

# ICOM 4015: Advanced Programming

## Lecture 12

### **Chapter Twelve: Object-Oriented Design**

CAY HORSTMANN



## Chapter Twelve: Object-Oriented Design

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

## 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

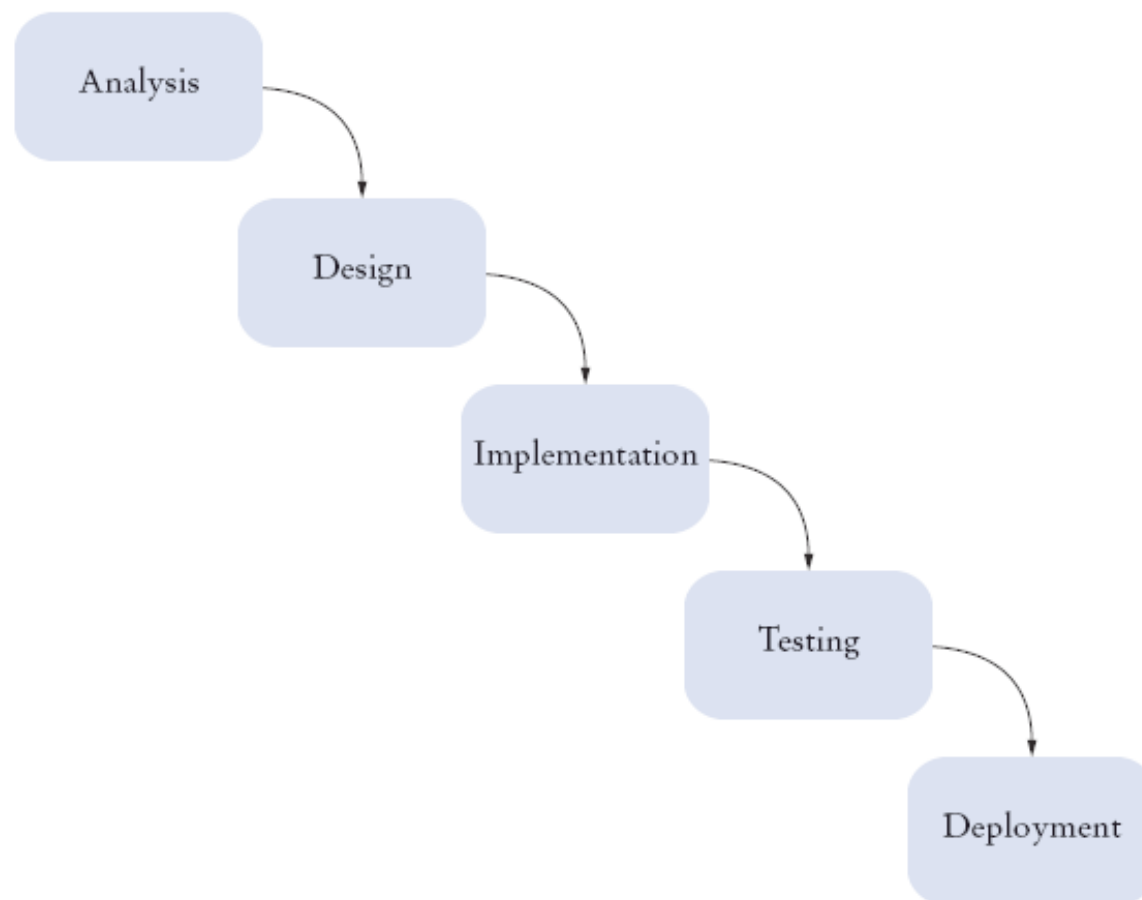
---

- 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



**Figure 1** The Waterfall Model

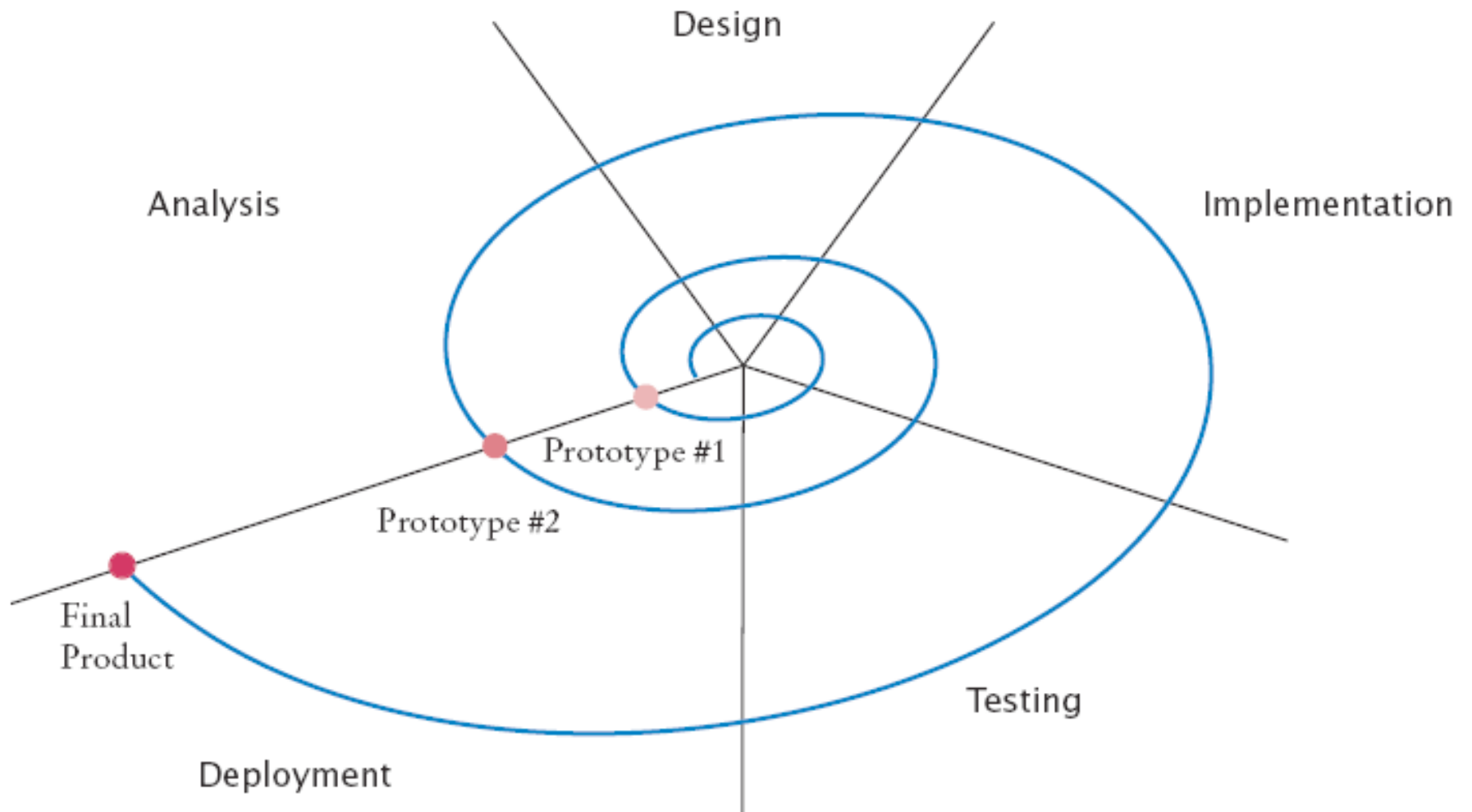
## The Spiral Model

---

