Chapter Twelve:
Object-Oriented Design
Chapter Goals

- To learn about the software life cycle
- To learn how to discover new classes and methods
- To understand the use of CRC cards for class discovery
- To be able to identify inheritance, aggregation, and dependency relationships between classes
- To master the use of UML class diagrams to describe class relationships
- To learn how to use object-oriented design to build complex programs
The Software Life Cycle

• Encompasses all activities from initial analysis until obsolescence

• Formal process for software development
  • Describes phases of the development process
  • Gives guidelines for how to carry out the phases

• Development process
  • Analysis
  • Design
  • Implementation
  • Testing
  • Deployment
Analysis

• Decide what the project is suppose to do
• Do not think about how the program will accomplish tasks
• Output: requirements document
  • Describes what program will do once completed
  • User manual: tells how user will operate program
  • Performance criteria
Design

• Plan how to implement the system
• Discover structures that underlie problem to be solved
• Decide what classes and methods you need

Output:
• *Description of classes and methods*
• *Diagrams showing the relationships among the classes*
Implementation

• Write and compile the code
• Code implements classes and methods discovered in the design phase
• Output: completed program
Testing

• Run tests to verify the program works correctly
• Output: a report of the tests and their results
Deployment

- Users install program
- Users use program for its intended purpose
The Waterfall Model

- Sequential process of analysis, design, implementation, testing, and deployment
- When rigidly applied, waterfall model did not work
The Spiral Model

• Breaks development process down into multiple phases
• Early phases focus on the construction of prototypes
• Lessons learned from development of one prototype can be applied to the next iteration
• Problem: can lead to many iterations, and process can take too long to complete
The Spiral Model (cont.)

Figure 2  A Spiral Model
Activity Levels in the Rational Unified Process

- Development process methodology by the inventors of UML

**Figure 3** Activity Levels in the Rational Unified Process Methodology
Extreme Programming

- Strives for simplicity
- Removes formal structure
- Focuses on best practices
  - Realistic planning
  - Small releases
  - Metaphor
  - Simplicity
  - Testing
  - Refactoring
  - Pair programming
  - Collective ownership
  - Continuous integration
  - 40-hour week
  - On-site customer
  - Coding standards
Extreme Programming

Realistic planning

- Customers make business decisions
- Programmers make technical decisions
- Update plan when it conflicts with reality

- Small releases
  - Release a useful system quickly
  - Release updates on a very short cycle

- Metaphor
  - Programmers have a simple shared story that explains the system
Extreme Programming

• Simplicity
  • Design as simply as possible instead of preparing for future complexities

• Testing
  • Programmers and customers write test cases
  • Test continuously

• Refactoring
  • Restructure the system continuously to improve code and eliminate duplication
Extreme Programming

• Pair programming
  • Two programmers write code on the same computer

• Collective ownership
  • All programmers can change all code as needed

• Continuous integration
  • Build the entire system and test it whenever a task is complete
Extreme Programming

- 40-hour week
  - Don't cover up unrealistic schedules with heroic effort

- On-site customer
  - A customer is accessible to the programming team at all times

- Coding standards
  - Follow standards that emphasize self-documenting code
Suppose you sign a contract, promising that you will, for an agreed-upon price, design, implement, and test a software package exactly as it has been specified in a requirements document. What is the primary risk you and your customer are facing with this business arrangement?

**Answer:** It is unlikely that the customer did a perfect job with the requirements document. If you don't accommodate changes, your customer may not like the outcome. If you charge for the changes, your customer may not like the cost.
Self Check 12.2

Does Extreme Programming follow a waterfall or a spiral model?

**Answer:** An "extreme" spiral model, with lots of iterations.
What is the purpose of the "on-site customer" in Extreme Programming?

**Answer:** To give frequent feedback as to whether the current iteration of the product fits customer needs.
Object-Oriented Design

• Discover classes
• Determine responsibilities of each class
• Describe relationships between the classes
Discovering Classes

- A class represents some useful concept
- Concrete entities: bank accounts, ellipses, and products
- Abstract concepts: streams and windows
- Find classes by looking for nouns in the task description
- Define the behavior for each class
- Find methods by looking for verbs in the task description
### Example: Invoice

**INVOICE**

Sam's Small Appliances  
100 Main Street  
Anytown, CA 98765

<table>
<thead>
<tr>
<th>Item</th>
<th>Qty</th>
<th>Price</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Toaster</td>
<td>3</td>
<td>$29.95</td>
<td>$89.85</td>
</tr>
<tr>
<td>Hair Dryer</td>
<td>1</td>
<td>$24.95</td>
<td>$24.95</td>
</tr>
<tr>
<td>Car Vacuum</td>
<td>2</td>
<td>$19.99</td>
<td>$39.98</td>
</tr>
</tbody>
</table>

**AMOUNT DUE:** $154.78

**Figure 4** An Invoice
Example: Invoice

- Classes that come to mind: Invoice, LineItem, and Customer
- Good idea to keep a list of candidate classes
- Brainstorm, simply put all ideas for classes onto the list
- You can cross not useful ones later
Finding Classes

