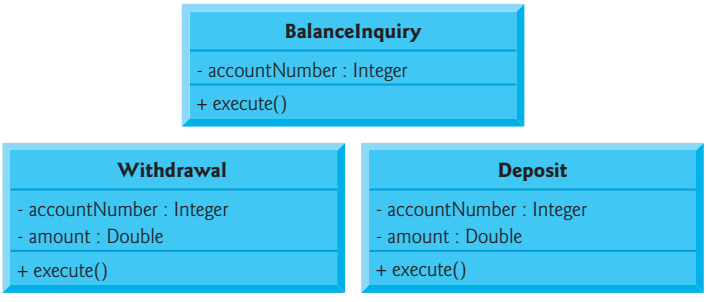

Fig. 26.8 | Attributes and operations of classes `BalanceInquiry`, `Withdrawal` and `Deposit`.

The UML specifies a relationship called a **generalization** to model inheritance. Figure 26.9 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`. Class `Transaction` is said to be a generalization of its derived classes. The derived classes are said to be **specializations** of class `Transaction`.

Classes `BalanceInquiry`, `Withdrawal` and `Deposit` share integer attribute `accountNumber`, so we factor out this common attribute and place it in base class `Transaction`. We no longer list `accountNumber` in the second compartment of each derived class, because the three derived classes inherit this attribute from `Transaction`. Recall, however, that derived classes cannot access `private` attributes of a base class. We therefore include

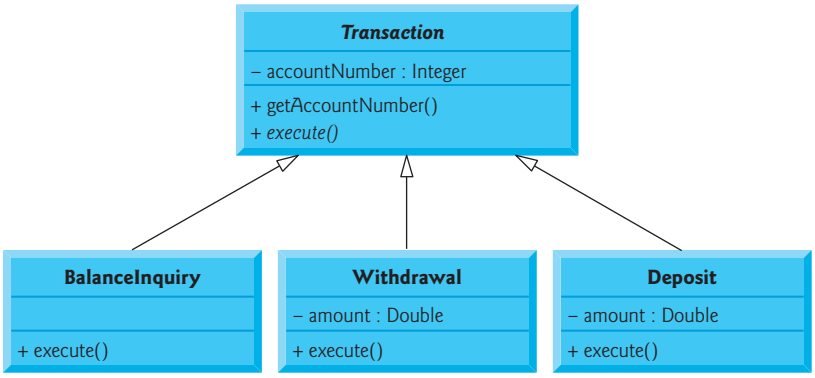

Fig. 26.9 | Class diagram modeling generalization relations\hip between base class `Transaction` and derived classes `BalanceInquiry`, `Withdrawal` and `Deposit`.

`public` member function `getAccountNumber` in class `Transaction`. Each derived class inherits this member function, enabling the derived class to access its `accountNumber` as needed to execute a transaction.

According to Fig. 26.8, classes `BalanceInquiry`, `Withdrawal` and `Deposit` also share operation `execute`, so base class `Transaction` should contain `public` member function `execute`. However, it does not make sense to implement `execute` in class `Transaction`, because the functionality that this member function provides depends on the specific type of the actual transaction. We therefore declare member function `execute` as a pure `virtual` function in base class `Transaction`. 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 pure `virtual` member function `execute` to make the derived class concrete. The UML requires that we place abstract class names (and pure `virtual` functions—**abstract operations** in the UML) in italics, so `Transaction` and its member function `execute` appear in italics in Fig. 26.9. Operation `execute` is not italicized in derived classes `BalanceInquiry`, `Withdrawal` and `Deposit`. Each derived class overrides base class `Transaction`'s `execute` member function with an appropriate implementation. Figure 26.9 includes operation `execute` in the third compartment of classes `BalanceInquiry`, `Withdrawal` and `Deposit`, because each class has a different concrete implementation of the overridden member function.

Processing `Transaction`s Polymorphically

A derived class can inherit interface and/or 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 functions that should be defined by each class in the hierarchy, but the individual derived classes provide their own implementations of the function(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." Each class derived from `Transaction` inherits some implementation details (e.g., data member `accountNumber`), but the primary benefit of incorporating inheritance into our system is that

the derived classes share a common interface (e.g., pure `virtual` member function `execute`). The `ATM` can aim a `Transaction` pointer at any transaction, and when the `ATM` invokes `execute` through this pointer, the version of `execute` appropriate to that transaction (i.e., the version implemented in that derived class's `.cpp` file) runs automatically. For example, suppose a user chooses to perform a balance inquiry. The `ATM` aims a `Transaction` pointer at a new object of class `BalanceInquiry`; the compiler allows this because a `BalanceInquiry` *is a* `Transaction`. When the `ATM` uses this pointer to invoke `execute`, `BalanceInquiry`'s version of `execute` is called.

This polymorphic approach also makes the system easily *extensible*. Should we wish to create a new transaction type (e.g., funds transfer or bill payment), we would just create an additional `Transaction` derived class that overrides the `execute` member function 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.

As you learned earlier in the chapter, an abstract class like `Transaction` is one for which you never intend to 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 executes. You may wonder why we bother to include pure `virtual` member function `execute` in class `Transaction` if `execute` lacks a concrete implementation. Conceptually, we include this member function because it's the defining behavior of all transactions—executing. Technically, we must include member function `execute` in base class `Transaction` so that the `ATM` (or any other class) can polymorphically invoke each derived class's overridden version of this function through a `Transaction` pointer or reference.

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

Derived classes `BalanceInquiry`, `Withdrawal` and `Deposit` inherit attribute `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 this attribute, 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).

Class Diagram with `Transaction` *Hierarchy Incorporated*

Figure 26.10 presents an updated class diagram of our model that incorporates inheritance and introduces class `Transaction`. We model an association between class `ATM` and class `Transaction` to show that the `ATM`, at any given moment, either is executing a transaction or is not (i.e., zero or one objects of type `Transaction` exist in the system at a time). Because a `Withdrawal` is a type of `Transaction`, we no longer draw an association line directly between class `ATM` and class `Withdrawal`—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 `BalanceInqui-`

ry and Deposit and class ATM. Note again the use of triangular hollow arrowheads to indicate the specializations of class Transaction, as indicated in Fig. 26.9.

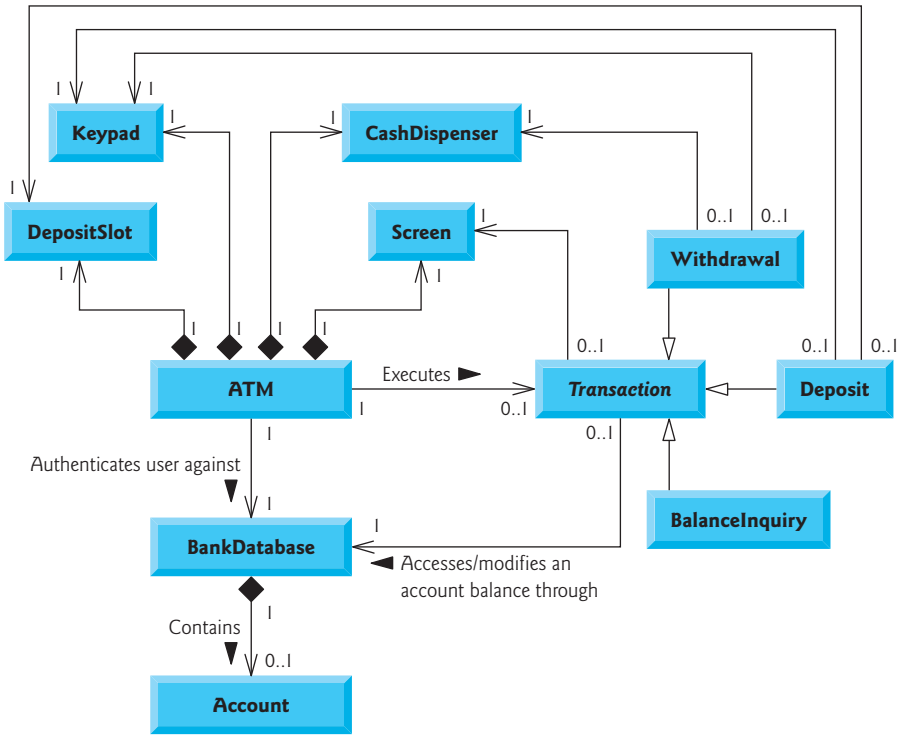

Fig. 26.10 | Class diagram of the ATM system (incorporating inheritance). Note that abstract class name Transaction appears in italics.

We also add an association between class Transaction and the BankDatabase (Fig. 26.10). All Transactions require a reference to the BankDatabase so they can access and modify account information. Each Transaction derived class inherits this reference, so we no longer model the association between class Withdrawal and the 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. We do not move these associations to base class Transaction, because the association with the Keypad applies only to classes Withdrawal and Deposit, and the association with the CashDispenser applies only to class Withdrawal.

Our class diagram incorporating inheritance (Fig. 26.10) also models Deposit and BalanceInquiry. We show associations between Deposit and both the DepositSlot and the Keypad. BalanceInquiry takes part in no associations other than those inherited from

class `Transaction`—a `BalanceInquiry` interacts only with the `BankDatabase` and the `Screen`.

Figure 26.1 showed attributes and operations with visibility markers. Now we present a modified class diagram in Fig. 26.11 that includes abstract base class `Transaction`. This abbreviated diagram does not show inheritance relationships (these appear in Fig. 26.10), but instead shows the attributes and operations after we've 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. 26.10—we do, however, include them in the C++ implementation. We also omit all operation parameters, as we did in Fig. 26.1—incorporating inheritance does not affect the parameters already modeled in Figs. 25.18–25.21.

Fig. 26.11 | Class diagram after incorporating inheritance into the system.

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 Fig. 26.10 and Fig. 26.11), 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. However, when examining classes modeled in this fashion, it's crucial to consider both class diagrams to get a complete view of the classes. For example, one must refer to Fig. 26.10 to observe the inheritance relationship between `Transaction` *and its derived classes that is omitted from Fig. 26.11.*

Implementing the ATM System Design Incorporating Inheritance

We now modify our implementation to 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 26.12 contains a portion of class `Withdrawal`'s header file, in which the class definition indicates the inheritance relationship between `Withdrawal` and `Transaction` (line 9).

```
1  // Fig. 26.12: Withdrawal.h
2  // Definition of class Withdrawal that represents a withdrawal transaction
3  #ifndef WITHDRAWAL_H
4  #define WITHDRAWAL_H
5
6  #include "Transaction.h" // Transaction class definition
7
8  // class Withdrawal derives from base class Transaction
9  class Withdrawal : public Transaction
10 {
11 }; // end class Withdrawal
12
13 #endif // WITHDRAWAL_H
```

Fig. 26.12 | `Withdrawal` class definition that derives from `Transaction`.