- 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

***Continued***

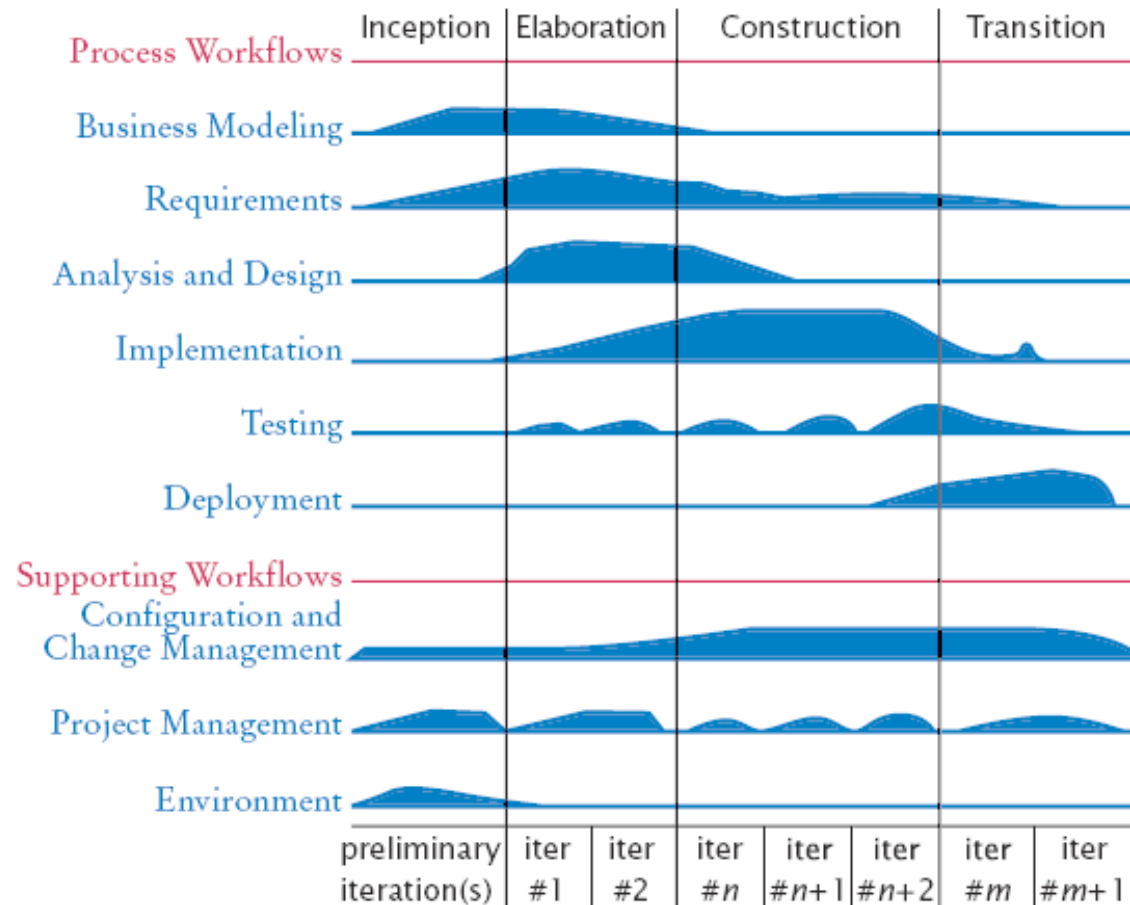
## The Spiral Model (cont.)



**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*

## Self Check 12.1

---

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.

## Self Check 12.3

---

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

---

<b>INVOICE</b>			
Sam's Small Appliances 100 Main Street Anytown, CA 98765			
<b>Item</b>	<b>Qty</b>	<b>Price</b>	<b>Total</b>
Toaster	3	\$29.95	\$89.85
Hair Dryer	1	\$24.95	\$24.95
Car Vacuum	2	\$19.99	\$39.98
<b>AMOUNT DUE: \$154.78</b>			

**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
    - Entities with multiple occurrences in problem description are good candidates for objects
    - Find out what they have in common
    - Design classes to capture commonalities
  - Represent some entities as objects, others as primitive types
    - 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

