UML: ATM Case Study

Outline

1.19 Software Engineering Case Study: Introduction to Object Technology and the UML
3.10 Software Engineering Case Study: Examining the ATM Requirements Document
4.12 Software Engineering Case Study: Identifying the Classes in the ATM Requirements Document
5.14 Software Engineering Case Study: Identifying Class Attributes in the ATM System
6.10 Software Engineering Case Study: Identifying Objects’ States and Activities in the ATM System
7.15 Software Engineering Case Study: Identifying Class Operations in the ATM System
8.14 Software Engineering Case Study: Collaboration Among Objects in the ATM System
10.22 Software Engineering Case Study: Starting to Program the Classes of the ATM System
12.9 Software Engineering Case Study: Incorporating Inheritance and Polymorphism into the ATM System

OBJECTIVES

In this chapter you will learn:
• Some basics of object technology.
• UML—the industry-standard object-oriented system modeling language.
• UML: ATM Requirements Document
• UML Use Case Diagrams
• UML Class Diagrams
• UML Class Attributes
• UML Objects’ States and Activities
• UML Class Operations
• UML Collaboration
• Class Programming
• Inheritance and Polymorphism

1.19 Introduction to Object Technology and the UML

• Humans think in terms of objects such as telephones, houses, traffic lights, and microwave ovens.
• Objects have attributes (e.g., size, shape, color and weight), and exhibit behaviors (e.g., a ball rolls, bounces, inflates and deflates; a baby cries, sleeps, crawls, walks and blinks).
1.19 Introduction to Object Technology and the UML (Cont.)

- **Object-oriented design (OOD)** models software in terms similar to those used to describe real-world objects.
- New classes of objects can **inherit** characteristics of existing classes and add unique characteristics of their own.
- OOD also models communication between objects, just as people send **messages** to one another.

1.19 Introduction to Object Technology and the UML (Cont.)

- OOD **encapsulates** attributes and **operations** into objects.
- Objects have the property of **information hiding**.
  - Objects may communicate with one another across well-defined **interfaces**, but implementation details are hidden within the objects themselves.
  - Similarly, you can drive a car without knowing the details of how it works internally.

1.19 Introduction to Object Technology and the UML (Cont.)

- **Languages** like C# are **object oriented**.
  - Programming in such a language is called **object-oriented programming (OOP)**.
  - In C#, the unit of programming is the **class** from which objects are eventually **instantiated** (an OOP term for “created”).
  - C# classes contain **methods** that implement operations, and data that implements attributes.

1.19 Introduction to Object Technology and the UML (Cont.)

- **Classes** are to objects as blueprints are to houses—a class is a “plan” for building objects of the class.
- Just as we can build many houses from one blueprint, we can instantiate (create) many objects from one class.
- The **data** components of a class are called attributes or **fields**.
- The **operation** components of a class are called **methods**.
1.19 Introduction to Object Technology and the UML (Cont.)

- Packaging software as classes makes it possible for future software systems to reuse the classes.
- Groups of related classes often are packaged as reusable components.

Software Engineering Observation 1.1

Reuse of existing classes when building new classes and programs saves time, money and effort. Reuse also helps programmers build more reliable and effective systems, because existing classes and components often have gone through extensive testing, debugging and performance tuning.

1.19 Introduction to Object Technology and the UML (Cont.)

- In the 1980s, many organizations began using OOP to build their applications.
- In 1994, James Rumbaugh and Grady Booch began working to unify several processes for representing OOP.
- The Object Management GroupTM (OMG™) invited submissions for a common modeling language.
- Several companies formed the UML Partners—a consortium that developed the UML version 1.1.
- The OMG accepted the proposal and, in 1997, assumed responsibility for the continuing maintenance and revision of the UML.

What Is the UML?

- The Unified Modeling Language (UML) is the most widely used graphical representation scheme for object-oriented systems.
- The UML is extensible (i.e., capable of being enhanced with new features).
- Developers use the UML to express their designs with a standard set of graphical notations.

3.10 Examining the ATM Requirements Document

- A requirements document specifies the overall purpose of a system and what it must do.
- A small local bank intends to install a new ATM (Fig. 3.28).

Fig. 3.28 | Automated teller machine user interface.
3.10 Examining the ATM Requirements Document (Cont.)

- The ATM should have the following functionality.
  - The screen prompts the user to enter an account number.
  - The user enters a five-digit account number, using the keypad.
  - For authentication purposes, the screen prompts the user to enter the PIN.
  - The user enters a five-digit PIN, using the keypad.
  - If the user enters a valid account number and the correct PIN for that account, the screen displays the main menu.