2. If class `A` is an abstract class and class `B` is derived from class `A`, then class `B` must implement the pure `virtual` functions of class `A` if class `B` is to be a concrete class. For example, class `Transaction` contains pure `virtual` function `execute`, so class `Withdrawal` must implement this member function if we want to instantiate a `Withdrawal` object. Figure 26.13 contains the C++ header file for class `Withdrawal` from Fig. 26.10 and Fig. 26.11. Class `Withdrawal` inherits data member `accountNumber` from base class `Transaction`, so `Withdrawal` does not declare this data member. Class `Withdrawal` also inherits references to the `Screen` and the `BankDatabase` from its base class `Transaction`, so we do not include these references in our code. Figure 26.11 specifies attribute `amount` and operation `execute` for class `Withdrawal`. Line 19 of Fig. 26.13 declares a data member for attribute `amount`. Line 16 contains the function prototype for operation `execute`. Recall

that, to be a concrete class, derived class `Withdrawal` must provide a concrete implementation of the pure `virtual` function `execute` in base class `Transaction`. The prototype in line 16 signals your intent to override the base class pure `virtual` function. You must provide this prototype if you'll provide an implementation in the `.cpp` file. We present this implementation in Section 26.4. The `keypad` and `cashDispenser` references (lines 20–21) are data members derived from `Withdrawal`'s associations in Fig. 26.10. In the implementation of this class in Section 26.4, a constructor initializes these references to actual objects. Once again, to be able to compile the declarations of the references in lines 20–21, we include the forward declarations in lines 8–9.

```
1  // Fig. 26.13: Withdrawal.h
2  // Definition of class Withdrawal that represents a withdrawal transaction
3  #ifndef WITHDRAWAL_H
4  #define WITHDRAWAL_H
5
6  #include "Transaction.h" // Transaction class definition
7
8  class Keypad; // forward declaration of class Keypad
9  class CashDispenser; // forward declaration of class CashDispenser
10
11 // class Withdrawal derives from base class Transaction
12 class Withdrawal : public Transaction
13 {
14 public:
15    // member function overriding execute in base class Transaction
16    virtual void execute(); // perform the transaction
17 private:
18    // attributes
19    double amount; // amount to withdraw
20    Keypad &keypad; // reference to ATM's keypad
21    CashDispenser &cashDispenser; // reference to ATM's cash dispenser
22 }; // end class Withdrawal
23
24 #endif // WITHDRAWAL_H
```

Fig. 26.13 | `Withdrawal` class header file based on Fig. 26.10 and Fig. 26.11.

ATM Case Study Wrap-Up

This concludes our object-oriented design of the ATM system. A complete C++ implementation of the ATM system in 850 lines of code appears in Section 26.4. This working implementation uses key programming notions, including classes, objects, encapsulation, visibility, composition, inheritance and polymorphism. The code is abundantly commented and conforms to the coding practices you've learned. Mastering this code is a wonderful capstone experience.

Self-Review Exercises for Section 26.3

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 operation names.

26.6 Write a C++ header file to begin implementing the design for class `Transaction` specified in Fig. 26.10 and Fig. 26.11. Be sure to include `private` references based on class `Transaction`'s associations. Also be sure to include `public` *get* functions for any of the `private` data members that the derived classes must access to perform their tasks.

26.4 ATM Case Study Implementation

This section contains the complete working implementation of the ATM system that we designed in Chapter 25 and this chapter. We consider the classes in the order in which we identified them in Section 25.4:

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

We apply the guidelines discussed in Sections 26.2 and 26.3 to code these classes based on how we modeled them in the UML class diagrams of Figs. 26.10 and 26.11. To develop the definitions of classes' member functions, we refer to the activity diagrams presented in Section 25.6 and the communication and sequence diagrams presented in Section 25.8. Note that our ATM design does not specify all the program logic and may not specify all the attributes and operations required to complete the ATM implementation. This is a normal part of the object-oriented design process. As we implement the system, we complete the program logic and add attributes and behaviors as necessary to construct the ATM system specified by the requirements specification in Section 25.3.

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

26.4.1 Class ATM

Class `ATM` (Figs. 26.14–26.15) represents the ATM as a whole. Figure 26.14 contains the `ATM` class definition, enclosed in `#ifndef`, `#define` and `#endif` preprocessor directives to ensure that this definition gets included only once in a program. We discuss lines 6–11 shortly. Lines 16–17 contain the function prototypes for the class's `public` member functions. The class diagram of Fig. 26.11 does not list any operations for class `ATM`, but we now declare a `public` member function `run` (line 17) in class `ATM` that allows an external client of the class (i.e., `ATMCaseStudy.cpp`) to tell the `ATM` to run. We also include a function prototype for a default constructor (line 16), which we discuss shortly.

```
1  // ATM.h
2  // ATM class definition. Represents an automated teller machine.
3  #ifndef ATM_H
4  #define ATM_H
5
6  #include "Screen.h" // Screen class definition
7  #include "Keypad.h" // Keypad class definition
8  #include "CashDispenser.h" // CashDispenser class definition
9  #include "DepositSlot.h" // DepositSlot class definition
10 #include "BankDatabase.h" // BankDatabase class definition
11 class Transaction; // forward declaration of class Transaction
12
13 class ATM
14 {
15 public:
16    ATM(); // constructor initializes data members
17    void run(); // start the ATM
18 private:
19    bool userAuthenticated; // whether user is authenticated
20    int currentAccountNumber; // current user's account number
21    Screen screen; // ATM's screen
22    Keypad keypad; // ATM's keypad
23    CashDispenser cashDispenser; // ATM's cash dispenser
24    DepositSlot depositSlot; // ATM's deposit slot
25    BankDatabase bankDatabase; // account information database
26
27    // private utility functions
28    void authenticateUser(); // attempts to authenticate user
29    void performTransactions(); // performs transactions
30    int displayMainMenu() const; // displays main menu
31
32    // return object of specified Transaction derived class
33    Transaction *createTransaction(int);
34 }; // end class ATM
35
36 #endif // ATM_H
```

Fig. 26.14 | Definition of class `ATM`, which represents the ATM.

Lines 19–25 of Fig. 26.14 implement the class's attributes as `private` data members. We determine all but one of these attributes from the class diagrams of Figs. 26.10– 26.11.

We implement the UML `Boolean` attribute `userAuthenticated` in Fig. 26.11 as a `bool` data member in C++ (line 19). Line 20 declares a data member not found in our UML design—an `int` data member `currentAccountNumber` that keeps track of the account number of the current authenticated user. We'll soon see how the class uses this data member.

Lines 21–24 create objects to represent the parts of the ATM. Recall from the class diagram of Fig. 26.10 that class `ATM` has composition relationships with classes `Screen`, `Keypad`, `CashDispenser` and `DepositSlot`, so class `ATM` is responsible for their creation. Line 25 creates a `BankDatabase`, with which the `ATM` interacts to access and manipulate bank account information. [*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.] Lines 6–10 `#include` the class definitions of `Screen`, `Keypad`, `CashDispenser`, `DepositSlot` and `BankDatabase` so that the `ATM` can store objects of these classes.

Lines 28–30 and 33 contain function prototypes for `private` utility functions that the class uses to perform its tasks. We'll see how these functions serve the class shortly. Member function `createTransaction` (line 33) returns a `Transaction` pointer. To include the class name `Transaction` in this file, we must at least include a forward declaration of class `Transaction` (line 11). Recall that a forward declaration tells the compiler that a class exists, but that the class is defined elsewhere. A forward declaration is sufficient here, as we are using a `Transaction` pointer as a return type—if we were creating or returning an actual `Transaction` object, we would need to `#include` the full `Transaction` header file.

`ATM` Class Member-Function Definitions

Figure 26.15 contains the member-function definitions for class `ATM`. Lines 3–7 `#include` the header files required by the implementation file `ATM.cpp`. Including the `ATM` header file allows the compiler to ensure that the class's member functions are defined correctly. This also allows the member functions to use the class's data members.

```
1  // ATM.cpp
2  // Member-function definitions for class ATM.
3  #include "ATM.h" // ATM class definition
4  #include "Transaction.h" // Transaction class definition
5  #include "BalanceInquiry.h" // BalanceInquiry class definition
6  #include "Withdrawal.h" // Withdrawal class definition
7  #include "Deposit.h" // Deposit class definition
8
9  // enumeration constants represent main menu options
10 enum MenuOption { BALANCE_INQUIRY = 1, WITHDRAWAL, DEPOSIT, EXIT };
11
12 // ATM default constructor initializes data members
13 ATM::ATM()
14    : userAuthenticated (false), // user is not authenticated to start
15      currentAccountNumber(0) // no current account number to start
16 {
```

Fig. 26.15 | `ATM` class member-function definitions. (Part 1 of 4.)

```
17      // empty body
18   } // end ATM default constructor
19
20   // start ATM
21   void ATM::run()
22   {
23      // welcome and authenticate user; perform transactions
24      while (true)
25      {
26         // loop while user is not yet authenticated
27         while (!userAuthenticated)
28         {
29            screen.displayMessageLine("\nWelcome!");
30            authenticateUser(); // authenticate user
31         } // end while
32
33         performTransactions(); // user is now authenticated
34         userAuthenticated = false; // reset before next ATM session
35         currentAccountNumber = 0; // reset before next ATM session
36         screen.displayMessageLine("\nThank you! Goodbye!");
37      } // end while
38   } // end function run
39
40   // attempt to authenticate user against database
41   void ATM::authenticateUser()
42   {
43      screen.displayMessage("\nPlease enter your account number: ");
44      int accountNumber = keypad.getInput(); // input account number
45      screen.displayMessage("\nEnter your PIN: "); // prompt for PIN
46      int pin = keypad.getInput(); // input PIN
47
48      // set userAuthenticated to bool value returned by database
49      userAuthenticated =
50         bankDatabase.authenticateUser(accountNumber, pin);
51
52      // check whether authentication succeeded
53      if (userAuthenticated)
54      {
55         currentAccountNumber = accountNumber; // save user's account #
56      } // end if
57      else
58         screen.displayMessageLine(
59            "Invalid account number or PIN. Please try again.");
60   } // end function authenticateUser
61
62   // display the main menu and perform transactions
63   void ATM::performTransactions()
64   {
65      // local pointer to store transaction currently being processed
66      Transaction *currentTransactionPtr;
67
68      bool userExited = false; // user has not chosen to exit
```