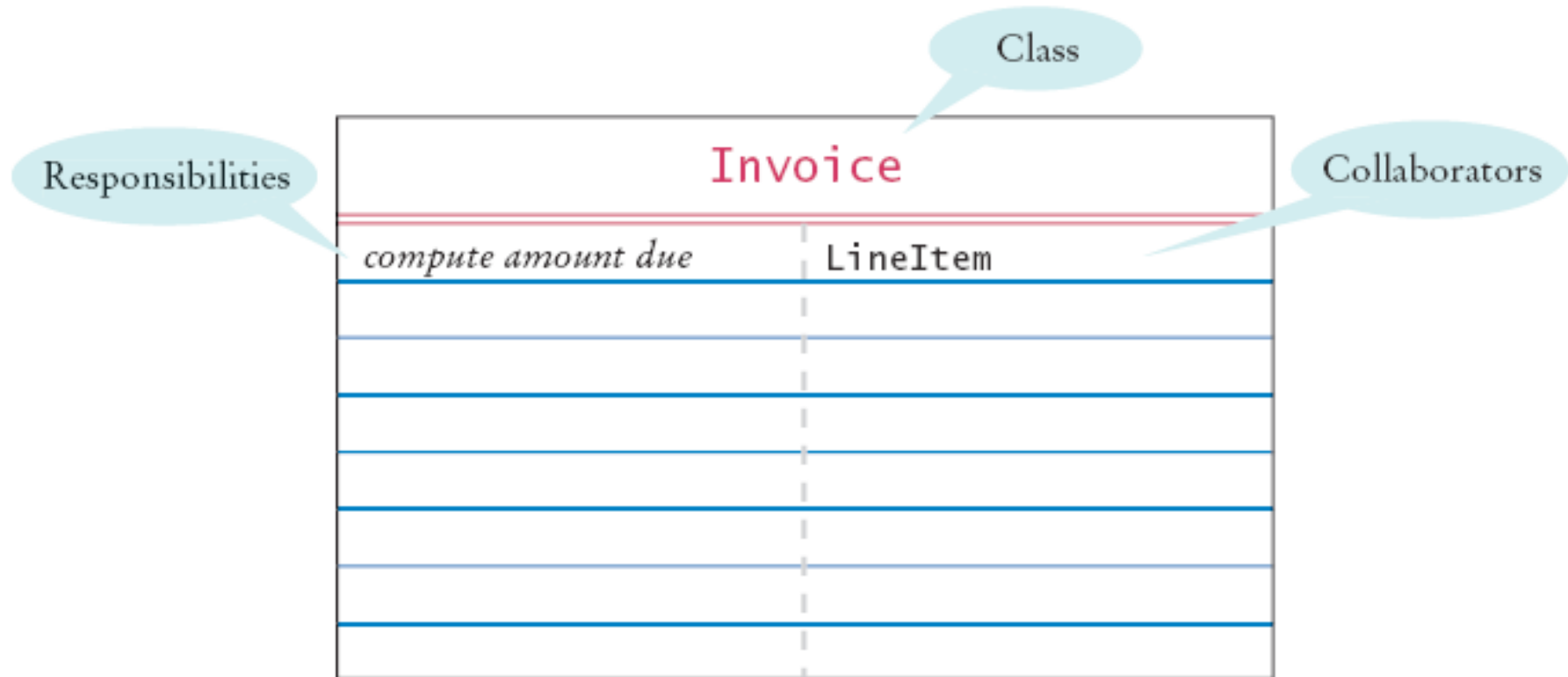
---

- 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***

## CRC Card (cont.)

---



**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.

## Self Check 12.6

---

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`?*
    - o 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:*

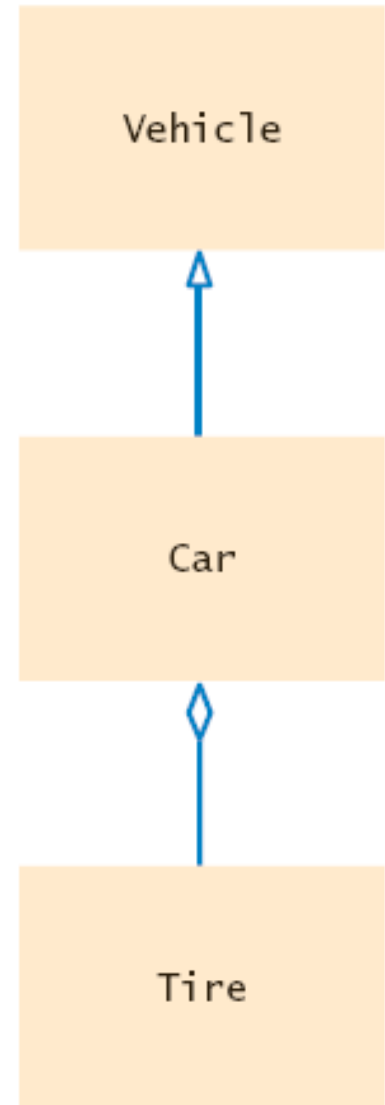
```
class Tire
{
    . . .
    private String rating;
    private Circle boundary;
}
```

- 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

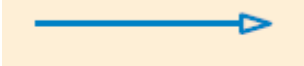


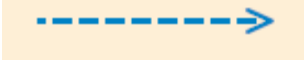
# Dependency

---

- *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

---

Relationship	Symbol	Line Style	Arrow Tip
Inheritance		Solid	Triangle
Interface Implementation		Dotted	Triangle
Aggregation		Solid	Diamond
Dependency		Dotted	Open