• Keep the following points in mind:
  • Class represents set of objects with the same behavior
    o Entities with multiple occurrences in problem description are good candidates for objects
    o Find out what they have in common
    o Design classes to capture commonalities
  • Represent some entities as objects, others as primitive types
    o Should we make a class Address or use a String?
  • Not all classes can be discovered in analysis phase
  • Some classes may already exist
CRC Card

- Describes a class, its responsibilities, and its collaborators
- Use an index card for each class
- Pick the class that should be responsible for each method (verb)
- Write the responsibility onto the class card
- Indicate what other classes are needed to fulfill responsibility (collaborators)

Continued
Figure 5  A CRC Card
Self Check 12.4

Suppose the invoice is to be saved to a file. Name a likely collaborator.

**Answer:** FileWriter
Self Check 12.5

Looking at the invoice in Figure 4, what is a likely responsibility of the Customer class?

**Answer:** To produce the shipping address of the customer.
What do you do if a CRC card has ten responsibilities?

**Answer:** Reword the responsibilities so that they are at a higher level, or come up with more classes to handle the responsibilities.
Relationships Between Classes

- Inheritance
- Aggregation
- Dependency
Inheritance

• *Is-a* relationship

• Relationship between a more general class (superclass) and a more specialized class (subclass)

• Every savings account is a bank account

• Every circle is an ellipse (with equal width and height)

• It is sometimes abused
  
  • *Should the class Tire be a subclass of a class Circle?*
    
    • The *has-a* relationship would be more appropriate
Aggregation

• *Has-a* relationship

• Objects of one class contain references to objects of another class

• Use an instance variable
  • *A tire has a circle as its boundary:*

```java
class Tire
{
  . . .
  private String rating;
  private Circle boundary;
}
```

• Every car has a tire (in fact, it has four)
Example

class Car extends Vehicle {
    . . .
    private Tire[] tires;
}

Figure 6
UML Notation for Inheritance and Aggregation
Dependency

- *Uses* relationship
- Example: many of our applications depend on the `Scanner` class to read input
- Aggregation is a stronger form of dependency
- Use aggregation to remember another object between method calls
## UML Relationship Symbols

<table>
<thead>
<tr>
<th>Relationship</th>
<th>Symbol</th>
<th>Line Style</th>
<th>Arrow Tip</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inheritance</td>
<td><img src="symbol" alt="Solid Line with Triangle Arrow Tip" /></td>
<td>Solid</td>
<td>Triangle</td>
</tr>
<tr>
<td>Interface Implementation</td>
<td><img src="symbol" alt="Dotted Line with Triangle Arrow Tip" /></td>
<td>Dotted</td>
<td>Triangle</td>
</tr>
<tr>
<td>Aggregation</td>
<td><img src="symbol" alt="Solid Line with Diamond Arrow Tip" /></td>
<td>Solid</td>
<td>Diamond</td>
</tr>
<tr>
<td>Dependency</td>
<td><img src="symbol" alt="Dotted Line with Open Arrow Tip" /></td>
<td>Dotted</td>
<td>Open</td>
</tr>
</tbody>
</table>
Self Check 12.7

Consider the Bank and BankAccount classes of Chapter 7. How are they related?

**Answer:** Through aggregation. The bank manages bank account objects.
Self Check 12.8

Consider the `BankAccount` and `SavingsAccount` objects of Chapter 10. How are they related?

Answer: Through inheritance.
Consider the BankAccountTester class of Chapter 3. Which classes does it depend on?

**Answer:** The BankAccount, System, and PrintStream classes.
Advanced Topic: Attributes and Methods in UML Diagrams

BankAccount

Attributes

balance

Methods
deposit()
withdraw()
Advanced Topic: Multiplicities

- any number (zero or more): *
- one or more: 1..*
- zero or one: 0..1
- exactly one: 1
Advanced Topic: Aggregation and Association

• Association: more general relationship between classes
• Use early in the design phase
• A class is associated with another if you can navigate from objects of one class to objects of the other
• Given a `Bank` object, you can navigate to `Customer` objects
Five-Part Development Process

1. Gather requirements
2. Use CRC cards to find classes, responsibilities, and collaborators
3. Use UML diagrams to record class relationships
4. Use javadoc to document method behavior
5. Implement your program
Printing an Invoice – Requirements

• Task: print out an invoice

• Invoice: describes the charges for a set of products in certain quantities

• Omit complexities
  • *Dates, taxes, and invoice and customer numbers*

• Print invoice
  • *Billing address, all line items, amount due*

• Line item
  • *Description, unit price, quantity ordered, total price*

• For simplicity, do not provide a user interface

• Test program: adds line items to the invoice and then prints it
## Sample Invoice

### INVOICE

Sam's Small Appliances  
100 Main Street  
Anytown, CA 98765

<table>
<thead>
<tr>
<th>Description</th>
<th>Price</th>
<th>Qty</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Toaster</td>
<td>29.95</td>
<td>3</td>
<td>89.85</td>
</tr>
<tr>
<td>Hair dryer</td>
<td>24.95</td>
<td>1</td>
<td>24.95</td>
</tr>
<tr>
<td>Car vacuum</td>
<td>19.99</td>
<td>2</td>
<td>39.98</td>
</tr>
</tbody>
</table>