3.10 Examining the ATM Requirements Document (Cont.)

- The main menu (Fig. 3.29) displays a numbered option for each of the three types of transactions, and the option to exit the system.

- The user enters 2 to make a withdrawal (Fig. 3.30).
- If the user enters 1, the screen displays the user’s account balance from the bank’s database.

- The user enters 3 to make a deposit:
  1. The screen prompts the user to enter a deposit amount.
  2. The screen displays a message telling the user to insert a deposit envelope into the deposit slot.
  3. If the deposit slot receives a deposit envelope, the ATM credits the deposit amount to the user’s account balance in the bank’s database.
3.10 Examining the ATM Requirements Document (Cont.)

- **Requirements gathering** might include interviews with potential users of the system and specialists in fields related to the system.
- The **software life cycle** specifies the stages through which software evolves from the time it is conceived to the time it is retired from use.
  - **Waterfall models** perform each stage once in succession.
  - **Iterative models** may repeat one or more stages several times throughout a product’s life cycle.

• To capture what a proposed system should do, developers often employ a technique known as **use case modeling**.
• Use cases represent different capabilities that the system provides to its clients.
• The simplified ATM system we build in this case study requires only the first three use cases (Fig. 3.31).

**Use Case Diagrams**

• A **use case diagram** models the interactions between a system’s clients and the system.
• Figure 3.31 shows the use case diagram for our ATM system.

• The stick figure represents an **actor**, which defines the roles that an external entity—such as a person or another system—plays when interacting with the system.
3.10 Examining the ATM Requirements Document (Cont.)

- A **design specification** should specify how the system should be constructed to satisfy the system requirements.
- A **system** is a set of components that interact to solve a problem.
- For example, to perform the ATM system’s designated tasks, our ATM system has a user interface (Fig. 3.28), contains software that executes financial transactions and interacts with a database of bank-account information.
  - **System structure** describes the system’s objects and their interrelationships.
  - **System behavior** describes how the system changes as its objects interact with one another.

4.12 Software Engineering Case Study: Identifying the Classes in the ATM Requirements Document

- We create classes only for the nouns and noun phrases in the ATM system (Fig. 4.21).
- We do not need to model some nouns such as “bank” which are not part of the ATM operations.

<table>
<thead>
<tr>
<th>Nouns and noun phrases in the requirements document</th>
</tr>
</thead>
<tbody>
<tr>
<td>bank</td>
</tr>
<tr>
<td>ATM</td>
</tr>
<tr>
<td>user</td>
</tr>
<tr>
<td>customer</td>
</tr>
<tr>
<td>transaction</td>
</tr>
<tr>
<td>account</td>
</tr>
<tr>
<td>balance</td>
</tr>
</tbody>
</table>

![Fig. 4.21 | Nouns and noun phrases in the requirements document.](image)

4.12 Identifying the Classes in the ATM Requirements Document (Cont.)

- **UML class diagrams** model the classes in the ATM system and their interrelationships (Fig. 4.22).
  - The top compartment contains the name of the class.
  - The middle compartment contains the class’s attributes.
  - The bottom compartment contains the class’s operations.

![Fig. 4.22 | Representing a class in the UML using a class diagram.](image)

4.12 Identifying the Classes in the ATM Requirements Document (Cont.)

- Figure 4.23 shows how our classes **ATM** and **Withdrawal** relate to one another.
  - The line that connects the two classes represents an **association**.
  - **Multiplicity** values indicate how many objects of each class participate in the association.
  - One ATM object participates in an association with either zero or one Withdrawal objects.
- **currentTransaction** is a **role name**, which identifies the role the Withdrawal object plays.

![Fig. 4.23 | Class diagram showing an association among classes.](image)
4.12 Software Engineering Case Study: Identifying the Classes in the ATM
Requirements Document (Cont.)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>1</td>
<td>One</td>
</tr>
<tr>
<td>m</td>
<td>An integer value</td>
</tr>
<tr>
<td>0..1</td>
<td>Zero or one</td>
</tr>
<tr>
<td>m, n</td>
<td>m or n</td>
</tr>
<tr>
<td>m..n</td>
<td>At least m, but not more than n</td>
</tr>
<tr>
<td>*</td>
<td>Any nonnegative integer (zero or more)</td>
</tr>
<tr>
<td>0..*</td>
<td>Zero or more (identical to *)</td>
</tr>
<tr>
<td>1..*</td>
<td>One or more</td>
</tr>
</tbody>
</table>

Fig. 4.24 | Multiplicity types.

