Allow the user to specify (via an input dialog) the number of shapes to generate. The program will then generate and display the shapes along with a status bar that informs the user how many of each shape were created.

10.2 *(Drawing Application Modification)* In Exercise 10.1, you created a `MyShape` hierarchy in which classes `MyLine`, `MyOval` and `MyRectangle` extend `MyShape` directly. If your hierarchy was properly designed, you should be able to see the similarities between the `MyOval` and `MyRectangle` classes. Redesign and reimplement the code for the `MyOval` and `MyRectangle` classes to "factor out" the common features into the abstract class `MyBoundedShape` to produce the hierarchy in Fig. 10.18.

Class `MyBoundedShape` should declare two constructors that mimic the constructors of class `MyShape`, only with an added parameter to set whether the shape is filled. Class `MyBoundedShape` should also declare *get* and *set* methods for manipulating the filled flag and methods that calculate the upper-left *x*-coordinate, upper-left *y*-coordinate, width and height. Remember, the values needed to draw an oval or a rectangle can be calculated from two *(x, y)* coordinates. If designed properly, the new `MyOval` and `MyRectangle` classes should each have two constructors and a `draw` method.

10.9 (Optional) Software Engineering Case Study: Incorporating Inheritance into the ATM System

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 elegant and efficient manner. We then modify our class diagram to incorporate the new inheritance relationships. Finally, we demonstrate how our updated design is translated into Java code.

In Section 3.10, 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 10.19 shows the attributes and operations of classes `BalanceInquiry`, `Withdrawal` and `Deposit`. Note that these classes have one attribute (`accountNumber`) and one operation (`execute`) in common. Each class requires attribute `accountNumber` to specify the account to which the

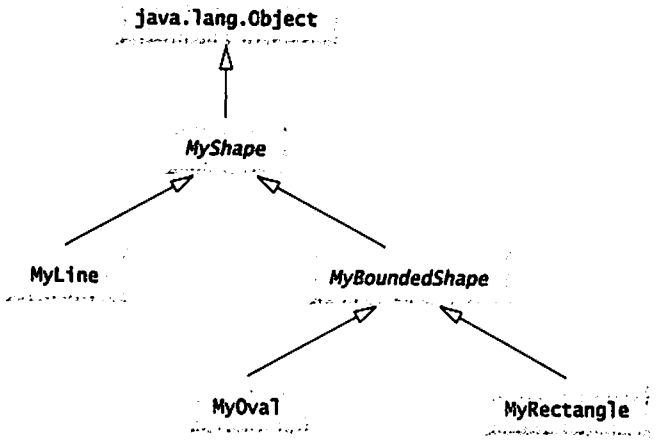

Fig. 10.18 | MyShape hierarchy with MyBoundedShape.

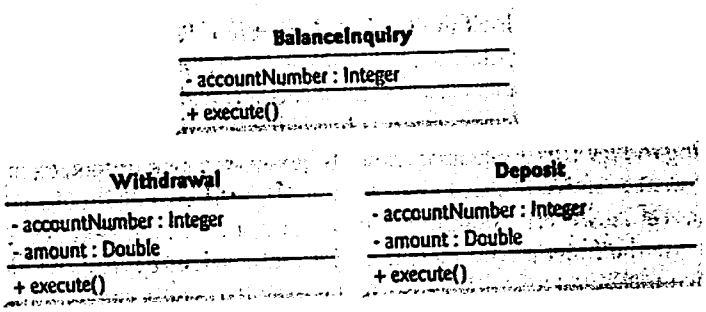

Fig. 10.19 | Attributes and operations of classes `BalanceInquiry`, `Withdrawal` and `Deposit`.

transaction applies. Each class contains operation `execute`, which the ATM invokes to perform a transaction. Clearly, `BalanceInquiry`, `Withdrawal` and `Deposit` represent *types of* transactions. Figure 10.19 reveals commonality among the transaction classes, so using inheritance to factor out the common features seems appropriate for designing classes `BalanceInquiry`, `Withdrawal` and `Deposit`. We place the common functionality in a superclass, `Transaction`, that classes `BalanceInquiry`, `Withdrawal` and `Deposit` extend.

The UML specifies a relationship called a generalization to model inheritance. Figure 10.20 is the class diagram that models the generalization of superclass `Transaction` and subclasses `BalanceInquiry`, `Withdrawal` and `Deposit`. The arrows with triangular hollow arrowheads indicate that classes `BalanceInquiry`, `Withdrawal` and `Deposit` extend class `Transaction`. Class `Transaction` is said to be a generalization of classes `BalanceInquiry`, `Withdrawal` and `Deposit`. Class `BalanceInquiry`, `Withdrawal` and `Deposit` are said to be specializations of class `Transaction`.