**AMOUNT DUE: $154.78**
Printing an Invoice – CRC Cards

• Discover classes

• Nouns are possible classes

Invoice
Address
LineItem
Product
Description
Price
Quantity
Total
Amount Due
• Analyze classes

Invoice
Address
LineItem // Records the product and the quantity
Product
Description // Field of the Product class
Price // Field of the Product class
Quantity // Not an attribute of a Product
Total // Computed – not stored anywhere
Amount Due // Computed – not stored anywhere

• Classes after a process of elimination

Invoice
Address
LineItem
Product
CRC Cards for Printing Invoice

Invoice and Address must be able to format themselves:

<table>
<thead>
<tr>
<th>Invoice</th>
</tr>
</thead>
<tbody>
<tr>
<td>format the invoice</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>format the address</td>
</tr>
</tbody>
</table>
CRC Cards for Printing Invoice

Add collaborators to invoice card:

<table>
<thead>
<tr>
<th>Invoice</th>
</tr>
</thead>
<tbody>
<tr>
<td>format the invoice</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Big Java by Cay Horstmann
Copyright © 2008 by John Wiley & Sons. All rights reserved.
## CRC Cards for Printing Invoice

Product and LineItem CRC cards:

<table>
<thead>
<tr>
<th>Product</th>
<th>LineItem</th>
</tr>
</thead>
<tbody>
<tr>
<td>get description</td>
<td>format the item</td>
</tr>
<tr>
<td>get unit price</td>
<td>Product</td>
</tr>
<tr>
<td></td>
<td>get total price</td>
</tr>
</tbody>
</table>
CRC Cards for Printing Invoice

Invoice must be populated with products and quantities:

<table>
<thead>
<tr>
<th>format the invoice</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>add a product and quantity</td>
<td>LineItem</td>
</tr>
<tr>
<td></td>
<td>Product</td>
</tr>
</tbody>
</table>

Big Java by Cay Horstmann
Copyright © 2008 by John Wiley & Sons. All rights reserved.
Printing an Invoice – UML Diagrams

Figure 7  The Relationships Between the Invoice Classes
Printing an Invoice – Method Documentation

• Use `javadoc` documentation to record the behavior of the classes

• Leave the body of the methods blank

• Run `javadoc` to obtain formatted version of documentation in HTML format

• Advantages:
  • *Share HTML documentation with other team members*
  • *Format is immediately useful: Java source files*
  • *Supply the comments of the key methods*
Method Documentation – Invoice class

/**
 * Describes an invoice for a set of purchased products.
 */
public class Invoice
{
    /**
     * Adds a charge for a product to this invoice.
     * @param aProduct the product that the customer ordered
     * @param quantity the quantity of the product
     */
    public void add(Product aProduct, int quantity)
    {
    }
}
/**
   * Formats the invoice.
   * @return the formatted invoice
   */

public String format()
{
}
}
/**
 * Describes a quantity of an article to purchase and its price.
 */

public class LineItem
{
    /**
     * Computes the total cost of this line item.
     * @return the total price
     */
    public double getTotalPrice()
    {
    }
}
/** Formats this item.
   * @return a formatted string of this line item
   */

public String format()
{
}
/**
   * Describes a product with a description and a price.
   * 
   * public class Product
   *
   */
    public String getgetDescription()
    {
    }
/**
   * Gets the product price.
   * @return the unit price
   */
*/

public double getPrice()
{
}
}
/**
 * Describes a mailing address.
 */

public class Address {

   /**
    * Formats the address.
    * @return the address as a string with three lines
    */
   public String format()
   {
      return address;
   }

}
The Class Documentation in the HTML Format

Figure 8  The Class Documentation in HTML Format
Printing an Invoice – Implementation

• The UML diagram will give instance variables

• Look for associated classes
  • *They yield instance variables*
Implementation

• Invoice aggregates Address and LineItem

• Every invoice has one billing address

• An invoice can have many line items:

```java
public class Invoice {
    ...
    private Address billingAddress;
    private ArrayList<LineItem> items;
}
```
A line item needs to store a Product object and quantity:

```java
public class LineItem {
    private int quantity;
    private Product theProduct;
}
```
Implementation

• The methods themselves are now very easy

• Example:
  • `getTotalPrice` of `LineItem` gets the unit price of the product and multiplies it with the quantity

```java
/**
   * Computes the total cost of this line item.
   * @return the total price
   */
public double getTotalPrice()
{
    return theProduct.getPrice() * quantity;
}
```
ch12/invoice/InvoicePrinter.java

```java
/**
 * This program demonstrates the invoice classes by printing a sample invoice.
 */

public class InvoicePrinter {
    public static void main(String[] args) {
        Address samsAddress = new Address("Sam's Small Appliances",
                                           "100 Main Street", "Anytown", "CA", "98765");

        Invoice samsInvoice = new Invoice(samsAddress);
        samsInvoice.add(new Product("Toaster", 29.95), 3);
        samsInvoice.add(new Product("Hair dryer", 24.95), 1);
        samsInvoice.add(new Product("Car vacuum", 19.99), 2);

        System.out.println(samsInvoice.format());
    }
}
```

import java.util.ArrayList;