4.12 Identifying the Classes in the ATM
Requirements Document (Cont.)

- Composition relationships have the following properties:
  - Only one class in the relationship can represent the whole.
  - The parts in the composition relationship exist only as long as the whole.
  - A part may belong to only one whole at a time.

- If a “has-a” relationship does not satisfy one or more of these criteria, hollow diamonds are used to indicate aggregation.

4.12 Identifying the Classes in the ATM
Requirements Document (Cont.)

- In Fig. 4.25, the solid diamonds indicate that class ATM has a composition relationship with classes Screen, Keypad, CashDispenser and DepositSlot.

- Composition implies a whole/part relationship—the ATM “has a” screen, a keypad, a cash dispenser and a deposit slot.

- The has-a relationship defines composition.

![Diagram showing composition relationships]

4.12 Identifying the Classes in the ATM
Requirements Document (Cont.)

- Figure 4.26 shows a class diagram for the ATM system.

- The class diagram shows that class ATM has a one-to-one relationship with class BankDatabase.

- We also model that one object of class BankDatabase participates in a composition relationship with zero or more objects of class Account.
4.12 Identifying the Classes in the ATM Requirements Document (Cont.)

Fig. 4.26 | Class diagram for the ATM system model.

5.14 Identifying Class Attributes in the ATM System

- A person’s attributes include height, weight and whether the person is left-handed, right-handed or ambidextrous.
- We can identify attributes of the classes in our system by looking for descriptive words and phrases in the requirements document.

<table>
<thead>
<tr>
<th>Class</th>
<th>Descriptive words and phrases</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATM</td>
<td>user is authenticated</td>
</tr>
<tr>
<td>Balance Inquiry</td>
<td>account number</td>
</tr>
<tr>
<td>withdrawal</td>
<td>account number amount</td>
</tr>
<tr>
<td>Deposit</td>
<td>account number amount</td>
</tr>
<tr>
<td>BankDatabase</td>
<td>[no descriptive words or phrases]</td>
</tr>
<tr>
<td>Account</td>
<td>account number</td>
</tr>
<tr>
<td>PIN</td>
<td>balance</td>
</tr>
<tr>
<td>Screen</td>
<td>[no descriptive words or phrases]</td>
</tr>
<tr>
<td>Keypad</td>
<td>[no descriptive words or phrases]</td>
</tr>
<tr>
<td>CashDispenser</td>
<td>begins each day loaded with 500 $20 bills</td>
</tr>
<tr>
<td>DepositSlot</td>
<td>[no descriptive words or phrases]</td>
</tr>
</tbody>
</table>

Fig. 5.18 | Descriptive words and phrases from the ATM requirements document.

5.14 Identifying Class Attributes in the ATM System (Cont.)

- The class diagram in Fig. 5.19 lists some of the attributes for the classes in our system.
- We list each attribute’s name and type, followed in some cases by an initial value.

Fig. 5.19 | Classes with attributes.
5.14 Identifying Class Attributes in the ATM System (Cont.)

- Consider the `userAuthenticated` attribute of class ATM:
  
  ```
  userAuthenticated : bool = false
  ```

- This attribute declaration contains:
  
  - the attribute name, `userAuthenticated`.
  - the attribute type, `bool`.
  - an initial value for an attribute, `false`.

Software Engineering Observation 5.6

Early in the design process, classes often lack attributes (and operations). Such classes should not be eliminated, however, because attributes (and operations) may become evident in the later phases of design and implementation.

6.10 Software Engineering Case Study: Identifying Objects’ States and Activities in the ATM System

- Each object in a system goes through a series of discrete states.

- An object’s state is indicated by the values of its attributes at that time.

- **State machine diagrams** model key states of an object and show under what circumstances the object changes state.

- Figure 6.26 is a simple state machine diagram that models two of the states of an object of class ATM.

Software Engineering Observation 6.5

Software designers do not generally create state machine diagrams showing every possible state and state transition for all attributes—there are simply too many of them. State machine diagrams typically show only the most important or complex states and state transitions.

- An activity diagram models an object’s workflow during application execution.

- The activity diagram in Fig. 6.27 models the actions involved in executing a `BalanceInquiry` transaction.
6.10 Software Engineering Case Study: Identifying Objects’ States and Activities in the ATM System (Cont.)

- Figure 6.28 shows an activity diagram for a withdrawal transaction.
- We assume that a withdrawal object has already been assigned a valid account number.