Fig. 26.15 | ATM class member-function definitions. (Part 2 of 4.)

```
69
70      // loop while user has not chosen option to exit system
71      while (!userExited)
72      {
73         // show main menu and get user selection
74         int mainMenuSelection = displayMainMenu();
75
76         // decide how to proceed based on user's menu selection
77         switch (mainMenuSelection)
78         {
79            // user chose to perform one of three transaction types
80            case BALANCE_INQUIRY:
81            case WITHDRAWAL:
82            case DEPOSIT:
83               // initialize as new object of chosen type
84               currentTransactionPtr =
85                  createTransaction(mainMenuSelection);
86
87               currentTransactionPtr->execute(); // execute transaction
88
89               // free the space for the dynamically allocated Transaction
90               delete currentTransactionPtr;
91
92               break;
93            case EXIT: // user chose to terminate session
94               screen.displayMessageLine("\nExiting the system...");
95               userExited = true; // this ATM session should end
96               break;
97            default: // user did not enter an integer from 1-4
98               screen.displayMessageLine(
99                  "\nYou did not enter a valid selection. Try again.");
100              break;
101        } // end switch
102     } // end while
103  } // end function performTransactions
104
105  // display the main menu and return an input selection
106  int ATM::displayMainMenu() const
107  {
108     screen.displayMessageLine("\nMain menu:");
109     screen.displayMessageLine("1 - View my balance");
110     screen.displayMessageLine("2 - Withdraw cash");
111     screen.displayMessageLine("3 - Deposit funds");
112     screen.displayMessageLine("4 - Exit\n");
113     screen.displayMessage("Enter a choice: ");
114     return keypad.getInput(); // return user's selection
115  } // end function displayMainMenu
116
117  // return object of specified Transaction derived class
118  Transaction *ATM::createTransaction(int type)
119  {
120     Transaction *tempPtr; // temporary Transaction pointer
121
```

Fig. 26.15 | ATM class member-function definitions. (Part 3 of 4.)

```
122     // determine which type of Transaction to create
123     switch (type)
124     {
125        case BALANCE_INQUIRY: // create new BalanceInquiry transaction
126           tempPtr = new BalanceInquiry(
127              currentAccountNumber, screen, bankDatabase);
128           break;
129        case WITHDRAWAL: // create new Withdrawal transaction
130           tempPtr = new Withdrawal(currentAccountNumber, screen,
131              bankDatabase, keypad, cashDispenser);
132           break;
133        case DEPOSIT: // create new Deposit transaction
134           tempPtr = new Deposit(currentAccountNumber, screen,
135              bankDatabase, keypad, depositSlot);
136           break;
137     } // end switch
138
139     return tempPtr; // return the newly created object
140  } // end function createTransaction
```

Fig. 26.15 | `ATM` class member-function definitions. (Part 4 of 4.)

Line 10 declares an `enum` named `MenuOption` that contains constants corresponding to the four options in the ATM's main menu (i.e., balance inquiry, withdrawal, deposit and exit). Note that setting `BALANCE_INQUIRY` to `1` causes the subsequent enumeration constants to be assigned the values `2`, `3` and `4`, as enumeration constant values increment by `1`.

Lines 13–18 define class `ATM`'s constructor, which initializes the class's data members. When an `ATM` object is first created, no user is authenticated, so line 14 uses a member initializer to set `userAuthenticated` to `false`. Likewise, line 15 initializes `currentAccountNumber` to `0` because there is no current user yet.

ATM Member Function `run`

`ATM` member function `run` (lines 21–38) uses an infinite loop (lines 24–37) to repeatedly welcome a user, attempt to authenticate the user and, if authentication succeeds, allow the user to perform transactions. After an authenticated user performs the desired transactions and chooses to exit, the ATM resets itself, displays a goodbye message to the user and restarts the process. We use an infinite loop here to simulate the fact that an ATM appears to run continuously until the bank turns it off (an action beyond the user's control). An ATM user has the option to exit the system, but does not have the ability to turn off the ATM completely.

Authenticating a User

Inside member function `run`'s infinite loop, lines 27–31 cause the ATM to repeatedly welcome and attempt to authenticate the user as long as the user has not been authenticated (i.e., `!userAuthenticated` is `true`). Line 29 invokes member function `displayMessageLine` of the `ATM`'s `screen` to display a welcome message. Like `Screen` member function `displayMessage` designed in the case study, member function `displayMessageLine` (declared in line 13 of Fig. 26.16 and defined in lines 20–23 of Fig. 26.17) displays a message to the user, but

this member function also outputs a newline after displaying the message. We've added this member function during implementation to give class `Screen`'s clients more control over the placement of displayed messages. Line 30 of Fig. 26.15 invokes class `ATM`'s `private` utility function `authenticateUser` (lines 41–60) to attempt to authenticate the user.

We refer to the requirements specification to determine the steps necessary to authenticate the user before allowing transactions to occur. Line 43 of member function `authenticateUser` invokes member function `displayMessage` of the `ATM`'s `screen` to prompt the user to enter an account number. Line 44 invokes member function `getInput` of the `ATM`'s `keypad` to obtain the user's input, then stores the integer value entered by the user in a local variable `accountNumber`. Member function `authenticateUser` next prompts the user to enter a PIN (line 45), and stores the PIN input by the user in a local variable `pin` (line 46). Next, lines 49–50 attempt to authenticate the user by passing the `accountNumber` and `pin` entered by the user to the `bankDatabase`'s `authenticateUser` member function. Class `ATM` sets its `userAuthenticated` data member to the `bool` value returned by this function—`userAuthenticated` becomes `true` if authentication succeeds (i.e., `accountNumber` and `pin` match those of an existing `Account` in `bankDatabase`) and remains `false` otherwise. If `userAuthenticated` is `true`, line 55 saves the account number entered by the user (i.e., `accountNumber`) in the `ATM` data member `currentAccountNumber`. The other member functions of class `ATM` use this variable whenever an ATM session requires access to the user's account number. If `userAuthenticated` is `false`, lines 58–59 use the `screen`'s `displayMessageLine` member function to indicate that an invalid account number and/or PIN was entered and the user must try again. Note that we set `currentAccountNumber` only after authenticating the user's account number and the associated PIN—if the database could not authenticate the user, `currentAccountNumber` remains `0`.

After member function `run` attempts to authenticate the user (line 30), if `userAuthenticated` is still `false`, the `while` loop in lines 27–31 executes again. If `userAuthenticated` is now `true`, the loop terminates and control continues with line 33, which calls class `ATM`'s utility function `performTransactions`.

Performing Transactions

Member function `performTransactions` (lines 63–103) carries out an ATM session for an authenticated user. Line 66 declares a local `Transaction` pointer, which we aim at a `BalanceInquiry`, `Withdrawal` or `Deposit` object representing the ATM transaction currently being processed. We use a `Transaction` pointer here to allow us to take advantage of polymorphism. Also, we use the role name included in the class diagram of Fig. 25.7—`currentTransaction`—in naming this pointer. As per our pointer-naming convention, we append "`Ptr`" to the role name to form the variable name `currentTransactionPtr`. Line 68 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 71–102) that allows the user to execute an unlimited number of transactions before choosing to exit. Within this loop, line 74 displays the main menu and obtains the user's menu selection by calling an `ATM` utility function `displayMainMenu` (defined in lines 106–115). This member function displays the main menu by invoking member functions of the `ATM`'s `screen` and returns a menu selection obtained from the user through the `ATM`'s `keypad`. Note that this member function is `const` because it does not modify the contents of the object. Line 74 stores the user's selection returned by `displayMainMenu` in local variable `mainMenuSelection`.

After obtaining a main menu selection, member function `performTransactions` uses a `switch` statement (lines 77–101) to respond to the selection appropriately. If `mainMenuSelection` is equal to any of the three enumeration constants representing transaction types (i.e., if the user chose to perform a transaction), lines 84–85 call utility function `createTransaction` (defined in lines 118–140) to return a pointer to a newly instantiated object of the type that corresponds to the selected transaction. Pointer `currentTransactionPtr` is assigned the pointer returned by `createTransaction`. Line 87 then uses `currentTransactionPtr` to invoke the new object's `execute` member function to execute the transaction. We'll discuss `Transaction` member function `execute` and the three `Transaction` derived classes shortly. Finally, when the `Transaction` derived class object is no longer needed, line 90 releases the memory dynamically allocated for it.

We aim the `Transaction` pointer `currentTransactionPtr` at an object of one of the three `Transaction` derived classes so that we can execute transactions *polymorphically*. For example, if the user chooses to perform a balance inquiry, `mainMenuSelection` equals `BALANCE_INQUIRY`, leading `createTransaction` to return a pointer to a `BalanceInquiry` object. Thus, `currentTransactionPtr` points to a `BalanceInquiry`, and invoking `currentTransactionPtr->execute()` results in `BalanceInquiry`'s version of `execute` being called.

Creating a Transaction

Member function `createTransaction` (lines 118–140) uses a `switch` statement (lines 123–137) to instantiate a new `Transaction` derived class object of the type indicated by the parameter `type`. Recall that member function `performTransactions` passes `mainMenuSelection` to this member function only when `mainMenuSelection` contains a value corresponding to one of the three transaction types. Therefore `type` equals either `BALANCE_INQUIRY`, `WITHDRAWAL` or `DEPOSIT`. Each `case` in the `switch` statement aims the temporary pointer `tempPtr` at a newly created object of the appropriate `Transaction` derived class. Each constructor has a unique parameter list, based on the specific data required to initialize the derived class object. A `BalanceInquiry` requires only the account number of the current user and references to the `ATM`'s `screen` and the `bankDatabase`. In addition to these parameters, a `Withdrawal` requires references to the `ATM`'s `keypad` and `cashDispenser`, and a `Deposit` requires references to the `ATM`'s `keypad` and `depositSlot`. As you'll soon see, the `BalanceInquiry`, `Withdrawal` and `Deposit` constructors each specify reference parameters to receive the objects representing the required parts of the ATM. Thus, when member function `createTransaction` passes objects in the `ATM` (e.g., `screen` and `keypad`) to the initializer for each newly created `Transaction` derived class object, the new object actually receives *references* to the `ATM`'s composite objects. We discuss the transaction classes in more detail in Sections 26.4.8–26.4.11.

Exiting the Main Menu and Processing Invalid Selections

After executing a transaction (line 87 in `performTransactions`), `userExited` remains `false` and the `while` loop in lines 71–102 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 95 sets `userExited` to `true`, causing the condition of the `while` loop (`!userExited`) to become `false`. This `while` is the final statement of member function `performTransactions`, so control returns to the calling function `run`. If the user enters an invalid main menu selection (i.e., not an integer from 1–4), lines 98–99 display an appropriate error message, `userExited` remains `false` and the user returns to the main menu to try again.

Awaiting the Next ATM User

When `performTransactions` returns control to member function `run`, the user has chosen to exit the system, so lines 34–35 reset the `ATM`'s data members `userAuthenticated` and `currentAccountNumber` to prepare for the next ATM user. Line 36 displays a goodbye message before the ATM starts over and welcomes the next user.