/**
 * Describes an invoice for a set of purchased products.
 */

public class Invoice {

    /**
     * Constructs an invoice.
     * @param anAddress the billing address
     */
    public Invoice(Address anAddress) {
        items = new ArrayList<LineItem>();
        billingAddress = anAddress;
    }

    /**
     * Adds a charge for a product to this invoice.
     * @param aProduct the product that the customer ordered
     * @param quantity the quantity of the product
     */

    ...Continued
public void add(Product aProduct, int quantity) {
    LineItem anItem = new LineItem(aProduct, quantity);
    items.add(anItem);
}

/**
 * Formats the invoice.
 * @return the formatted invoice
 */
public String format() {
    String r = " INVOICE
    + billingAddress.format()
    + String.format("\n\n%-30s%8s%5s%8s\n", "Description", "Price", "Qty", "Total");
    for (LineItem i : items) {
        r = r + i.format() + "\n";
    }
    return r;
}
```java
r = r + String.format("\nAMOUNT DUE: $%8.2f", getAmountDue());
return r;
/**
 * Computes the total amount due.
 * @return the amount due
 */
public double getAmountDue()
{
    double amountDue = 0;
    for (LineItem i : items)
    {
        amountDue = amountDue + i.getTotalPrice();
    }
    return amountDue;
private Address billingAddress;
private ArrayList<LineItem> items;
```
ch12/invoice/LineItem.java

01: /**
02:   Describes a quantity of an article to purchase.
03: */
04: public class LineItem
05: {
06:   /**
07:     Constructs an item from the product and quantity.
08:     @param aProduct the product
09:     @param aQuantity the item quantity
10:   */
11:   public LineItem(Product aProduct, int aQuantity)
12:   {
13:     theProduct = aProduct;
14:     quantity = aQuantity;
15:   }
16:
17:   /**
18:     Computes the total cost of this line item.
19:     @return the total price
20:   */
21:   public double getTotalPrice()
22:   {
23:       return theProduct.getPrice() * quantity;
24:   }
25:
26:   /**
27:     * Formats this item.
28:     * @return a formatted string of this item
29:     */
30:   public String format()
31:   {
32:       return String.format("%-30s%8.2f%5d%8.2f",
33:           theProduct.getDescription(), theProduct.getPrice(),
34:           quantity, getTotalPrice());
35:   }
36:
37:   private int quantity;
38:   private Product theProduct;
public class Product {

    public Product(String aDescription, double aPrice) {
        description = aDescription;
        price = aPrice;
    }

    /**
     * Gets the product description.
     * @return the description
     */

    public String getDescription() {
        return description;
    }

    public double getPrice() {
        return price;
    }

    public void setDescription(String description) {
        this.description = description;
    }

    public void setPrice(double price) {
        this.price = price;
    }

}
public String getDescription()
{
    return description;
}

/**
 * Gets the product price.
 * @return the unit price
 */
public double getPrice()
{
    return price;
}

private String description;
private double price;
Describes a mailing address.

public class Address {

    Constructs a mailing address.
    @param aName the recipient name
    @param aStreet the street
    @param aCity the city
    @param aState the two-letter state code
    @param aZip the ZIP postal code
    *

    public Address(String aName, String aStreet,
                   String aCity, String aState, String aZip)
    {
        name = aName;
        street = aStreet;
        city = aCity;
        state = aState;
        zip = aZip;
    }

    // Continued

23:     /**
24:      * Formats the address.
25:      * @return the address as a string with three lines
26:      */
27:     public String format()
28:     {
29:         return name + "\n" + street + "\n"
30:             + city + ", " + state + " " + zip;
31:     }
32: 
33:     private String name;
34:     private String street;
35:     private String city;
36:     private String state;
37:     private String zip;
38: }
Self Check 12.10

Which class is responsible for computing the amount due? What are its collaborators for this task?

**Answer:** The *Invoice* class is responsible for computing the amount due. It collaborates with the *LineItem* class.
Self Check 12.11

Why do the format methods return `String` objects instead of directly printing to `System.out`?