7.15 Software Engineering Case Study: Identifying Class Operations in the ATM System

- An operation is a service that objects of a class provide to clients of the class.
- The verbs and verb phrases in Fig. 7.20 help us determine the operations of our classes.

<table>
<thead>
<tr>
<th>Class</th>
<th>Verbs and verb phrases</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATM</td>
<td>executes financial transactions</td>
</tr>
<tr>
<td>BalanceInquiry</td>
<td>[none in the requirements document]</td>
</tr>
<tr>
<td>Withdrawal</td>
<td>[none in the requirements document]</td>
</tr>
<tr>
<td>Deposit</td>
<td>[none in the requirements document]</td>
</tr>
</tbody>
</table>
Class | Verbs and verb phrases
---|---
BankDatabase | authenticates a user, retrieves an account balance, credits an account, debits an account
Account | retrieves an account balance, credits a deposit amount to an account, debits a withdrawal amount to an account
Screen | displays a message to the user
Keypad | receives numeric input from the user
CashDispenser | dispenses cash, indicates whether it contains enough cash to satisfy a withdrawal request
DepositSlot | receives a deposit envelope

Modeling Operations

- We place operations in the third compartment of the three transaction classes in the updated class diagram of Fig. 7.21.

- UML operations are implemented as methods in C#.
- The UML represents operations by listing the operation name, followed by a comma-separated list of parameters in parentheses, a colon and the return type.
- Each parameter in the comma-separated parameter list consists of a parameter name, followed by a colon and the parameter type:
Operations of Class `BankDatabase` and `Account`

- Class `BankDatabase` needs an operation that provides an authentication service to the ATM.
- We place the operation `AuthenticateUser` in the third compartment of class `BankDatabase`.
- We add a `ValidatePIN` operation to class `Account`.
- Note that we specify a return type of bool for the `AuthenticateUser` and `ValidatePIN` operations.
- The database acts as an intermediary between the ATM and the account data, preventing unauthorized access.

Identifying and Modeling Operation Parameters

- We identify an operation’s parameters by examining what data the operation requires to perform its assigned task.
- The class diagram in Fig. 7.22 models class `BankDatabase`. 

Fig. 7.22 | Class `BankDatabase` with operation parameters.
7.15 Software Engineering Case Study: Identifying Class Operations in the ATM System (Cont.)

- The class diagram in Fig. 7.23 models the parameters of class `Account`’s operations.

8.14 Software Engineering Case Study: Collaboration Among Objects in the ATM System

- When two objects communicate with each other to accomplish a task, they are said to collaborate.
- A collaboration consists of an object of one class sending a message to an object of another class.
- Messages are sent in C# via method calls.
8.14 Software Engineering Case Study: Collaboration Among Objects in the ATM System (Cont.)

Identifying the Collaborations in a System

• For each action or step described in the requirements document, we decide which objects in our system must interact to achieve the desired result.
• We identify one object as the sending object and another as the receiving object.
• We then select one of the receiving object’s operations that must be invoked by the sending object to produce the proper behavior.

<table>
<thead>
<tr>
<th>An object of class...</th>
<th>sends the message...</th>
<th>to an object of class...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Withdrawal</td>
<td>DisplayMessage</td>
<td>Screen</td>
</tr>
<tr>
<td></td>
<td>GetInput</td>
<td>Keypad</td>
</tr>
<tr>
<td></td>
<td>GetAvailableBalance</td>
<td>BankDatabase</td>
</tr>
<tr>
<td></td>
<td>IsSufficientCashAvailable</td>
<td>CashDispenser</td>
</tr>
<tr>
<td></td>
<td>Debit</td>
<td>BankDatabase</td>
</tr>
<tr>
<td></td>
<td>DispenseCash</td>
<td>CashDispenser</td>
</tr>
<tr>
<td>Deposit</td>
<td>DisplayMessage</td>
<td>Screen</td>
</tr>
<tr>
<td></td>
<td>GetInput</td>
<td>Keypad</td>
</tr>
<tr>
<td></td>
<td>IsDepositEnvelopeReceived</td>
<td>DepositSlot</td>
</tr>
<tr>
<td></td>
<td>Credit</td>
<td>BankDatabase</td>
</tr>
<tr>
<td>BankDatabase</td>
<td>ValidatePIN</td>
<td>Account</td>
</tr>
<tr>
<td></td>
<td>AvailableBalance (get)</td>
<td>Account</td>
</tr>
<tr>
<td></td>
<td>TotalBalance (get)</td>
<td>Account</td>
</tr>
<tr>
<td></td>
<td>Debit</td>
<td>Account</td>
</tr>
<tr>
<td></td>
<td>Credit</td>
<td>Account</td>
</tr>
</tbody>
</table>

Interaction Diagrams

• The UML provides several types of **interaction diagrams** that model the behavior of a system by modeling how objects interact with one another.
• The **communication diagram** emphasizes *which objects* participate in collaborations.
• The **sequence diagram** emphasizes *when* messages are sent between objects.
8.14 Software Engineering Case Study: Collaboration Among Objects in the ATM System (Cont.)