26.4.2 Class Screen

Class `Screen` (Figs. 26.16–26.17) represents the screen of the ATM and encapsulates all aspects of displaying output to the user. Class `Screen` approximates a real ATM's screen with a computer monitor and outputs text messages using `cout` and the stream insertion operator (<<). In this case study, we designed class `Screen` to have one operation—`displayMessage`. For greater flexibility in displaying messages to the `Screen`, we now declare three `Screen` member functions—`displayMessage`, `displayMessageLine` and `displayDollarAmount`. The prototypes for these member functions appear in lines 12–14 of Fig. 26.16.

```
1  // Screen.h
2  // Screen class definition. Represents the screen of the ATM.
3  #ifndef SCREEN_H
4  #define SCREEN_H
5
6  #include <string>
7  using namespace std;
8
9  class Screen
10 {
11 public:
12    void displayMessage(string) const; // output a message
13    void displayMessageLine(string) const; // output message with newline
14    void displayDollarAmount(double) const; // output a dollar amount
15 }; // end class Screen
16
17 #endif // SCREEN_H
```

Fig. 26.16 | `Screen` class definition.

```
1  // Screen.cpp
2  // Member-function definitions for class Screen.
3  #include <iostream>
4  #include <iomanip>
5  #include "Screen.h" // Screen class definition
6  using namespace std;
7
8  // output a message without a newline
9  void Screen::displayMessage(string message) const
10 {
11    cout << message;
12 } // end function displayMessage
```

Fig. 26.17 | `Screen` class member-function definitions. (Part 1 of 2.)

```
13
14  // output a message with a newline
15  void Screen::displayMessageLine(string message) const
16  {
17     cout << message << endl;
18  } // end function displayMessageLine
19
20  // output a dollar amount
21  void Screen::displayDollarAmount(double amount) const
22  {
23     cout << fixed << setprecision(2) << "$" << amount;
24  } // end function displayDollarAmount
```

Fig. 26.17 | `Screen` class member-function definitions. (Part 2 of 2.)

`Screen` Class Member-Function Definitions

Figure 26.17 contains the member-function definitions for class `Screen`. Line 5 `#includes` the `Screen` class definition. Member function `displayMessage` (lines 9–12) takes a `string` as an argument and prints it to the console using `cout` and the stream insertion operator (`<<`). The cursor stays on the same line, making this member function appropriate for displaying prompts to the user. Member function `displayMessageLine` (lines 15–18) also prints a `string`, but outputs a newline to move the cursor to the next line. Finally, member function `displayDollarAmount` (lines 21–24) outputs a properly formatted dollar amount (e.g., `$123.45`). Line 23 uses stream manipulators `fixed` and `setprecision` to output a value formatted with two decimal places.

26.4.3 Class Keypad

Class `Keypad` (Figs. 26.18–26.19) 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. A computer keyboard contains many keys not found on the ATM's keypad. However, we assume that the user presses only the keys on the computer keyboard that also appear on the keypad—the keys numbered 0–9 and the *Enter* key. Line 9 of Fig. 26.18 contains the function prototype for class `Keypad`'s one member function `getInput`. This member function is declared `const` because it does not change the object.

```
1  // Keypad.h
2  // Keypad class definition. Represents the keypad of the ATM.
3  #ifndef KEYPAD_H
4  #define KEYPAD_H
5
6  class Keypad
7  {
8  public:
9     int getInput() const; // return an integer value entered by user
10 }; // end class Keypad
11
12 #endif // KEYPAD_H
```

Fig. 26.18 | `Keypad` class definition.

Keypad Class Member-Function Definition

In the `Keypad` implementation file (Fig. 26.19), member function `getInput` (defined in lines 9–14) uses the standard input stream `cin` and the stream extraction operator (`>>`) to obtain input from the user. Line 11 declares a local variable to store the user's input. Line 12 reads input into local variable `input`, then line 13 returns this value. Recall that `getInput` obtains all the input used by the ATM. `Keypad`'s `getInput` member function 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. [*Note:* Using the standard input stream `cin` and the stream extraction operator (`>>`) allows noninteger input to be read from the user. Because the real ATM's keypad permits only integer input, however, we assume that the user enters an integer and do not attempt to fix problems caused by noninteger input.]

```
1  // Keypad.cpp
2  // Member-function definition for class Keypad (the ATM's keypad).
3  #include <iostream>
4  using namespace std;
5
6  #include "Keypad.h" // Keypad class definition
7
8  // return an integer value entered by user
9  int Keypad::getInput() const
10 {
11    int input; // variable to store the input
12    cin >> input; // we assume that user enters an integer
13    return input; // return the value entered by user
14 } // end function getInput
```

Fig. 26.19 | `Keypad` class member-function definition.

26.4.4 Class CashDispenser

Class `CashDispenser` (Figs. 26.20–26.21) represents the cash dispenser. Figure 26.20 contains the function prototype for a default constructor (line 9). Class `CashDispenser` declares two additional `public` member functions—`dispenseCash` (line 12) and `isSufficientCashAvailable` (line 15). The class trusts that a client (i.e., `Withdrawal`) calls `dispenseCash` only after establishing that sufficient cash is available by calling `isSufficientCashAvailable`. Thus, `dispenseCash` simply simulates dispensing the requested amount without checking whether sufficient cash is available. Line 17 declares `private` constant `INITIAL_COUNT`, which indicates the initial count of bills in the cash dispenser when the ATM starts (i.e., `500`). Line 18 implements attribute `count` (modeled in Fig. 26.11), which keeps track of the number of bills remaining in the `CashDispenser` at any time.

```
1  // CashDispenser.h
2  // CashDispenser class definition. Represents the ATM's cash dispenser.
3  #ifndef CASH_DISPENSER_H
4  #define CASH_DISPENSER_H
```

Fig. 26.20 | `CashDispenser` class definition. (Part 1 of 2.)

```
5
6  class CashDispenser
7  {
8  public:
9     CashDispenser(); // constructor initializes bill count to 500
10
11    // simulates dispensing of specified amount of cash
12    void dispenseCash(int);
13
14    // indicates whether cash dispenser can dispense desired amount
15    bool isSufficientCashAvailable(int) const;
16 private:
17    static const int INITIAL_COUNT = 500;
18    int count; // number of $20 bills remaining
19 }; // end class CashDispenser
20
21 #endif // CASH_DISPENSER_H
```

Fig. 26.20 | `CashDispenser` class definition. (Part 2 of 2.)

`CashDispenser` Class Member-Function Definitions

Figure 26.21 contains the definitions of class `CashDispenser`'s member functions. The constructor (lines 6–9) sets `count` to the initial count (i.e., `500`). Member function `dispenseCash` (lines 13–17) simulates cash dispensing. If our system were hooked up to a real hardware cash dispenser, this member function would interact with the hardware device to physically dispense cash. Our simulated version of the member function simply decreases the `count` of bills remaining by the number required to dispense the specified `amount` (line 16). Line 15 calculates the number of $20 bills required to dispense the specified `amount`. The ATM allows the user to choose only withdrawal amounts that are multiples of $20, so we divide `amount` by `20` to obtain the number of `billsRequired`. Also, it's the responsibility of the class's client (i.e., `Withdrawal`) to inform the user that cash has been dispensed—`CashDispenser` cannot interact directly with `Screen`.

```
22 // CashDispenser.cpp
23 // Member-function definitions for class CashDispenser.
24 #include "CashDispenser.h" // CashDispenser class definition
25
26 // CashDispenser default constructor initializes count to default
27 CashDispenser::CashDispenser()
28 {
29    count = INITIAL_COUNT; // set count attribute to default
30 } // end CashDispenser default constructor
31
32 // simulates dispensing of specified amount of cash; assumes enough cash
33 // is available (previous call to isSufficientCashAvailable returned true)
34 void CashDispenser::dispenseCash(int amount)
35 {
36    int billsRequired = amount / 20; // number of $20 bills required
37    count -= billsRequired; // update the count of bills
38 } // end function dispenseCash
```

Fig. 26.21 | `CashDispenser` class member-function definitions. (Part 1 of 2.)

```
39
40 // indicates whether cash dispenser can dispense desired amount
41 bool CashDispenser::isSufficientCashAvailable(int amount) const
42 {
43    int billsRequired = amount / 20; // number of $20 bills required
44
45    if (count >= billsRequired)
46       return true; // enough bills are available
47    else
48       return false; // not enough bills are available
49 } // end function isSufficientCashAvailable
```

Fig. 26.21 | `CashDispenser` class member-function definitions. (Part 2 of 2.)

Member function `isSufficientCashAvailable` (lines 20–28) has a parameter `amount` that specifies the amount of cash in question. Lines 24–27 return `true` if the `CashDispenser`'s `count` is greater than or equal to `billsRequired` (i.e., enough bills are available) and `false` otherwise (i.e., not enough bills). For example, if a user wishes to withdraw $80 (i.e., `billsRequired` is 4), but only three bills remain (i.e., `count` is 3), the member function returns `false`.

26.4.5 Class `DepositSlot`

Class `DepositSlot` (Figs. 26.22–26.23) represents the deposit slot of the ATM. Like the version of class `CashDispenser` presented here, this version of class `DepositSlot` merely simulates the functionality of a real hardware deposit slot. `DepositSlot` has no data members and only one member function—`isEnvelopeReceived` (declared in line 9 of Fig. 26.22 and defined in lines 7–10 of Fig. 26.23)—that indicates whether a deposit envelope was received.

```
1  // DepositSlot.h
2  // DepositSlot class definition. Represents the ATM's deposit slot.
3  #ifndef DEPOSIT_SLOT_H
4  #define DEPOSIT_SLOT_H
5
6  class DepositSlot
7  {
8  public:
9     bool isEnvelopeReceived() const; // tells whether envelope was received
10 }; // end class DepositSlot
11
12 #endif // DEPOSIT_SLOT_H
```

Fig. 26.22 | `DepositSlot` class definition.

```
1 // DepositSlot.cpp
2 // Member-function definition for class DepositSlot.
3 #include "DepositSlot.h" // DepositSlot class definiton
```

Fig. 26.23 | `DepositSlot` class member-function definition. (Part 1 of 2.)

```
4
5  // indicates whether envelope was received (always returns true,
6  // because this is only a software simulation of a real deposit slot)
7  bool DepositSlot::isEnvelopeReceived() const
8  {
9     return true; // deposit envelope was received
10 } // end function isEnvelopeReceived
```

Fig. 26.23 | `DepositSlot` class member-function definition. (Part 2 of 2.)

Recall from the requirements specification that the ATM allows the user up to two minutes to insert an envelope. The current version of member function `isEnvelopeReceived` simply returns `true` immediately (line 9 of Fig. 26.23), because this is only a software simulation, and we assume that the user has inserted an envelope within the required time frame. If an actual hardware deposit slot were connected to our system, member function `isEnvelopeReceived` might be implemented to wait for a maximum of two minutes to receive a signal from the hardware deposit slot indicating that the user has indeed inserted a deposit envelope. If `isEnvelopeReceived` were to receive such a signal within two minutes, the member function would return `true`. If two minutes elapsed and the member function still had not received a signal, then the member function would return `false`.