**Answer:** This design decision reduces coupling. It enables us to reuse the classes when we want to show the invoice in a dialog box or on a web page.
An Automatic Teller Machine – Requirements

• ATM is used by bank customers. A customer has a
  • Checking account
  • Savings account
  • Customer number
  • PIN
An Automatic Teller Machine – Requirements

- Customers can select an account
- The balance of the selected account is displayed
- Then, customer can deposit and withdraw money
- Process is repeated until the customer chooses to exit
An Automatic Teller Machine – Requirements

- GUI Interface
  - Keypad
  - Display
  - Buttons A, B, C
  - Buttons function depend on the state of the machine

![Graphical User Interface for the Automatic Teller Machine](image_url)
An Automatic Teller Machine – Requirements

• At start up the customer is expected to
  • *Enter customer number*
  • *Press the A button*
  • *The display shows:*

  
  Enter Customer Number
  A = OK
An Automatic Teller Machine – Requirements

• The customer is expected to
  • Enter a PIN
  • Press A button
  • The display shows:

    Enter PIN
    A = OK
An Automatic Teller Machine – Requirements

• Search for the customer number and PIN
  • *If it matches a bank customer, proceed*
  • *Else return to start up screen*
An Automatic Teller Machine – Requirements

• If the customer is authorized
  • The display shows:

    Select Account
    A = Checking
    B = Savings
    C = Exit
An Automatic Teller Machine – Requirements

• If the user presses C
  • *The ATM reverts to its original state*
  • *ATM asks next user to enter a customer number*

• If the user presses A or B
  • *The ATM remembers selected account*
  • *The display shows:*

  \[
  \text{Balance} = \text{balance of selected account} \\
  \text{Enter amount and select transaction} \\
  A = \text{Withdraw} \\
  B = \text{Deposit} \\
  C = \text{Cancel}
  \]
An Automatic Teller Machine – Requirements

• If the user presses A or B
  • The value entered is withdrawn or deposited
  • Simulation: no money is dispensed and no deposit is accepted
  • The ATM reverts to previous state

• If the user presses C
  • The ATM reverts to previous state
An Automatic Teller Machine – Requirements

• Text-based interaction
  • *Read input from System.in instead of the buttons*
  • *Here is a typical dialog:*

```
Enter account number: 1
Enter PIN: 1234
A=Checking, B=Savings, C=Quit: A
Balance=0.0
A=Deposit, B=Withdrawal, C=Cancel: A
Amount: 1000
A=Checking, B=Savings, C=Quit: C
```
Nouns are possible classes

ATM
User
Keypad
Display
Display message
Button
State
Bank account
Checking account
Savings account
Customer
Customer number
PIN
Bank
CRC Cards for Automatic Teller Machine

Customer

- get accounts
- match number and PIN

Bank

- find customer
- read customers

Copyright © 2008 by John Wiley & Sons. All rights reserved.
 CRC Cards for Automatic Teller Machine (cont.)

<table>
<thead>
<tr>
<th>Action</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>manage state</td>
<td>Customer</td>
</tr>
<tr>
<td>select customer</td>
<td>Bank</td>
</tr>
<tr>
<td>select account</td>
<td>BankAccount</td>
</tr>
<tr>
<td>execute transaction</td>
<td></td>
</tr>
</tbody>
</table>

*Big Java* by Cay Horstmann
Copyright © 2008 by John Wiley & Sons. All rights reserved.
ATM States

1. START: Enter customer ID
2. PIN: Enter PIN
3. ACCOUNT: Select account
4. TRANSACT: Select transaction
State Diagram for ATM Class

START

Customer number entered

PIN

Customer not found

Exit selected

ACCOUNT

Customer found

Account selected

TRANSACT

Transaction completed or canceled

Figure 10  State Diagram for the ATM Class
An Automatic Teller Machine – UML Diagrams

Figure 11  Relationships Between the ATM Classes
An ATM that accesses a bank.

```
public class ATM {
    /**
     * Constructs an ATM for a given bank.
     * @param aBank the bank to which this ATM connects
     */
    public ATM(Bank aBank) { }
    /**
     * Sets the current customer number
     * and sets state to PIN.
     * (Precondition: state is START)
     * @param number the customer number
     */
    public void setCustomerNumber(int number) { }
}
```
/**
 * Finds customer in bank.
 * If found sets state to ACCOUNT, else to START.
 * (Precondition: state is PIN)
 * @param pin the PIN of the current customer
 */
 public void selectCustomer(int pin) {}
public void selectAccount(int account) { }
/**
 * Withdraws amount from current account.
 * (Precondition: state is TRANSACT)
 * @param value the amount to withdraw
 */
public void withdraw(double value) { }
...
An Automatic Teller Machine – Implementation

• Start implementation with classes that don't depend on others
  • Keypad
  • BankAccount

• Then implement Customer which depends only on BankAccount

• This bottom-up approach allows you to test your classes individually
An Automatic Teller Machine – Implementation

• Aggregated classes in UML diagram give instance variables

```java
public class ATM {
    ...
    private Bank theBank;
}
```

• From description of ATM states, it is clear that we require additional instance variables:

```java
public class ATM {
    ...
    private int state;
    private Customer currentCustomer;
    private BankAccount currentAccount;
}
```
• Most methods are very straightforward to implement

• Consider `selectCustomer`:
  ```java
  /**
   * Finds customer in bank.
   * If found sets state to ACCOUNT, else to START.
   * (Precondition: state is PIN)
   * @param pin the PIN of the current customer
   */
  ```
An Automatic Teller Machine – Implementation