**Communication Diagrams**

- Figure 8.25 shows a communication diagram that models the ATM executing a `BalanceInquiry`.

![Communication diagram of the ATM executing a balance inquiry.]

8.14 Software Engineering Case Study: Collaboration Among Objects in the ATM System (Cont.)

**Sequence of Messages in a Communication Diagram**

- Figure 8.26 shows a communication diagram that models the interactions among objects in the system when an object of class `BalanceInquiry` executes.

![Communication diagram for executing a BalanceInquiry.]

8.14 Software Engineering Case Study: Collaboration Among Objects in the ATM System (Cont.)

- Communicating objects are connected with solid lines.
- Messages are passed between objects along these lines in the direction shown by arrows with filled arrowheads.
- The filled arrow in Fig. 8.25 represents a message—or synchronous call—in the UML and a method call in C#.

- Note that Fig. 8.26 models two additional messages passing from the `BankDatabase` to an `Account` (message 1.1 and message 2.1).
- The `BankDatabase` must send get messages to the `Account`'s `AvailableBalance` and `TotalBalance` properties.
- A message passed within the handling of another message is called a nested message.
- The UML recommends using a decimal numbering scheme to indicate nested messages. If the `BankDatabase` needed to pass a second nested message while processing message 1, it would be numbered 1.2.
Sequence Diagrams

- A sequence diagram helps model the timing of collaborations more clearly.
- Figure 8.27 shows a sequence diagram modeling the sequence of interactions that occur when a withdrawal executes.

8.14 Software Engineering Case Study: Collaboration Among Objects in the ATM System (Cont.)

- The dotted line extending down from an object’s rectangle is that object’s lifeline, which represents the progression of time.
- Actions typically occur along an object’s lifeline in chronological order from top to bottom.
- An activation, shown as a thin vertical rectangle, indicates that an object is executing.

10.22 Software Engineering Case Study: Starting to Program the Classes of the ATM System

Visibility

- Access modifiers determine the visibility, or accessibility, of an object’s attributes and operations to other objects.
- The UML employs visibility markers for modeling the visibility of attributes and operations.
  - Public visibility is indicated by a plus sign (+).
  - A minus sign (−) indicates private visibility.
10.22 Software Engineering Case Study: Starting to Program the Classes of the ATM System (Cont.)

- Figure 10.30 shows our updated class diagram with visibility markers included.

- Figure 10.31 further refines the relationships among classes in the ATM system by adding navigability arrows to the association lines.

- Navigability arrows indicate in which direction an association can be traversed and are based on collaborations.

- Programmers use navigability arrows to help determine which objects need references to other objects.

- Associations in a class diagram that have navigability arrows at both ends or do not have navigability arrows at all indicate bidirectional navigability.

```
// Fig. 10.32: Withdrawal.cs
1 // Class Withdrawal represents an ATM withdrawal transaction
2 public class Withdrawal
3 {
4     // parameterless constructor
5     public Withdrawal()
6     {
7         // constructor body code
8         // constructor body code
9     } // end constructor
10 } // end class Withdrawal
```

Outline

- Withdrawal.cs

Implementing the ATM System from Its UML Design

- We follow these four guidelines for each class, using Withdrawal as an example:
  - Use the name located in the first compartment of a class in a class diagram to declare the class as a public class with an empty parameterless constructor.
  - Class Withdrawal initially yields the code in Fig. 10.32.
• Use the attributes located in the class’s second compartment to declare the instance variables.

• The private attributes accountNumber and amount of class Withdrawal yield the code in Fig. 10.33.

```csharp
public class Withdrawal
{
    // attributes
    private int accountNumber; // account to withdraw funds from
    private decimal amount; // amount to withdraw from account

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

Fig. 10.33 | Incorporating private variables for class Withdrawal based on Figs. 10.30–10.31.

• Use the associations described in the class diagram to declare references to other objects. Fig. 10.34 declares the appropriate references as private instance variables.