26.4.6 Class Account

Class `Account` (Figs. 26.24–26.25) represents a bank account. Lines 9–15 in the class definition (Fig. 26.24) contain function prototypes for the class's constructor and six member functions, which we discuss shortly. Each `Account` has four attributes (modeled in Fig. 26.11)—`accountNumber`, `pin`, `availableBalance` and `totalBalance`. Lines 17–20 implement these attributes as `private` data members. Data member `availableBalance` represents the amount of funds available for withdrawal. Data member `totalBalance` represents the amount of funds available, plus the amount of deposited funds still pending confirmation or clearance.

```
1  // Account.h
2  // Account class definition. Represents a bank account.
3  #ifndef ACCOUNT_H
4  #define ACCOUNT_H
5
6  class Account
7  {
8  public:
9     Account(int, int, double, double); // constructor sets attributes
10    bool validatePIN(int) const; // is user-specified PIN correct?
11    double getAvailableBalance() const; // returns available balance
12    double getTotalBalance() const; // returns total balance
13    void credit(double); // adds an amount to the Account balance
14    void debit(double); // subtracts an amount from the Account balance
15    int getAccountNumber() const; // returns account number
```

Fig. 26.24 | `Account` class definition. (Part 1 of 2.)

```
16   private:
17      int accountNumber; // account number
18      int pin; // PIN for authentication
19      double availableBalance; // funds available for withdrawal
20      double totalBalance; // funds available + funds waiting to clear
21   }; // end class Account
22
23   #endif // ACCOUNT_H
```

Fig. 26.24 | `Account` class definition. (Part 2 of 2.)

`Account` Class Member-Function Definitions

Figure 26.25 presents the definitions of class `Account`'s member functions. The class's constructor (lines 6–14) takes an account number, the PIN established for the account, the initial available balance and the initial total balance as arguments. Lines 8–11 assign these values to the class's data members using member initializers.

Member function `validatePIN` (lines 17–23) determines whether a user-specified PIN (i.e., parameter `userPIN`) matches the PIN associated with the account (i.e., data member `pin`). Recall that we modeled this member function's parameter `userPIN` in the UML class diagram of Fig. 25.19. If the two PINs match, the member function returns `true` (line 20); otherwise, it returns `false` (line 22).

Member functions `getAvailableBalance` (lines 26–29) and `getTotalBalance` (lines 32–35) are *get* functions that return the values of `double` data members `availableBalance` and `totalBalance`, respectively.

Member function `credit` (lines 38–41) adds an amount of money (i.e., parameter `amount`) to an `Account` as part of a deposit transaction. Note that this member function adds the `amount` only to data member `totalBalance` (line 40). The money credited to an account during a deposit does not become available immediately, so we modify only the total balance. We assume that the bank updates the available balance appropriately at a later time. Our implementation of class `Account` includes only member functions required for carrying out ATM transactions. Therefore, we omit the member functions that some other bank system would invoke to add to data member `availableBalance` (to confirm a deposit) or subtract from data member `totalBalance` (to reject a deposit).

```
1   // Account.cpp
2   // Member-function definitions for class Account.
3   #include "Account.h" // Account class definition
4
5   // Account constructor initializes attributes
6   Account::Account(int theAccountNumber, int thePIN,
7      double theAvailableBalance, double theTotalBalance)
8      : accountNumber(theAccountNumber),
9        pin(thePIN),
10       availableBalance(theAvailableBalance),
11       totalBalance(theTotalBalance)
12  {
13     // empty body
14  } // end Account constructor
```

Fig. 26.25 | `Account` class member-function definitions. (Part 1 of 2.)

```
15
16  // determines whether a user-specified PIN matches PIN in Account
17  bool Account::validatePIN(int userPIN) const
18  {
19     if (userPIN == pin)
20        return true;
21     else
22        return false;
23  } // end function validatePIN
24
25  // returns available balance
26  double Account::getAvailableBalance() const
27  {
28     return availableBalance;
29  } // end function getAvailableBalance
30
31  // returns the total balance
32  double Account::getTotalBalance() const
33  {
34     return totalBalance;
35  } // end function getTotalBalance
36
37  // credits an amount to the account
38  void Account::credit(double amount)
39  {
40     totalBalance += amount; // add to total balance
41  } // end function credit
42
43  // debits an amount from the account
44  void Account::debit(double amount)
45  {
46     availableBalance -= amount; // subtract from available balance
47     totalBalance -= amount; // subtract from total balance
48  } // end function debit
49
50  // returns account number
51  int Account::getAccountNumber() const
52  {
53     return accountNumber;
54  } // end function getAccountNumber
```

Fig. 26.25 | `Account` class member-function definitions. (Part 2 of 2.)

Member function `debit` (lines 44–48) subtracts an amount of money (i.e., parameter `amount`) from an `Account` as part of a withdrawal transaction. This member function subtracts the `amount` from both data member `availableBalance` (line 46) and data member `totalBalance` (line 47), because a withdrawal affects both measures of an account balance.

Member function `getAccountNumber` (lines 51–54) provides access to an `Account`'s `accountNumber`. We include this member function in our implementation so that a client of the class (i.e., `BankDatabase`) can identify a particular `Account`. For example, `BankDatabase` contains many `Account` objects, and it can invoke this member function on each of its `Account` objects to locate the one with a specific account number.

26.4.7 Class BankDatabase

Class `BankDatabase` (Figs. 26.26–26.27) models the bank's database with which the ATM interacts to access and modify a user's account information. The class definition (Fig. 26.26) declares function prototypes for the class's constructor and several member functions. We discuss these momentarily. The class definition also declares the `BankDatabase`'s data members. We determine one data member for class `BankDatabase` based on its composition relationship with class `Account`. Recall from Fig. 26.10 that a `BankDatabase` is composed of zero or more objects of class `Account`. Line 24 of Fig. 26.26 implements data member `accounts`—a `vector` of `Account` objects—to implement this composition relationship. Lines 6–7 allow us to use `vector` in this file. Line 27 contains the function prototype for a `private` utility function `getAccount` that allows the member functions of the class to obtain a pointer to a specific `Account` in the `accounts` `vector`.

```
1  // BankDatabase.h
2  // BankDatabase class definition. Represents the bank's database.
3  #ifndef BANK_DATABASE_H
4  #define BANK_DATABASE_H
5
6  #include <vector> // class uses vector to store Account objects
7  using namespace std;
8
9  #include "Account.h" // Account class definition
10
11 class BankDatabase
12 {
13 public:
14    BankDatabase(); // constructor initializes accounts
15
16    // determine whether account number and PIN match those of an Account
17    bool authenticateUser(int, int); // returns true if Account authentic
18
19    double getAvailableBalance(int); // get an available balance
20    double getTotalBalance(int); // get an Account's total balance
21    void credit(int, double); // add amount to Account balance
22    void debit(int, double); // subtract amount from Account balance
23 private:
24    vector< Account > accounts; // vector of the bank's Accounts
25
26    // private utility function
27    Account * getAccount(int); // get pointer to Account object
28 }; // end class BankDatabase
29
30 #endif // BANK_DATABASE_H
```

Fig. 26.26 | `BankDatabase` class definition.

`BankDatabase` Class Member-Function Definitions

Figure 26.27 contains the member-function definitions for class `BankDatabase`. We implement the class with a default constructor (lines 6–15) that adds `Account` objects to data member `accounts`. For the sake of testing the system, we create two new `Account` objects

with test data (lines 9–10), then add them to the end of the `vector` (lines 13–14). The `Account` constructor has four parameters—the account number, the PIN assigned to the account, the initial available balance and the initial total balance.

```
1  // BankDatabase.cpp
2  // Member-function definitions for class BankDatabase.
3  #include "BankDatabase.h" // BankDatabase class definition
4
5  // BankDatabase default constructor initializes accounts
6  BankDatabase::BankDatabase()
7  {
8     // create two Account objects for testing
9     Account account1(12345, 54321, 1000.0, 1200.0);
10    Account account2(98765, 56789, 200.0, 200.0);
11
12    // add the Account objects to the vector accounts
13    accounts.push_back(account1); // add account1 to end of vector
14    accounts.push_back(account2); // add account2 to end of vector
15 } // end BankDatabase default constructor
16
17 // retrieve Account object containing specified account number
18 Account * BankDatabase::getAccount(int accountNumber)
19 {
20    // loop through accounts searching for matching account number
21    for (size_t i = 0; i < accounts.size(); i++)
22    {
23       // return current account if match found
24       if (accounts[ i ].getAccountNumber() == accountNumber)
25          return &accounts[ i ];
26    } // end for
27
28    return NULL; // if no matching account was found, return NULL
29 } // end function getAccount
30
31 // determine whether user-specified account number and PIN match
32 // those of an account in the database
33 bool BankDatabase::authenticateUser(int userAccountNumber,
34    int userPIN)
35 {
36    // attempt to retrieve the account with the account number
37    Account * const userAccountPtr = getAccount(userAccountNumber);
38
39    // if account exists, return result of Account function validatePIN
40    if (userAccountPtr != NULL)
41       return userAccountPtr->validatePIN(userPIN);
42    else
43       return false; // account number not found, so return false
44 } // end function authenticateUser
45
46 // return available balance of Account with specified account number
47 double BankDatabase::getAvailableBalance(int userAccountNumber)
48 {
```

Fig. 26.27 | `BankDatabase` class member-function definitions. (Part 1 of 2.)

```
49      Account * const userAccountPtr = getAccount(userAccountNumber);
50      return userAccountPtr->getAvailableBalance();
51   } // end function getAvailableBalance
52
53   // return total balance of Account with specified account number
54   double BankDatabase::getTotalBalance(int userAccountNumber)
55   {
56      Account * const userAccountPtr = getAccount(userAccountNumber);
57      return userAccountPtr->getTotalBalance();
58   } // end function getTotalBalance
59
60   // credit an amount to Account with specified account number
61   void BankDatabase::credit(int userAccountNumber, double amount)
62   {
63      Account * const userAccountPtr = getAccount(userAccountNumber);
64      userAccountPtr->credit(amount);
65   } // end function credit
66
67   // debit an amount from Account with specified account number
68   void BankDatabase::debit(int userAccountNumber, double amount)
69   {
70      Account * const userAccountPtr = getAccount(userAccountNumber);
71      userAccountPtr->debit(amount);
72   } // end function debit
```

Fig. 26.27 | `BankDatabase` class member-function definitions. (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, the member functions of class `BankDatabase` do nothing more than invoke the corresponding member functions of the `Account` object belonging to the current ATM user.