## 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.

## Self Check 12.9

---

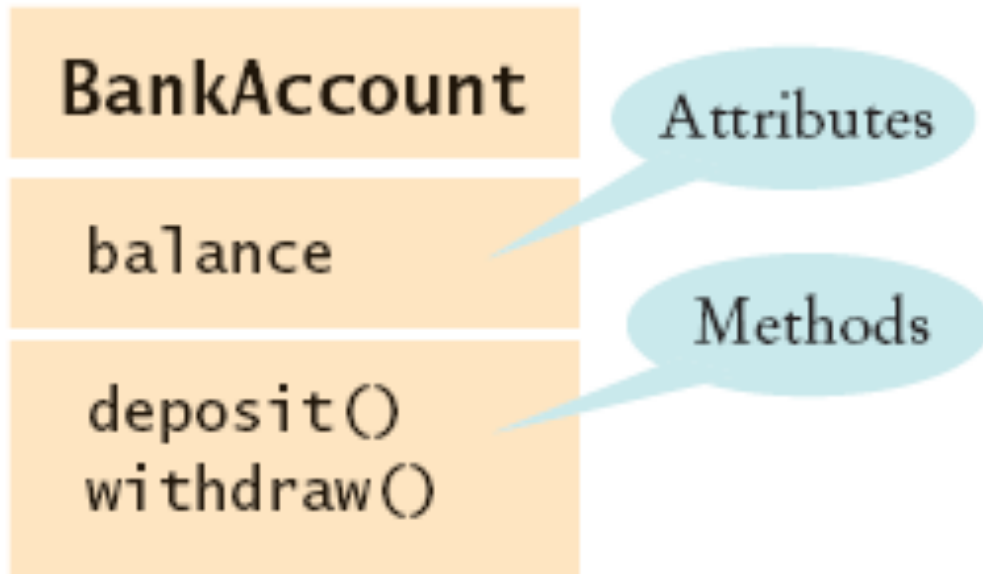
Consider the `BankAccountTester` class of Chapter 3. Which classes does it depend on?

**Answer:** The `BankAccount`, `System`, and `PrintStream` classes.



## Advanced Topic: Attributes and Methods in UML Diagrams

---



Attributes and Methods in a Class Diagram

## Advanced Topic: Multiplicities

---

- any number (zero or more): \*
- one or more: 1..\*
- zero or one: 0..1
- exactly one: 1



An Aggregation Relationship with Multiplicities

## 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



An Association Relationship

## 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

Description	Price	Qty	Total
Toaster	29.95	3	89.85
Hair dryer	24.95	1	24.95
Car vacuum	19.99	2	39.98

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

## Printing an Invoice – CRC Cards

---

- 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:

Invoice
<i>format the invoice</i>

Address
<i>format the address</i>

## CRC Cards for Printing Invoice

---

Add collaborators to invoice card:

Invoice	
<i>format the invoice</i>	Address
	LineItem

# CRC Cards for Printing Invoice

---

Product and LineItem CRC cards:

Product	
<i>get description</i>	
<i>get unit price</i>	

LineItem	
<i>format the item</i>	Product
<i>get total price</i>	

## CRC Cards for Printing Invoice

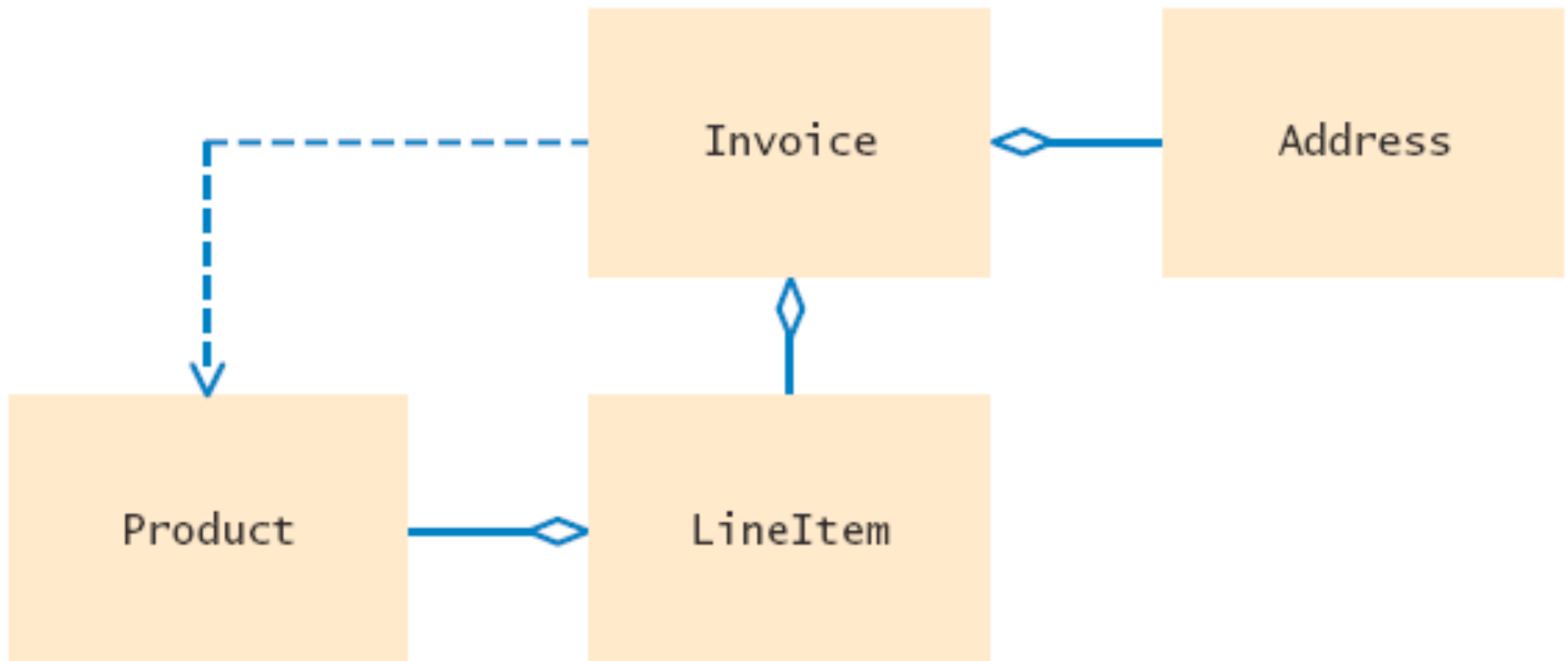
---

Invoice must be populated with products and quantities:

Invoice	
<i>format the invoice</i>	Address
<i>add a product and quantity</i>	LineItem
	Product

## 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)
    {
    }
}
```

***Continued***

## Method Documentation – Invoice class (cont.)

---

```
/**
    Formats the invoice.
    @return the formatted invoice
 */
public String format()
{
}
}
```



## Method Documentation – LineItem class

---

```
/**
    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()
    {
    }
```

***Continued***

## Method Documentation – LineItem class (cont.)

---

```
    /** Formats this item.  
    @return a formatted string of this line item  
    */  
    public String format()  
    {  
    }  
}
```

## Method Documentation – Product class

---

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

***Continued***

## Method Documentation – Product class (cont.)

---

```
*/  
public double getPrice()  
{  
}  
}
```

## Method Documentation – Address class

---

```
/**
Describes a mailing address.
*/
public class Address
{
    /**
    Formats the address.
    @return the address as a string with three lines
    */
    public String format()
    {
    }
}
```

# The Class Documentation in the HTML Format

Invoice - Firefox

file:///home/cay/BigJava/ch12/invoice/index.html

All Classes

- [Address](#)
- [Invoice](#)
- [InvoicePrinter](#)
- [LineItem](#)
- [Product](#)

[Package](#) [Class](#) [Tree](#) [Deprecated](#) [Index](#) [Help](#)

[PREV CLASS](#) [NEXT CLASS](#) [FRAMES](#) [NO FRAMES](#)

[SUMMARY: NESTED | FIELD | CONSTR | METHOD](#) [DETAIL: FIELD | CONSTR | METHOD](#)

## Class Invoice

java.lang.Object  
└ Invoice

```
public class Invoice
extends java.lang.Object
```

Describes an invoice for a set of purchased products.

### Constructor Summary

<a href="#">Invoice</a> ( <a href="#">Address</a> anAddress)
Constructs an invoice.

### Method Summary

void	<a href="#">add</a> ( <a href="#">Product</a> aProduct, int quantity)	Adds a charge for a product to this invoice.
java.lang.String	<a href="#">format</a> ()	Formats the invoice.
double	<a href="#">getAmountDue</a> ()	Computes the total amount due.

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:

```
public class Invoice
{
    . . .
    private Address billingAddress;
    private ArrayList<LineItem> items;
}
```



## Implementation

---

A line item needs to store a Product object and quantity:

```
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*

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

## ch12/invoice/InvoicePrinter.java

---

```
01: /**
02:     This program demonstrates the invoice classes by printing
03:     a sample invoice.
04: */
05: public class InvoicePrinter
06: {
07:     public static void main(String[] args)
08:     {
09:         Address samsAddress
10:             = new Address("Sam's Small Appliances",
11:                 "100 Main Street", "Anytown", "CA", "98765");
12:
13:         Invoice samsInvoice = new Invoice(samsAddress);
14:         samsInvoice.add(new Product("Toaster", 29.95), 3);
15:         samsInvoice.add(new Product("Hair dryer", 24.95), 1);
16:         samsInvoice.add(new Product("Car vacuum", 19.99), 2);
17:
18:         System.out.println(samsInvoice.format());
19:     }
20: }
21:
22:
23:
```

## ch12/invoice/Invoice.java

---

```
01: import java.util.ArrayList;
02:
03: /**
04:     Describes an invoice for a set of purchased products.
05: */
06: public class Invoice
07: {
08:     /**
09:         Constructs an invoice.
10:         @param anAddress the billing address
11:     */
12:     public Invoice(Address anAddress)
13:     {
14:         items = new ArrayList<LineItem>();
15:         billingAddress = anAddress;
16:     }
17:
18:     /**
19:         Adds a charge for a product to this invoice.
20:         @param aProduct the product that the customer ordered
21:         @param quantity the quantity of the product
22:     */
```

**Continued**

## ch12/invoice/Invoice.java (cont.)

---

```
23:     public void add(Product aProduct, int quantity)
24:     {
25:         LineItem anItem = new LineItem(aProduct, quantity);
26:         items.add(anItem);
27:     }
28:
29:     /**
30:      * Formats the invoice.
31:      * @return the formatted invoice
32:      */
33:     public String format()
34:     {
35:         String r = "                I N V O I C E\n\n"
36:             + billingAddress.format()
37:             + String.format("\n\n%-30s%8s%5s%8s\n",
38:                 "Description", "Price", "Qty", "Total");
39:
40:         for (LineItem i : items)
41:         {
42:             r = r + i.format() + "\n";
43:         }
44:
```

**Continued**

## ch12/invoice/Invoice.java (cont.)

---

```
45:         r = r + String.format("\nAMOUNT DUE: $%8.2f", getAmountDue());
46:
47:         return r;
48:     }
49:
50:     /**
51:      * Computes the total amount due.
52:      * @return the amount due
53:      */
54:     public double getAmountDue()
55:     {
56:         double amountDue = 0;
57:         for (LineItem i : items)
58:         {
59:             amountDue = amountDue + i.getTotalPrice();
60:         }
61:         return amountDue;
62:     }
63:
64:     private Address billingAddress;
65:     private ArrayList<LineItem> items;
66: }
```

## 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:     */
```

***Continued***

## ch12/invoice/LineItem.java (cont.)

---

```
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;
39: }
```



## ch12/invoice/Product.java

---

```
01: /**
02:     Describes a product with a description and a price.
03: */
04: public class Product
05: {
06:     /**
07:         Constructs a product from a description and a price.
08:         @param aDescription the product description
09:         @param aPrice the product price
10:     */
11:     public Product(String aDescription, double aPrice)
12:     {
13:         description = aDescription;
14:         price = aPrice;
15:     }
16:
17:     /**
18:         Gets the product description.
19:         @return the description
20:     */
```

***Continued***

## ch12/invoice/Product.java (cont.)

---

```
21:     public String getDescription()
22:     {
23:         return description;
24:     }
25:
26:     /**
27:      * Gets the product price.
28:      * @return the unit price
29:      */
30:     public double getPrice()
31:     {
32:         return price;
33:     }
34:
35:     private String description;
36:     private double price;
37: }
38:
```

## ch12/invoice/Address.java

---