• Description can be almost literally translated to Java instructions:

```java
public void selectCustomer(int pin)
{
    assert state == PIN;
    currentCustomer = theBank.findCustomer(customerNumber, pin);
    if (currentCustomer == null)
        state = START;
    else
        state = ACCOUNT;
}
```
/**
 * An ATM that accesses a bank.
 */
public class ATM {
    /**
     * Constructs an ATM for a given bank.
     * @param aBank the bank to which this ATM connects
     */
    public ATM(Bank aBank) {
        theBank = aBank;
        reset();
    }

    /**
     * Resets the ATM to the initial state.
     */
    public void reset() {

Continued
ch12/atm/ATM.java (cont.)

```java
021:   customerNumber = -1;
022:   currentAccount = null;
023:   state = START;
024: }
025: }
026: /**
027:   Sets the current customer number
028:   and sets state to PIN.
029:   (Precondition: state is START)
030:   @param number the customer number.
031: */
032: public void setCustomerNumber(int number)
033: {
034:     assert state == START;
035:     customerNumber = number;
036:     state = PIN;
037: }
038: }
039: /**
040:   Finds customer in bank.
041:   If found sets state to ACCOUNT, else to START.

Continued
```

*Big Java* by Cay Horstmann
Copyright © 2008 by John Wiley & Sons. All rights reserved.
042:     (Precondition: state is PIN)
043:     @param pin the PIN of the current customer
044:     */
045:     public void selectCustomer(int pin)
046:     {
047:         assert state == PIN;
048:         currentCustomer
049:             = theBank.findCustomer(customerNumber, pin);
050:         if (currentCustomer == null)
051:             state = START;
052:         else
053:             state = ACCOUNT;
054:     }
055: 
056:     /**
057:         Sets current account to checking or savings. Sets
058:         state to TRANSACT.
059:         (Precondition: state is ACCOUNT or TRANSACT)
060:         @param account one of CHECKING or SAVINGS
061:     */

Continued
public void selectAccount(int account)
{
    assert state == ACCOUNT || state == TRANSACT;
    if (account == CHECKING)
    {
        currentAccount = currentCustomer.getCheckingAccount();
    }
    else
    {
        currentAccount = currentCustomer.getSavingsAccount();
    }
    state = TRANSACT;
}

/**
Withdraws amount from current account.
(Precondition: state is TRANSACT)
@param value the amount to withdraw
*/
public void withdraw(double value)
{
    assert state == TRANSACT;
    currentAccount.withdraw(value);
}
public void deposit(double value) {
    assert state == TRANSACT;
    currentAccount.deposit(value);
}

/**
 * Gets the balance of the current account.
 * (Precondition: state is TRANSACT)
 * @return the balance
 */
public double getBalance() {
    assert state == TRANSACT;
    return currentAccount.getBalance();
}
ch12/atm/ATM.java  (cont.)

104:     /**
105:     **/
106:     /**
107:     * Moves back to the previous state.
108:     */
109:     public void back()
110:     {
111:         if (state == TRANSACT)
112:             state = ACCOUNT;
113:         else if (state == ACCOUNT)
114:             state = PIN;
115:         else if (state == PIN)
116:             state = START;
117:     }
118:     /**
119:     * Gets the current state of this ATM.
120:     * @return the current state
121:     */
122:     public int getState()
123:     {
124:         return state;
125:     }

Continued
ch12/atm/ATM.java (cont.)

126:
127:    private int state;
128:    private int customerNumber;
129:    private Customer currentCustomer;
130:    private BankAccount currentAccount;
131:    private Bank theBank;
132:
133:    public static final int START = 1;
134:    public static final int PIN = 2;
135:    public static final int ACCOUNT = 3;
136:    public static final int TRANSACT = 4;
137:
138:    public static final int CHECKING = 1;
139:    public static final int SAVINGS = 2;
140: }
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Scanner;

/**
 * A bank contains customers with bank accounts.
 *
 */
public class Bank {
    /**
     * Constructs a bank with no customers.
     *
     */
    public Bank() {
        customers = new ArrayList<Customer>();
    }

    /**
     * Reads the customer numbers and pins and initializes the bank accounts.
     * @param filename the name of the customer file
     */
    Continued
public void readCustomers(String filename) throws IOException {
    Scanner in = new Scanner(new FileReader(filename));
    while (in.hasNext()) {
        int number = in.nextInt();
        int pin = in.nextInt();
        Customer c = new Customer(number, pin);
        addCustomer(c);
    }
    in.close();
}

/**
 * Adds a customer to the bank.
 * @param c the customer to add
 */
public void addCustomer(Customer c) {
    customers.add(c);
}

Continued
 /**
* Finds a customer in the bank.
* @param aNumber a customer number
* @param aPin a personal identification number
* @return the matching customer, or null if no customer matches
*/
public Customer findCustomer(int aNumber, int aPin)
{
    for (Customer c : customers)
    {
        if (c.match(aNumber, aPin))
            return c;
    }
    return null;
}

private ArrayList<Customer> customers;
/**
 * A bank customer with a checking and a savings account.
 */

class Customer {
    /**
     * Constructs a customer with a given number and PIN.
     * @param aNumber the customer number
     * @param aPin the personal identification number
     */
    public Customer(int aNumber, int aPin) {
        customerNumber = aNumber;
        pin = aPin;
        checkingAccount = new BankAccount();
        savingsAccount = new BankAccount();
    }