We include *`private` utility function* `getAccount` (lines 18–29) to allow the `BankDatabase` to obtain a pointer to a particular `Account` within `vector accounts`. To locate the user's `Account`, the `BankDatabase` compares the value returned by member function `getAccountNumber` for each element of `accounts` to a specified account number until it finds a match. Lines 21–26 traverse the `accounts vector`. If the account number of the current `Account` (i.e., `accounts[ i ]`) equals the value of parameter `accountNumber`, the member function immediately returns the address of the current `Account` (i.e., a pointer to the current `Account`). If no account has the given account number, then line 28 returns `NULL`. Note that this member function must return a pointer, as opposed to a reference, because there is the possibility that the return value could be `NULL`—*a reference cannot be `NULL`, but a pointer can.*

Note that `vector` function `size` (invoked in the loop-continuation condition in line 21) returns the number of elements in a `vector` as a value of type `size_t` (which is usually `unsigned int`). As a result, we declare the control variable `i` to be of type `size_t`, too. On some compilers, declaring `i` as an `int` would cause the compiler to issue a warning message, because the loop-continuation condition would compare a `signed` value (i.e., an `int`) and an `unsigned` value (i.e., a value of type `size_t`).

Member function `authenticateUser` (lines 33–44) proves or disproves the an ATM user's identity. This function takes a user-specified account number and user-specified

PIN as arguments and indicates whether they match the account number and PIN of an `Account` in the database. Line 37 calls utility function `getAccount`, which returns either a pointer to an `Account` with `userAccountNumber` as its account number or `NULL` to indicate that `userAccountNumber` is invalid. We declare `userAccountPtr` to be a `const` pointer because, once the member function aims this pointer at the user's `Account`, the pointer should not change. If `getAccount` returns a pointer to an `Account` object, line 41 returns the `bool` value returned by that object's `validatePIN` member function. `BankDatabase`'s `authenticateUser` member function does not perform the PIN comparison itself—rather, it forwards `userPIN` to the `Account` object's `validatePIN` member function to do so. The value returned by `Account` member function `validatePIN` indicates whether the user-specified PIN matches the PIN of the user's `Account`, so member function `authenticateUser` simply returns this value to the client of the class (i.e., `ATM`).

`BankDatabase` trusts the `ATM` to invoke member function `authenticateUser` and receive a return value of `true` before allowing the user to perform transactions. `BankDatabase` also trusts that each `Transaction` object created by the `ATM` contains the valid account number of the current authenticated user and that this is the account number passed to the remaining `BankDatabase` member functions as argument `userAccountNumber`. Member functions `getAvailableBalance` (lines 47–51), `getTotalBalance` (lines 54–58), `credit` (lines 61–65) and `debit` (lines 68–72) therefore simply retrieve a pointer to the user's `Account` object with utility function `getAccount`, then use this pointer to invoke the appropriate `Account` member function on the user's `Account` object. We know that the calls to `getAccount` within these member functions will never return `NULL`, because `userAccountNumber` must refer to an existing `Account`. Note that `getAvailableBalance` and `getTotalBalance` return the values returned by the corresponding `Account` member functions. Also, `credit` and `debit` simply redirect parameter `amount` to the `Account` member functions they invoke.

26.4.8 Class `Transaction`

Class `Transaction` (Figs. 26.28–26.29) 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`. Figure 26.28 expands upon the `Transaction` header file first developed in Section 26.3. Lines 13, 17–19 and 22 contain function prototypes for the class's constructor and four member functions, which we discuss shortly. Line 15 defines a *`virtual` destructor* with an empty body—this makes all derived-class destructors `virtual` (even those defined implicitly by the compiler) and ensures that dynamically allocated derived-class objects get destroyed properly when they are deleted via a base-class pointer. Lines 24–26 declare the class's `private` data members. Recall from the class diagram of Fig. 26.11 that class `Transaction` contains an attribute `accountNumber` (implemented in line 24) that indicates the account involved in the `Transaction`. We derive data members `screen` (line 25) and `bankDatabase` (line 26) from class `Transaction`'s associations modeled in Fig. 26.10—all transactions require access to the ATM's screen and the bank's database, so we include references to a `Screen` and a `BankDatabase` as data members of class `Transaction`. As you'll soon see, `Transaction`'s constructor initializes these references. The forward declarations in lines 6–7 signify that the header file contains references to objects of classes `Screen` and `BankDatabase`, but that the definitions of these classes lie outside the header file.

```
// Transaction.h
// Transaction abstract base class definition.
#ifndef TRANSACTION_H
#define TRANSACTION_H

class Screen; // forward declaration of class Screen
class BankDatabase; // forward declaration of class BankDatabase

class Transaction
{
public:
   // constructor initializes common features of all Transactions
   Transaction(int, Screen &, BankDatabase &);

   virtual ~Transaction() { } // virtual destructor with empty body

   int getAccountNumber() const; // return account number
   Screen &getScreen() const; // return reference to screen
   BankDatabase &getBankDatabase() const; // return reference to database

   // pure virtual function to perform the transaction
   virtual void execute() = 0; // overridden in derived classes
private:
   int accountNumber; // indicates account involved
   Screen &screen; // reference to the screen of the ATM
   BankDatabase &bankDatabase; // reference to the account info database
}; // end class Transaction

#endif // TRANSACTION_H
```

Fig. 26.28 | `Transaction` class definition.

```
// Transaction.cpp
// Member-function definitions for class Transaction.
#include "Transaction.h" // Transaction class definition
#include "Screen.h" // Screen class definition
#include "BankDatabase.h" // BankDatabase class definition

// constructor initializes common features of all Transactions
Transaction::Transaction(int userAccountNumber, Screen &atmScreen,
   BankDatabase &atmBankDatabase)
   : accountNumber(userAccountNumber),
     screen(atmScreen),
     bankDatabase(atmBankDatabase)
{
   // empty body
} // end Transaction constructor

// return account number
int Transaction::getAccountNumber() const
{
```

Fig. 26.29 | `Transaction` class member-function definitions. (Part 1 of 2.)

```
20      return accountNumber;
21   } // end function getAccountNumber
22
23   // return reference to screen
24   Screen &Transaction::getScreen() const
25   {
26      return screen;
27   } // end function getScreen
28
29   // return reference to bank database
30   BankDatabase &Transaction::getBankDatabase() const
31   {
32      return bankDatabase;
33   } // end function getBankDatabase
```

Fig. 26.29 | `Transaction` class member-function definitions. (Part 2 of 2.)

Class `Transaction` has a constructor (declared in line 13 of Fig. 26.28 and defined in lines 8–15 of Fig. 26.29) that takes the current user's account number and references to the ATM's screen and the bank's database as arguments. Because `Transaction` is an abstract class, this constructor will never be called directly to instantiate `Transaction` objects. Instead, the constructors of the `Transaction` derived classes will use *base-class initializer syntax* to invoke this constructor.

Class `Transaction` has three `public` *get* functions—`getAccountNumber` (declared in line 17 of Fig. 26.28 and defined in lines 18–21 of Fig. 26.29), `getScreen` (declared in line 18 of Fig. 26.28 and defined in lines 24–27 of Fig. 26.29) and `getBankDatabase` (declared in line 19 of Fig. 26.28 and defined in lines 30–33 of Fig. 26.29). `Transaction` derived classes inherit these member functions from `Transaction` and use them to gain access to class `Transaction`'s `private` data members.

Class `Transaction` also declares a pure `virtual` function `execute` (line 22 of Fig. 26.28). It does not make sense to provide an implementation for this member function, because a generic transaction cannot be executed. Thus, we declare this member function to be a pure `virtual` function and force each `Transaction` derived class to provide its own concrete implementation that executes that particular type of transaction.

26.4.9 Class `BalanceInquiry`

Class `BalanceInquiry` (Figs. 26.30–26.31) derives from abstract base class `Transaction` and represents a balance-inquiry ATM transaction. `BalanceInquiry` does not have any data members of its own, but it inherits `Transaction` data members `accountNumber`, `screen` and `bankDatabase`, which are accessible through `Transaction`'s `public` *get* functions. Line 6 `#includes` the definition of base class `Transaction`. The `BalanceInquiry` constructor (declared in line 11 of Fig. 26.30 and defined in lines 8–13 of Fig. 26.31) takes arguments corresponding to the `Transaction` data members and simply forwards them to `Transaction`'s constructor, using *base-class initializer syntax* (line 10 of Fig. 26.31). Line 12 of Fig. 26.30 contains the function prototype for member function `execute`, which is required to indicate the intention to override the base class's pure `virtual` function of the same name.

```
1  // BalanceInquiry.h
2  // BalanceInquiry class definition. Represents a balance inquiry.
3  #ifndef BALANCE_INQUIRY_H
4  #define BALANCE_INQUIRY_H
5
6  #include "Transaction.h" // Transaction class definition
7
8  class BalanceInquiry : public Transaction
9  {
10 public:
11    BalanceInquiry(int, Screen &, BankDatabase &); // constructor
12    virtual void execute(); // perform the transaction
13 }; // end class BalanceInquiry
14
15 #endif // BALANCE_INQUIRY_H
```

Fig. 26.30 | `BalanceInquiry` class definition.

```
1  // BalanceInquiry.cpp
2  // Member-function definitions for class BalanceInquiry.
3  #include "BalanceInquiry.h" // BalanceInquiry class definition
4  #include "Screen.h" // Screen class definition
5  #include "BankDatabase.h" // BankDatabase class definition
6
7  // BalanceInquiry constructor initializes base-class data members
8  BalanceInquiry:: BalanceInquiry(int userAccountNumber, Screen &atmScreen,
9     BankDatabase &atmBankDatabase)
10    : Transaction(userAccountNumber, atmScreen, atmBankDatabase)
11 {
12    // empty body
13 } // end BalanceInquiry constructor
14
15 // performs transaction; overrides Transaction's pure virtual function
16 void BalanceInquiry::execute()
17 {
18    // get references to bank database and screen
19    BankDatabase &bankDatabase = getBankDatabase();
20    Screen &screen = getScreen();
21
22    // get the available balance for the current user's Account
23    double availableBalance =
24       bankDatabase.getAvailableBalance(getAccountNumber());
25
26    // get the total balance for the current user's Account
27    double totalBalance =
28       bankDatabase.getTotalBalance(getAccountNumber());
29
30    // display the balance information on the screen
31    screen.displayMessageLine("\nBalance Information:");
32    screen.displayMessage(" - Available balance: ");
33    screen.displayDollarAmount(availableBalance);
```

Fig. 26.31 | `BalanceInquiry` class member-function definitions. (Part 1 of 2.)

```
34      screen.displayMessage("\n - Total balance:     ");
35      screen.displayDollarAmount(totalBalance);
36      screen.displayMessageLine("");
37   } // end function execute
```

Fig. 26.31 | `BalanceInquiry` class member-function definitions. (Part 2 of 2.)