```csharp
public class Withdrawal
{
    // attributes
    private int accountNumber; // account to withdraw funds from
    private decimal amount; // amount to withdraw

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

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

Fig. 10.34 | Incorporating private reference handles for the associations of class Withdrawal based on Figs. 10.30 and 10.31. (Part 1 of 2.)

• Use the operations located in the third compartment of Fig. 10.30 to declare the shells of the methods. If we have not yet specified a return type for an operation, we declare the method with return type void. Refer to the class diagrams to declare any necessary parameters.

• Adding the public operation Execute (which has an empty parameter list) in class Withdrawal yields the code in Fig. 10.35.

```csharp
public class Withdrawal
{
    // attributes
    private int accountNumber; // account to withdraw funds from
    private decimal amount; // amount to withdraw

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

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

Fig. 10.35 | C# code incorporating method Execute in class Withdrawal based on Figs. 10.30 and 10.31. (Part 1 of 2.)
12.9 Software Engineering Case Study: Incorporating Inheritance and Polymorphism into the ATM System (Cont.)

- The UML specifies a relationship called a generalization to model inheritance.
- Figure 12.20 is the class diagram that models the inheritance relationship between base class Transaction and its three derived classes.

- As Fig. 12.19 shows, classes BalanceInquiry, Withdrawal and Deposit share private int attribute accountNumber.
- Because the derived classes in this case do not need to modify attribute accountNumber, we have chosen to replace private attribute accountNumber in our model with the public read-only property AccountNumber.
- Since this is a read-only property, it provides only a get accessor to access the account number.
- We declare Execute as an abstract operation in base class Transaction—it will become an abstract method in the C# implementation.

Software Engineering Observation 10.13

- Many UML modeling tools can convert UML-based designs into C# code, considerably speeding up the implementation process. For more information on these “automatic” code generators, refer to the web resources listed at the end of Section 3.10.

12.9 Software Engineering Case Study: Incorporating Inheritance and Polymorphism into the ATM System

- The arrows with triangular hollow arrowheads indicate that classes BalanceInquiry, Withdrawal and Deposit are derived from class Transaction by inheritance.
  - Class Transaction is said to be a generalization of its derived classes.
  - The derived classes are said to be specializations of class Transaction.
12.9 Software Engineering Case Study: Incorporating Inheritance and Polymorphism into the ATM System (Cont.)

- 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.
- Because a Withdrawal is a type of Transaction, we no longer draw an association line directly between class ATM and class Withdrawal—derived class Withdrawal inherits base class Transaction’s association with class ATM.
- Derived classes BalanceInquiry and Deposit also inherit this association, which replaces the previously omitted associations between classes BalanceInquiry and Deposit, and class ATM.

- • Figure 12.21 presents an updated class diagram of our model that incorporates inheritance and introduces abstract base class Transaction.

- We also add an association between Transaction and BankDatabase (Fig. 12.21). All Transactions require a reference to the BankDatabase so that they can access and modify account information.
- We include an association between class Transaction and the Screen because all Transactions display output to the user via the Screen.
- Class Withdrawal still participates in associations with the CashDispenser and the Keypad.
12.9 Software Engineering Case Study: Incorporating Inheritance and Polymorphism into the ATM System (Cont.)

• We present a modified class diagram in Fig. 12.22 that includes abstract base class Transaction.

Software Engineering Observation 12.12

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, operations and associations is substantial, a good practice is to divide this information between two class diagrams—one focusing on associations and the other on attributes and operations.

Implementing the ATM System Design Incorporating Inheritance

• If a class A is a generalization of class B, then class B is derived from (and is a specialization of) class A.
• Figure 12.23 contains the shell of class Withdrawal, in which the class definition indicates the inheritance relationship between Withdrawal and Transaction.

Outline

• Withdrawal.cs

Withdrawal.cs

1 // Fig. 12.24: Withdrawal.cs
2 // Class Withdrawal represents an ATM withdrawal transaction.
3 public class Withdrawal : Transaction
4 {
5     // attributes
6     private decimal amount; // amount to withdraw
7     private Keypad keypad; // reference to keypad
8     private CashDispenser cashDispenser; // reference to cash dispenser
9     // parameterless constructor
10     public Withdrawal()
11     {
12         // constructor body code
13     } // end constructor
14 } // end class Withdrawal
15
16 } // end class Withdrawal

• Figure 12.24 contains the portions of the C# code for class Withdrawal.