Classes `BalanceInquiry`, `Withdrawal` and `Deposit` share integer attribute `accountNumber`, so we factor out this common attribute and place it in superclass `Transaction`.

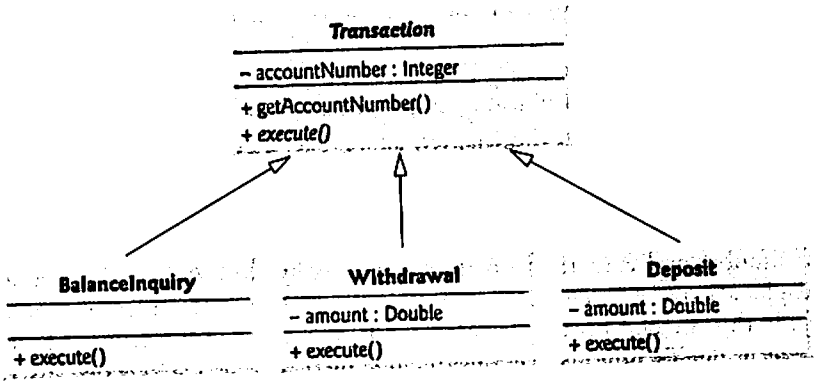

Fig. 10.20 | Class diagram modeling generalization of superclass `Transaction` and subclasses `BalanceInquiry`, `Withdrawal` and `Deposit`. Note that abstract class names (e.g., `Transaction`) and method names (e.g., `execute` in class `Transaction`) appear in italics.

We no longer list accountNumber in the second compartment of each subclass, because the three subclasses inherit this attribute from Transaction. Recall, however, that subclasses cannot access private attributes of a superclass. We therefore include public method getAccountNumber in class Transaction. Each subclass will inherit this method, enabling the subclass to access its accountNumber as needed to execute a transaction.

According to Fig. 10.19, classes BalanceInquiry, Withdrawal and Deposit also share operation execute, so we decided that superclass Transaction should contain public method execute. However, it does not make sense to implement execute in class Transaction, because the functionality that this method provides depends on the type of the actual transaction. We therefore declare method execute as abstract in superclass Transaction. Any class that contains at least one abstract method must also be declared abstract. This forces any subclass of Transaction that must be a concrete class (i.e., BalanceInquiry, Withdrawal and Deposit) to implement method execute. The UML requires that we place abstract class names (and abstract methods) in italics, so Transaction and its method execute appear in italics in Fig. 10.20. Note that method execute is not italicized in subclasses BalanceInquiry, Withdrawal and Deposit. Each subclass overrides superclass Transaction's execute method with a concrete implementation that performs the steps appropriate for completing that type of transaction. Note that Fig. 10.20 includes operation execute in the third compartment of classes BalanceInquiry, Withdrawal and Deposit, because each class has a different concrete implementation of the overridden method.

Incorporating inheritance provides the ATM with an elegant way to execute all transactions "in the general." For example, suppose a user chooses to perform a balance inquiry. The ATM sets a Transaction reference to a new object of class BalanceInquiry. When the ATM uses its Transaction reference to invoke method 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 subclass that overrides the execute method with a version of the method appropriate for executing 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 subclass. The ATM could execute transactions of the new type using the current code, because it executes all transactions polymorphically using a general Transaction reference.

As you learned earlier in the chapter, an abstract class like Transaction is one for which the programmer never intends to instantiate objects. An abstract class simply declares common attributes and behaviors of its subclasses 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 abstract method execute in class Transaction if it lacks a concrete implementation. Conceptually, we include this method because it corresponds to the defining behavior of all transactions—executing. Technically, we must include method execute in superclass Transaction so that the ATM (or any other class) can polymorphically invoke each subclass's overridden version of this method through a Transaction reference. Also, from a software engineering perspective, including an abstract method in a superclass forces the implementor of the subclasses to override that method with concrete implementations in the

subclasses, or else the subclasses, too, will be abstract, preventing objects of those subclasses from being instantiated.

Subclasses `BalanceInquiry`, `Withdrawal` and `Deposit` inherit attribute `accountNumber` from superclass `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` subclasses share this attribute, we do not place it in superclass `Transaction`—we place only features common to all the subclasses in the superclass, otherwise subclasses could inherit attributes (and methods) that they do not need and should not have.