Class `BalanceInquiry` overrides `Transaction`'s pure `virtual` function `execute` to provide a concrete implementation (lines 16–37 of Fig. 26.31) that performs the steps involved in a balance inquiry. Lines 19–20 get references to the bank database and the ATM's screen by invoking member functions inherited from base class `Transaction`. Lines 23–24 retrieve the available balance of the account involved by invoking member function `getAvailableBalance` of `bankDatabase`. Line 24 uses inherited member function `getAccountNumber` to get the account number of the current user, which it then passes to `getAvailableBalance`. Lines 27–28 retrieve the total balance of the current user's account. Lines 31–36 display the balance information on the ATM's screen. Recall that `displayDollarAmount` takes a `double` argument and outputs it to the screen formatted as a dollar amount. For example, if a user's `availableBalance` is `700.5`, line 33 outputs `$700.50`. Line 36 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`).

26.4.10 Class `Withdrawal`

Class `Withdrawal` (Figs. 26.32–26.33) derives from `Transaction` and represents a withdrawal ATM transaction. Figure 26.32 expands upon the header file for this class developed in Fig. 26.13. Class `Withdrawal` has a constructor and one member function `execute`, which we discuss shortly. Recall from the class diagram of Fig. 26.11 that class `Withdrawal` has one attribute, `amount`, which line 16 implements as an `int` data member. Figure 26.10 models associations between class `Withdrawal` and classes `Keypad` and `CashDispenser`, for which lines 17–18 implement references `keypad` and `cashDispenser`, respectively. Line 19 is the function prototype of a `private` utility function that we soon discuss.

```
1  // Withdrawal.h
2  // Withdrawal class definition. Represents a withdrawal transaction.
3  #ifndef WITHDRAWAL_H
4  #define WITHDRAWAL_H
5
6  #include "Transaction.h" // Transaction class definition
7  class Keypad; // forward declaration of class Keypad
8  class CashDispenser; // forward declaration of class CashDispenser
9
10 class Withdrawal : public Transaction
11 {
12 public:
13    Withdrawal(int, Screen &, BankDatabase &, Keypad &, CashDispenser &);
14    virtual void execute(); // perform the transaction
```

Fig. 26.32 | `Withdrawal` class definition. (Part 1 of 2.)

```
15   private:
16      int amount; // amount to withdraw
17      Keypad &keypad; // reference to ATM's keypad
18      CashDispenser &cashDispenser; // reference to ATM's cash dispenser
19      int displayMenuOfAmounts() const; // display the withdrawal menu
20   }; // end class Withdrawal
21
22   #endif // WITHDRAWAL_H
```

Fig. 26.32 | `Withdrawal` class definition. (Part 2 of 2.)

`Withdrawal` Class Member-Function Definitions

Figure 26.33 contains the member-function definitions for class `Withdrawal`. Line 3 `#in-cludes` the class's definition, and lines 4–7 `#include` the definitions of the other classes used in `Withdrawal`'s member functions. Line 11 declares a global constant corresponding to the cancel option on the withdrawal menu. We'll soon discuss how the class uses this constant.

```
1   // Withdrawal.cpp
2   // Member-function definitions for class Withdrawal.
3   #include "Withdrawal.h" // Withdrawal class definition
4   #include "Screen.h" // Screen class definition
5   #include "BankDatabase.h" // BankDatabase class definition
6   #include "Keypad.h" // Keypad class definition
7   #include "CashDispenser.h" // CashDispenser class definition
8
9   // global constant that corresponds to menu option to cancel
10  static const int CANCELED = 6;
11
12  // Withdrawal constructor initialize class's data members
13  Withdrawal::Withdrawal(int userAccountNumber, Screen &atmScreen,
14     BankDatabase &atmBankDatabase, Keypad &atmKeypad,
15     CashDispenser &atmCashDispenser)
16     : Transaction(userAccountNumber, atmScreen, atmBankDatabase),
17       keypad(atmKeypad), cashDispenser(atmCashDispenser)
18  {
19     // empty body
20  } // end Withdrawal constructor
21
22  // perform transaction; overrides Transaction's pure virtual function
23  void Withdrawal::execute()
24  {
25     bool cashDispensed = false; // cash was not dispensed yet
26     bool transactionCanceled = false; // transaction was not canceled yet
27
28     // get references to bank database and screen
29     BankDatabase &bankDatabase = getBankDatabase();
30     Screen &screen = getScreen();
31
```

Fig. 26.33 | `Withdrawal` class member-function definitions. (Part 1 of 3.)

```
32      // loop until cash is dispensed or the user cancels
33      do
34      {
35         // obtain the chosen withdrawal amount from the user
36         int selection = displayMenuOfAmounts();
37
38         // check whether user chose a withdrawal amount or canceled
39         if (selection != CANCELED)
40         {
41            amount = selection; // set amount to the selected dollar amount
42
43            // get available balance of account involved
44            double availableBalance =
45               bankDatabase.getAvailableBalance(getAccountNumber());
46
47            // check whether the user has enough money in the account
48            if (amount <= availableBalance)
49            {
50               // check whether the cash dispenser has enough money
51               if (cashDispenser.isSufficientCashAvailable(amount))
52               {
53                  // update the account involved to reflect withdrawal
54                  bankDatabase.debit(getAccountNumber(), amount);
55
56                  cashDispenser.dispenseCash(amount); // dispense cash
57                  cashDispensed = true; // cash was dispensed
58
59                  // instruct user to take cash
60                  screen.displayMessageLine(
61                     "\nPlease take your cash from the cash dispenser.");
62               } // end if
63               else // cash dispenser does not have enough cash
64                  screen.displayMessageLine(
65                     "\nInsufficient cash available in the ATM."
66                     "\n\nPlease choose a smaller amount.");
67            } // end if
68            else // not enough money available in user's account
69            {
70               screen.displayMessageLine(
71                  "\nInsufficient funds in your account."
72                  "\n\nPlease choose a smaller amount.");
73            } // end else
74         } // end if
75         else // user chose cancel menu option
76         {
77            screen.displayMessageLine("\nCanceling transaction...");
78            transactionCanceled = true; // user canceled the transaction
79         } // end else
80      } while (!cashDispensed && !transactionCanceled); // end do...while
81   } // end function execute
82
```

Fig. 26.33 | `Withdrawal` class member-function definitions. (Part 2 of 3.)

```
83  // display a menu of withdrawal amounts and the option to cancel;
84  // return the chosen amount or 0 if the user chooses to cancel
85  int Withdrawal::displayMenuOfAmounts() const
86  {
87     int userChoice = 0; // local variable to store return value
88
89     Screen &screen = getScreen(); // get screen reference
90
91     // array of amounts to correspond to menu numbers
92     int amounts[] = { 0, 20, 40, 60, 100, 200 };
93
94     // loop while no valid choice has been made
95     while (userChoice == 0)
96     {
97        // display the menu
98        screen.displayMessageLine("\nWithdrawal options:");
99        screen.displayMessageLine("1 - $20");
100       screen.displayMessageLine("2 - $40");
101       screen.displayMessageLine("3 - $60");
102       screen.displayMessageLine("4 - $100");
103       screen.displayMessageLine("5 - $200");
104       screen.displayMessageLine("6 - Cancel transaction");
105       screen.displayMessage("\nChoose a withdrawal option (1-6): ");
106
107       int input = keypad.getInput(); // get user input through keypad
108
109       // determine how to proceed based on the input value
110       switch (input)
111       {
112          case 1: // if the user chose a withdrawal amount
113          case 2: // (i.e., chose option 1, 2, 3, 4 or 5), return the
114          case 3: // corresponding amount from amounts array
115          case 4:
116          case 5:
117             userChoice = amounts[ input ]; // save user's choice
118             break;
119          case CANCELED: // the user chose to cancel
120             userChoice = CANCELED; // save user's choice
121             break;
122          default: // the user did not enter a value from 1-6
123             screen.displayMessageLine(
124                "\nIvalid selection. Try again.");
125       } // end switch
126    } // end while
127
128    return userChoice; // return withdrawal amount or CANCELED
129 } // end function displayMenuOfAmounts
```

Fig. 26.33 | `Withdrawal` class member-function definitions. (Part 3 of 3.)

Class `Withdrawal`'s constructor (defined in lines 13–20 of Fig. 26.33) has five parameters. It uses a base-class initializer in line 16 to pass parameters `userAccountNumber`, `atmScreen` and `atmBankDatabase` to base class `Transaction`'s constructor to set the data members that `Withdrawal` inherits from `Transaction`. The constructor also takes refer-

ences atmKeypad and atmCashDispenser as parameters and assigns them to reference data members keypad and cashDispenser using member initializers (line 17).

Class Withdrawal overrides Transaction's pure virtual function execute with a concrete implementation (lines 23–81) that performs the steps involved in a withdrawal. Line 25 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 26 declares and initializes to false a bool variable transactionCanceled that indicates whether the transaction has been canceled by the user. Lines 29–30 get references to the bank database and the ATM's screen by invoking member functions inherited from base class Transaction.

Lines 33–80 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). This loop continuously returns 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 36 displays a menu of withdrawal amounts and obtains a user selection by calling private utility function displayMenuOfAmounts (defined in lines 85–129). This function displays the menu of amounts and returns either an int withdrawal amount or the int constant CANCELED to indicate that the user has chosen to cancel the transaction.

Member function displayMenuOfAmounts (lines 85–129) first declares local variable userChoice (initially 0) to store the value that the member function will return (line 87). Line 89 gets a reference to the screen by calling member function getScreen inherited from base class Transaction. Line 92 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 in lines 95–126 repeats until userChoice takes on a value other than 0. We'll see shortly that this occurs when the user makes a valid selection from the menu. Lines 98–105 display the withdrawal menu on the screen and prompt the user to enter a choice. Line 107 obtains integer input through the keypad. The switch statement in lines 110–125 determines how to proceed based on the user's input. If the user selects a number between 1 and 5, line 117 sets userChoice to the value of the element in amounts at index input. For example, if the user enters 3 to withdraw $60, line 117 sets userChoice to the value of amounts[3] (i.e., 60). Line 118 terminates the switch. Variable userChoice no longer equals 0, so the while in lines 95–126 terminates and line 128 returns userChoice. If the user selects the cancel menu option, lines 120–121 execute, setting userChoice to CANCELED and causing the member function to return this value. If the user does not enter a valid menu selection, lines 123–124 display an error message and the user is returned to the withdrawal menu.