    /**
     * Tests if this customer matches a customer number and PIN.
     * @param aNumber a customer number
     * @param aPin a personal identification number
     */
}
@return true if the customer number and PIN match
*/
public boolean match(int aNumber, int aPin)
{
    return customerNumber == aNumber && pin == aPin;
}

/**
 * Gets the checking account of this customer.
 * @return the checking account
 */
public BankAccount getCheckingAccount()
{
    return checkingAccount;
}

/**
 * Gets the savings account of this customer.
 * @return the checking account
 */
public BankAccount getSavingsAccount()
return savingsAccount;
}

private int customerNumber;
private int pin;
private BankAccount checkingAccount;
private BankAccount savingsAccount;
import java.io.IOException;
import java.util.Scanner;

/**
 * A text-based simulation of an automatic teller machine.
 */
public class ATMSimulator {
    public static void main(String[] args) {
        ATM theATM;
        try {
            Bank theBank = new Bank();
            theBank.readCustomers("customers.txt");
            theATM = new ATM(theBank);
        } catch (IOException e) {
            System.out.println("Error opening accounts file.");
            return;
        }
    }
}
Scanner in = new Scanner(System.in);

while (true)
{
    int state = theATM.getState();
    if (state == ATM.START)
    {
        System.out.print("Enter customer number: ");
        int number = in.nextInt();
        theATM.setCustomerNumber(number);
    }
    else if (state == ATM.PIN)
    {
        System.out.print("Enter PIN: ");
        int pin = in.nextInt();
        theATM.selectCustomer(pin);
    }
    else if (state == ATM.ACCOUNT)
    {
        System.out.print("A=Checking, B=Savings, C=Quit: ");
        String command = in.next();
    }
45:     if (command.equalsIgnoreCase("A"))
46:         theATM.selectAccount(ATM.CHECKING);
47:     else if (command.equalsIgnoreCase("B"))
48:         theATM.selectAccount(ATM.SAVINGS);
49:     else if (command.equalsIgnoreCase("C"))
50:         theATM.reset();
51:     else
52:         System.out.println("Illegal input!");
53: }
54: else if (state == ATM.TRANSACT)
55: {
56:     System.out.println("Balance=" + theATM.getBalance());
57:     System.out.print("A=Deposit, B=Withdrawal, C=Cancel: ");
58:     String command = in.next();
59:     if (command.equalsIgnoreCase("A"))
60:         {
61:             System.out.print("Amount: ");
62:             double amount = in.nextDouble();
63:             theATM.deposit(amount);
64:             theATM.back();
65:         }
else if (command.equalsIgnoreCase("B"))
{
    System.out.print("Amount: ");
    double amount = in.nextDouble();
    theATM.withdraw(amount);
    theATM.back();
}
else if (command.equalsIgnoreCase("C"))
    theATM.back();
else
    System.out.println("Illegal input!");
Enter account number: 1
Enter PIN: 1234
A=Checking, B=Savings, C=Quit: A
Balance=0.0
A=Deposit, B=Withdrawal, C=Cancel: A
Amount: 1000
A=Checking, B=Savings, C=Quit: C

...
import java.io.IOException;
import javax.swing.JFrame;
import javax.swing.JOptionPane;

/**
 * A graphical simulation of an automatic teller machine.
 */
public class ATMViewer {
    public static void main(String[] args) {
        ATM theATM;
        try {
            Bank theBank = new Bank();
            theBank.readCustomers("customers.txt");
            theATM = new ATM(theBank);
        } catch (IOException e) {
            // Continued
        }
    }
}
ch12/atm/ATMViewer.java  (cont.)

```java
22:     JOptionPane.showMessageDialog(null,
23:         "Error opening accounts file.");
24:     return;
25: }
26:
27:     JFrame frame = new ATMFrame(theATM);
28:     frame.setTitle("First National Bank of Java");
29:     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
30:     frame.setVisible(true);
31: }
32: }
33:
```
ch12/atm/ATMFrame.java

001: import java.awt.FlowLayout;
002: import java.awt.GridLayout;
003: import java.awt.event.ActionEvent;
004: import java.awt.event.ActionListener;
005: import javax.swing.JButton;
006: import javax.swing.JFrame;
007: import javax.swing.JPanel;
008: import javax.swing.JTextArea;
009:
010: /**
011:  * A frame displaying the components of an ATM.
012:  */
013: public class ATMFrame extends JFrame
014: {
015:     /**
016:      * Constructs the user interface of the ATM frame.
017:      */
018:     public ATMFrame(ATM anATM)
019:     {
020:         theATM = anATM;
021:
022:         // Construct components

Continued

Big Java by Cay Horstmann
Copyright © 2008 by John Wiley & Sons. All rights reserved.
pad = new KeyPad();
display = new JTextArea(4, 20);

aButton = new JButton("  A  ");
aButton.addActionListener(new AButtonListener());
bButton = new JButton("  B  ");
bButton.addActionListener(new BButtonListener());
cButton = new JButton("  C  ");
cButton.addActionListener(new CButtonListener());