Figure 10.21 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, is either executing a transaction or it 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`. Subclass `Withdrawal` inherits superclass `Transaction`'s association with class `ATM`. Subclasses `BalanceInquiry` and `Deposit`

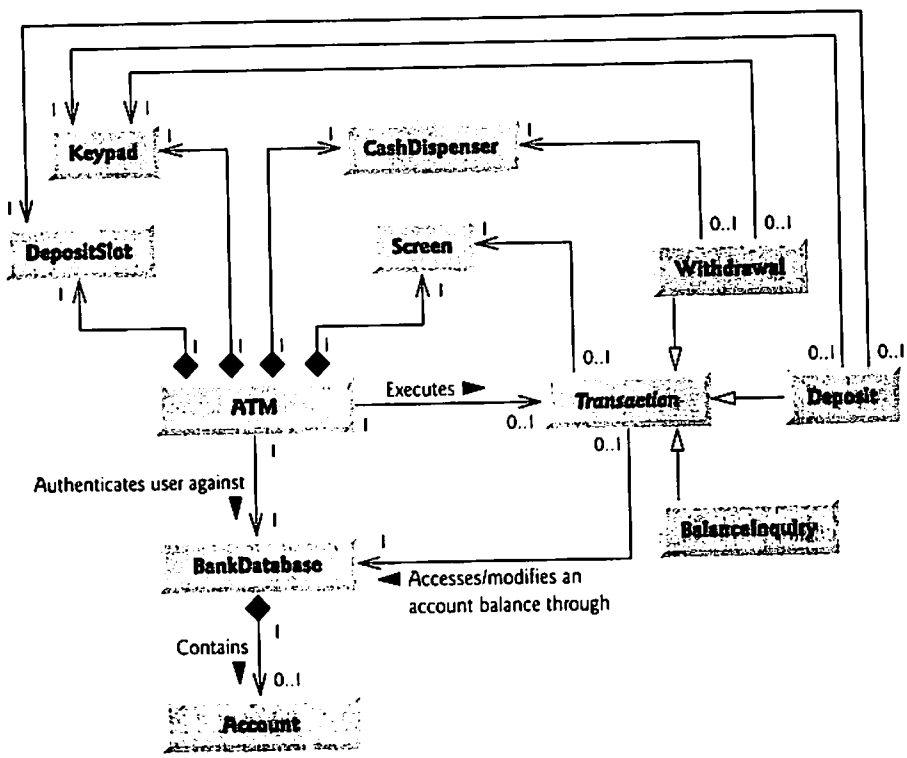

Fig. 10.21 | Class diagram of the ATM system (incorporating inheritance). Note that abstract class names (e.g., `Transaction`) appear in italics.

inherit this association, too, so the previously omitted associations between ATM and classes BalanceInquiry and Deposit no longer exist either.

We also add an association between class Transaction and the BankDatabase (Fig. 10.21). All Transactions require a reference to the BankDatabase so they can access and modify account information. Because each Transaction subclass inherits this reference, we no longer model the association between class Withdrawal and the BankDatabase. Similarly, the previously omitted associations between the BankDatabase and classes BalanceInquiry and Deposit no longer exist.

We show an association between class Transaction and the Screen. All Transactions display output to the user via the Screen. Thus, we no longer include the association previously modeled between Withdrawal and the Screen, although Withdrawal still participates in associations with the CashDispenser and the Keypad. Our class diagram incorporating inheritance also models Deposit and BalanceInquiry. We show associations between Deposit and both the DepositSlot and the Keypad. Note that class BalanceInquiry takes part in no associations other than those inherited from class Transaction—a BalanceInquiry needs to interact only with the BankDatabase and with the Screen.

The class diagram of Fig. 8.24 showed attributes and operations with visibility markers. Now we present a modified class diagram that incorporates inheritance in Fig. 10.22. This abbreviated diagram does not show inheritance relationships, but instead shows the attributes and methods after we have employed inheritance in our system. To save space, as we did in Fig. 4.24, we do not include those attributes shown by associations in Fig. 10.21—we do, however, include them in the Java implementation in Appendix J. We also omit all operation parameters, as we did in Fig. 8.24—incorporating inheritance does not affect the parameters already modeled in Fig. 6.22–Fig. 6.25.

Software Engineering Observation 10.12

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, methods and associations is substantial (as in Fig. 10.21 and Fig. 10.22), 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 methods.

Implementing the ATM System Design (Incorporating Inheritance)

In Section 8.19, we began implementing the ATM system design in Java code. 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 extends class A in the class declaration. For example, abstract superclass Transaction is a generalization of class Withdrawal. Figure 10.23 contains the shell of class Withdrawal containing the appropriate class declaration.
2. If class A is an abstract class and class B is a subclass of class A, then class B must implement the abstract methods of class A if class B is to be a concrete class. For example, class Transaction contains abstract method execute, so class Withdrawal must implement this method if we want to instantiate a Withdrawal object. Figure 10.24 is the Java code for class Withdrawal from Fig. 10.21 and Fig. 10.22. Class Withdrawal inherits field accountNumber from superclass Transaction, so Withdrawal does not need to declare this field. Class Withdrawal also inherits references to the Screen and the BankDatabase from its superclass Transaction, so