The if statement in line 39 in member function execute determines whether the user has selected a withdrawal amount or chosen to cancel. If the user cancels, lines 77–78 execute to display an appropriate message to the user and set transactionCanceled to true. This causes the loop-continuation test in line 80 to fail and control to return to the calling member function (i.e., ATM member function performTransactions). If the user has chosen a withdrawal amount, line 41 assigns local variable selection to data member amount. Lines 44–45 retrieve the available balance of the current user's Account and store it in a local double variable availableBalance. Next, the if statement in line 48 deter-

mines whether the selected amount is less than or equal to the user's available balance. If it isn't, lines 70–72 display an appropriate error message. Control then continues to the end of the `do...while`, and the loop repeats because both `cashDispensed` and `transactionCanceled` are still `false`. If the user's balance is high enough, the `if` statement in line 51 determines whether the cash dispenser has enough money to satisfy the withdrawal request by invoking the `cashDispenser`'s `isSufficientCashAvailable` member function. If this member function returns `false`, lines 64–66 display an appropriate error message and the `do...while` repeats. If sufficient cash is available, then the requirements for the withdrawal are satisfied, and line 54 debits `amount` from the user's account in the database. 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 that cash has been dispensed. Because `cashDispensed` is now `true`, control continues after the `do...while`. No additional statements appear below the loop, so the member function returns control to class `ATM`.

In the function calls in lines 64–66 and lines 70–72, we divide the argument to `Screen` member function `displayMessageLine` into two string literals, each placed on a separate line in the program. We do so because each argument is too long to fit on a single line. *C++ concatenates (i.e., combines) string literals adjacent to each other, even if they are on separate lines.* For example, if you write `"Happy " "Birthday"` in a program, C++ will view these two adjacent string literals as the single string literal `"Happy Birthday"`. As a result, when lines 64–66 execute, `displayMessageLine` receives a single `string` as a parameter, even though the argument in the function call appears as two string literals.

26.4.11 Class `Deposit`

Class `Deposit` (Figs. 26.34–26.35) derives from `Transaction` and represents a deposit ATM transaction. Figure 26.34 contains the `Deposit` class definition. Like derived classes `BalanceInquiry` and `Withdrawal`, `Deposit` declares a constructor (line 13) and member function `execute` (line 14)—we discuss these momentarily. Recall from the class diagram of Fig. 26.11 that class `Deposit` has one attribute `amount`, which line 16 implements as an `int` data member. Lines 17–18 create reference data members `keypad` and `depositSlot` that implement the associations between class `Deposit` and classes `Keypad` and `DepositSlot` modeled in Fig. 26.10. Line 19 contains the function prototype for a `private` utility function `promptForDepositAmount` that we'll discuss shortly.

```
1  // Deposit.h
2  // Deposit class definition. Represents a deposit transaction.
3  #ifndef DEPOSIT_H
4  #define DEPOSIT_H
5
6  #include "Transaction.h" // Transaction class definition
7  class Keypad; // forward declaration of class Keypad
8  class DepositSlot; // forward declaration of class DepositSlot
9
10 class Deposit : public Transaction
11 {
```

Fig. 26.34 | `Deposit` class definition. (Part 1 of 2.)

```
12  public:
13     Deposit(int, Screen &, BankDatabase &, Keypad &, DepositSlot &);
14     virtual void execute(); // perform the transaction
15  private:
16     double amount; // amount to deposit
17     Keypad &keypad; // reference to ATM's keypad
18     DepositSlot &depositSlot; // reference to ATM's deposit slot
19     double promptForDepositAmount() const; // get deposit amount from user
20  }; // end class Deposit
21
22  #endif // DEPOSIT_H
```

Fig. 26.34 | `Deposit` class definition. (Part 2 of 2.)

`Deposit` Class Member-Function Definitions

Figure 26.35 presents the `Deposit` class implementation. Line 3 `#includes` the `Deposit` class definition, and lines 4–7 `#include` the class definitions of the other classes used in `Deposit`'s member functions. Line 9 declares a constant `CANCELED` that corresponds to the value a user enters to cancel a deposit. We'll soon discuss how the class uses this constant.

```
1  // Deposit.cpp
2  // Member-function definitions for class Deposit.
3  #include "Deposit.h" // Deposit class definition
4  #include "Screen.h" // Screen class definition
5  #include "BankDatabase.h" // BankDatabase class definition
6  #include "Keypad.h" // Keypad class definition
7  #include "DepositSlot.h" // DepositSlot class definition
8
9  static const int CANCELED = 0; // constant representing cancel option
10
11 // Deposit constructor initializes class's data members
12 Deposit::Deposit(int userAccountNumber, Screen &atmScreen,
13    BankDatabase &atmBankDatabase, Keypad &atmKeypad,
14    DepositSlot &atmDepositSlot)
15    : Transaction(userAccountNumber, atmScreen, atmBankDatabase),
16      keypad(atmKeypad), depositSlot(atmDepositSlot)
17 {
18    // empty body
19 } // end Deposit constructor
20
21 // performs transaction; overrides Transaction's pure virtual function
22 void Deposit::execute()
23 {
24    BankDatabase &bankDatabase = getBankDatabase(); // get reference
25    Screen &screen = getScreen(); // get reference
26
27    amount = promptForDepositAmount(); // get deposit amount from user
28
29    // check whether user entered a deposit amount or canceled
30    if (amount != CANCELED)
31    {
```

Fig. 26.35 | `Deposit` class member-function definitions. (Part 1 of 2.)

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

Fig. 26.35 | `Deposit` class member-function definitions. (Part 2 of 2.)

Like class `Withdrawal`, class `Deposit` contains a constructor (lines 12–19) that passes three parameters to base class `Transaction`'s constructor using a base-class initializer (line 15). The constructor also has parameters `atmKeypad` and `atmDepositSlot`, which it assigns to its corresponding data members (line 16).

Member function `execute` (lines 22–62) overrides pure `virtual` function `execute` in base class `Transaction` with a concrete implementation that performs the steps required in a deposit transaction. Lines 24–25 get references to the database and the screen. Line 27 prompts the user to enter a deposit amount by invoking `private` utility function `promptForDepositAmount` (defined in lines 65–81) and sets data member `amount` to the value returned. Member function `promptForDepositAmount` asks the user to enter a deposit amount as an integer number of cents (because the ATM's keypad does not contain a decimal point; this is consistent with many real ATMs) and returns the `double` value representing the dollar amount to be deposited.

Line 67 in member function `promptForDepositAmount` gets a reference to the ATM's screen. Lines 70–71 display a message on the screen asking the user to input a deposit amount as a number of cents or "0" to cancel the transaction. Line 72 receives the user's input from the keypad. The `if` statement in lines 75–80 determines whether the user has entered a real deposit amount or chosen to cancel. If the user chooses to cancel, line 76 returns the constant `CANCELED`. Otherwise, line 79 returns the deposit amount after converting from the number of cents to a dollar amount by casting `input` to a `double`, then dividing by `100`. For example, if the user enters `125` as the number of cents, line 79 returns `125.0` divided by `100`, or `1.25`—125 cents is $1.25.

The `if` statement in lines 30–61 in member function `execute` determines whether the user has chosen to cancel the transaction instead of entering a deposit amount. If the user cancels, line 60 displays an appropriate message, and the member function returns. If the user enters a deposit amount, lines 33–36 instruct the user to insert a deposit envelope with the correct amount. Recall that `Screen` member function `displayDollarAmount` outputs a `double` formatted as a dollar amount.

Line 39 sets a local `bool` variable to the value returned by `depositSlot`'s `isEnvelopeReceived` member function, indicating whether a deposit envelope has been received. Recall that we coded `isEnvelopeReceived` (lines 7–10 of Fig. 26.23) to always return `true`, because we are simulating the functionality of the deposit slot and assume that the user always inserts an envelope. However, we code member function `execute` of class `Deposit` to test for the possibility that the user does not insert an envelope—good software engineering demands that programs account for all possible return values. Thus, class `Deposit` is prepared for future versions of `isEnvelopeReceived` that could return `false`. Lines 44–50 execute if the deposit slot receives an envelope. Lines 44–47 display an appropriate message to the user. Line 50 then credits the deposit amount to the user's account in the database. Lines 54–55 will execute if the deposit slot does not receive a deposit envelope. In this case, we display a message to the user stating that the ATM has canceled the transaction. The member function then returns without modifying the user's account.

26.4.12 Test Program `ATMCaseStudy.cpp`

`ATMCaseStudy.cpp` (Fig. 26.36) is a simple C++ program that allows us to start, or "turn on," the ATM and test the implementation of our ATM system model. The program's

`main` function (lines 6–11) does nothing more than instantiate a new `ATM` object named `atm` (line 8) and invoke its `run` member function (line 9) to start the ATM.

```
1  // ATMCaseStudy.cpp
2  // Driver program for the ATM case study.
3  #include "ATM.h" // ATM class definition
4
5  // main function creates and runs the ATM
6  int main()
7  {
8     ATM atm; // create an ATM object
9     atm.run(); // tell the ATM to start
10 } // end main
```

Fig. 26.36 | `ATMCaseStudy.cpp` starts the ATM system.

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. We've mentioned "friendship" as an exception to private visibility. Friendship is discussed in Chapter 9.

26.2 b.

26.3 The design for class `Account` yields the header file in Fig. 26.37.

```
1  // Fig. 26.37: Account.h
2  // Account class definition. Represents a bank account.
3  #ifndef ACCOUNT_H
4  #define ACCOUNT_H
5
6  class Account
7  {
8  public:
9     bool validatePIN(int); // is user-specified PIN correct?
10    double getAvailableBalance(); // returns available balance
11    double getTotalBalance(); // returns total balance
12    void credit(double); // adds an amount to the Account
13    void debit(double); // subtracts an amount from the Account
14 private:
15    int accountNumber; // account number
16    int pin; // PIN for authentication
```

Fig. 26.37 | `Account` class header file based on Fig. 26.1 and Fig. 26.2. (Part 1 of 2.)

```
17      double availableBalance; // funds available for withdrawal
18      double totalBalance; // funds available + funds waiting to clear
19   }; // end class Account
20
21   #endif // ACCOUNT_H
```

Fig. 26.37 | `Account` class header file based on Fig. 26.1 and Fig. 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 header file in Fig. 26.38. In the implementation, a constructor initializes `private` reference attributes `screen` and `bankDatabase` to actual objects, and member functions `getScreen` and `getBankDatabase` access these attributes. These member functions allow classes derived from `Transaction` to access the ATM's screen and interact with the bank's database.

```
1   // Fig. 36.38: Transaction.h
2   // Transaction abstract base class definition.
3   #ifndef TRANSACTION_H
4   #define TRANSACTION_H
5
6   class Screen; // forward declaration of class Screen
7   class BankDatabase; // forward declaration of class BankDatabase
8
9   class Transaction
10  {
11  public:
12     int getAccountNumber(); // return account number
13     Screen &getScreen(); // return reference to screen
14     BankDatabase &getBankDatabase(); // return reference to bank database
15
16     // pure virtual function to perform the transaction
17     virtual void execute() = 0; // overridden in derived classes
18  private:
19     int accountNumber; // indicates account involved
20     Screen &screen; // reference to the screen of the ATM
21     BankDatabase &bankDatabase; // reference to the account info database
22  }; // end class Transaction
23
24  #endif // TRANSACTION_H
```

Fig. 26.38 | `Transaction` class header file based on Fig. 26.10 and Fig. 26.11.