// Add components

buttonPanel = new JPanel();
buttonPanel.add(aButton);
buttonPanel.add(bButton);
buttonPanel.add(cButton);

setLayout(new FlowLayout());
add(pad);
add(display);
047:    add(buttonPanel);
048:    showState();
049:    setSize(FRAME_WIDTH, FRAME_HEIGHT);
050:  }
051:  
052:  /**
053:   * Updates display message.
054:   */
055:  public void showState()
056:  {
057:      int state = theATM.getState();
058:      pad.clear();
059:      if (state == ATM.START)
060:         display.setText("Enter customer number\nA = OK");
061:      else if (state == ATM.PIN)
062:         display.setText("Enter PIN\nA = OK");
063:      else if (state == ATM.ACCOUNT)
064:         display.setText("Select Account\n" + "A = Checking\nB = Savings\nC = Exit");
065:      else if (state == ATM.TRANSACT)
066:         display.setText("Balance = " + theATM.getBalance());
067:  
Continued
Enter amount and select transaction
A = Withdraw
B = Deposit
C = Cancel
}

private class AButtonListener implements ActionListener {
    public void actionPerformed(ActionEvent event) {
        int state = theATM.getState();
        if (state == ATM.START) 
            theATM.setCustomerNumber((int) pad.getValue());
        else if (state == ATM.PIN) 
            theATM.selectCustomer((int) pad.getValue());
        else if (state == ATM.ACCOUNT) 
            theATM.selectAccount(ATM.CHECKING);
        else if (state == ATM.TRANSACT) 
        { 
            theATM.withdraw(pad.getValue());
            theATM.back();
        } 
        showState();
    }
}
private class BButtonListener implements ActionListener {
    public void actionPerformed(ActionEvent event) {
        int state = theATM.getState();
        if (state == ATM.ACCOUNT) {
            theATM.selectAccount(ATM.SAVINGS);
        } else if (state == ATM.TRANSACT) {
            theATM.deposit(pad.getValue());
            theATM.back();
        }
        showState();
    }
}

private class CButtonListener implements ActionListener {
    public void actionPerformed(ActionEvent event) {
        int state = theATM.getState();
        if (state == ATM.ACCOUNT) {
            theATM.reset();
        }
    }
}
```java
else if (state == ATM.TRANSACT)
    theATM.back();
    showState();
```

```java
private JButton aButton;
private JButton bButton;
private JButton cButton;
private KeyPad pad;
private JTextArea display;
private ATM theATM;
private static final int FRAME_WIDTH = 300;
private static final int FRAME_HEIGHT = 400;
```
display = new JTextField();
add(display, "North");

// Make button panel
buttonPanel = new JPanel();
buttonPanel.setLayout(new GridLayout(4, 3));

// Add digit buttons
addButton("7");
addButton("8");
addButton("9");
addButton("4");
addButton("5");
addButton("6");
addButton("1");
addButton("2");
addButton("3");
addButton("0");
addButton(". ");

// Add clear entry button
Continued
File KeyPad.java (cont.)

047:    clearButton = new JButton("CE");
048:    buttonPanel.add(clearButton);
049:    
050:    class ClearButtonListener implements ActionListener
051:    {
052:        public void actionPerformed(ActionEvent event)
053:        {
054:            display.setText("");  
055:        }
056:    }  
057:    ActionListener listener = new ClearButtonListener();
058:    clearButton.addActionListener(new
059:    ClearButtonListener());
060:    
061:    add(buttonPanel, "Center");
062:    
063:    /**
064:     * Adds a button to the button panel
065:     * @param label the button label
066:     */
067: 
068:     
069:     

Continued
private void addButton(final String label) {
    class DigitButtonListener implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            // Don't add two decimal points
            if (label.equals(".") && display.getText().indexOf(".") != -1)
                return;
            // Append label text to button
            display.setText(display.getText() + label);
        }
    } // End DigitButtonListener
    JButton button = new JButton(label);
    buttonPanel.add(button);
    ActionListener listener = new DigitButtonListener();
    button.addActionListener(listener);
} // End addButton
/**
* Gets the value that the user entered.
* @return the value in the text field of the keypad
*/
public double getValue()
{
    return Double.parseDouble(display.getText());
}

/**
* Clears the display.
*/
public void clear()
{
    display.setText("" occupational

private JPanel buttonPanel;
private JButton clearButton;
private JTextField display;
Self Check 12.12

Why does the Bank class in this example not store an array list of bank accounts?

Answer: The bank needs to store the list of customers so that customers can log in. We need to locate all bank accounts of a customer, and we chose to simply store them in the customer class. In this program, there is no further need to access bank accounts.
Suppose the requirements change – you need to save the current account balances to a file after every transaction and reload them when the program starts. What is the impact of this change on the design?

**Answer:** The Bank class needs to have an additional responsibility: to load and save the accounts. The bank can carry out this responsibility because it has access to the customer objects and, through them, to the bank accounts.