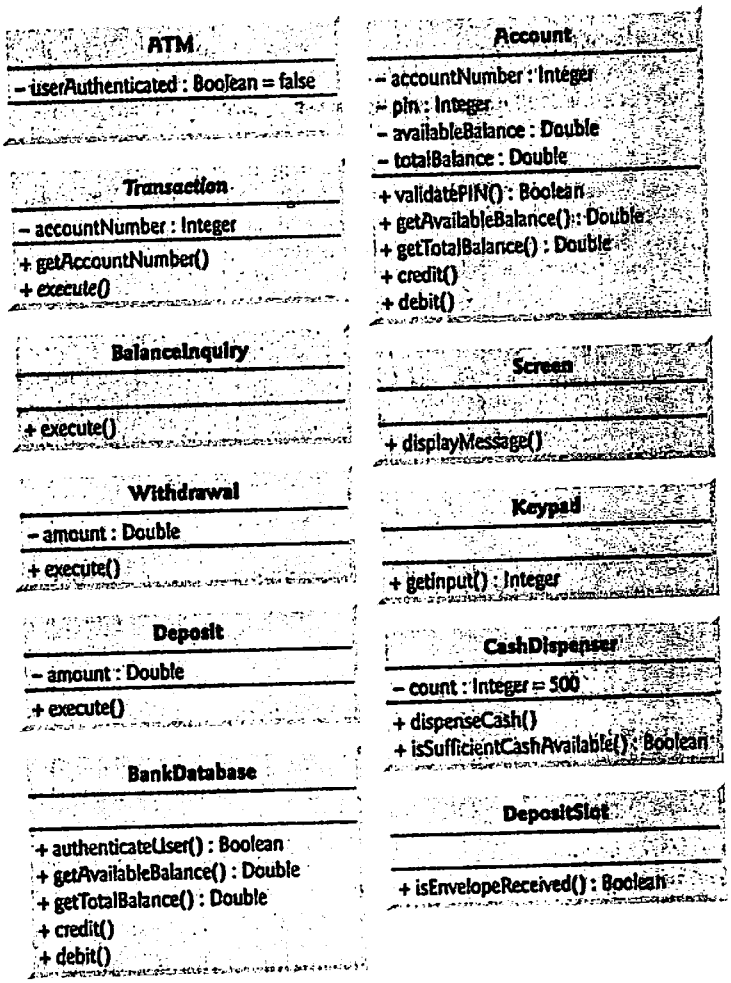

Fig. 10.22 | Class diagram with attributes and operations (incorporating inheritance). Note that abstract class names (e.g., `Transaction`) and method names (e.g., `execute` in class `Transaction`) appear in italics.

we do not include these references in our code. Figure 10.22 specifies attribute `amount` and operation `execute` for class `Withdrawal`. Line 6 of Fig. 10.24 declares a field for attribute `amount`. Lines 16–18 declare the shell of a method for operation `execute`. Recall that subclass `Withdrawal` must provide a concrete implementation of the `abstract` method `execute` in superclass `Transaction`. The keypad and `cashDispenser` references (lines 7–8) are fields derived from `Withdrawal`'s associations in Fig. 10.21. [*Note:* The constructor in the complete working version of this class will initialize these references to actual objects.]

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

Fig. 10.23 | Java code for shell of class `Withdrawal`.

```
1  // Withdrawal.java
2  // Generated using the class diagrams in Fig. 10.21 and Fig. 10.22
3  public class Withdrawal extends Transaction
4  {
5     // attributes
6     private double amount; // amount to withdraw
7     private Keypad keypad; // reference to keypad
8     private CashDispenser cashDispenser; // reference to cash dispenser
9
10    // no-argument constructor
11    public Withdrawal()
12    {
13    } // end no-argument Withdrawal constructor
14
15    // method overriding execute
16    public void execute()
17    {
18    } // end method execute
19 } // end class Withdrawal
```

Fig. 10.24 | Java code for class `Withdrawal` based on Fig. 10.21 and Fig. 10.22.

Software Engineering Observation 10.13

Several UML modeling tools convert UML-based designs into Java code and can speed the implementation process considerably. For more information on these tools, refer to the Internet and Web Resources listed at the end of Section 2.9.

Congratulations on completing the design portion of the case study! This concludes our object-oriented design of the ATM system. We completely implement the ATM system in 670 lines of Java code in Appendix J. We recommend that you carefully read the code and its description. The code is abundantly commented and precisely follows the design with which you are now familiar. The accompanying description is carefully written to guide your understanding of the implementation based on the UML design. Mastering this code is a wonderful culminating accomplishment after studying Chapters 1–8.

Software Engineering Case Study Self-Review Exercises

10.1 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