```
01: /**
02:     Describes a mailing address.
03: */
04: public class Address
05: {
06:     /**
07:         Constructs a mailing address.
08:         @param aName the recipient name
09:         @param aStreet the street
10:         @param aCity the city
11:         @param aState the two-letter state code
12:         @param aZip the ZIP postal code
13:     */
14:     public Address(String aName, String aStreet,
15:         String aCity, String aState, String aZip)
16:     {
17:         name = aName;
18:         street = aStreet;
19:         city = aCity;
20:         state = aState;
21:         zip = aZip;
22:     }
```

***Continued***

## ch12/invoice/Address.java (cont.)

---

```
23:
24:     /**
25:         Formats the address.
26:         @return the address as a string with three lines
27:     */
28:     public String format()
29:     {
30:         return name + "\n" + street + "\n"
31:             + city + ", " + state + " " + zip;
32:     }
33:
34:     private String name;
35:     private String street;
36:     private String city;
37:     private String state;
38:     private String zip;
39: }
40:
```

## 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

---

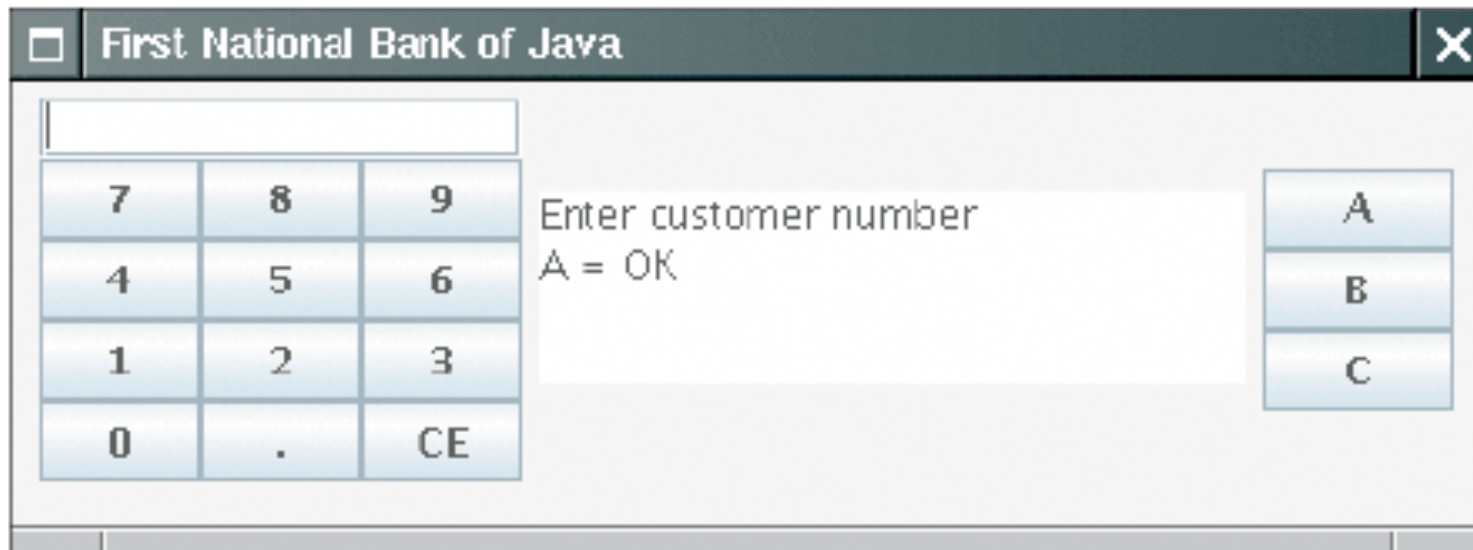
- Customers can select an account
- The balance of the selected account is displayed
- Then, customer can deposit and withdraw money
- Process is repeated until the customer chooses to exit



# An Automatic Teller Machine – Requirements

---

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



**Figure 9** Graphical User Interface for the Automatic Teller Machine

# 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:*

*Balance = balance of selected account*

*Enter amount and select transaction*

*A = Withdraw*

*B = Deposit*

*C = 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*



# An Automatic Teller Machine – CRC

---

## 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	
<i>get accounts</i>	
<i>match number and PIN</i>	

Bank	
<i>find customer</i>	Customer
<i>read customers</i>	

# CRC Cards for Automatic Teller Machine (cont.)

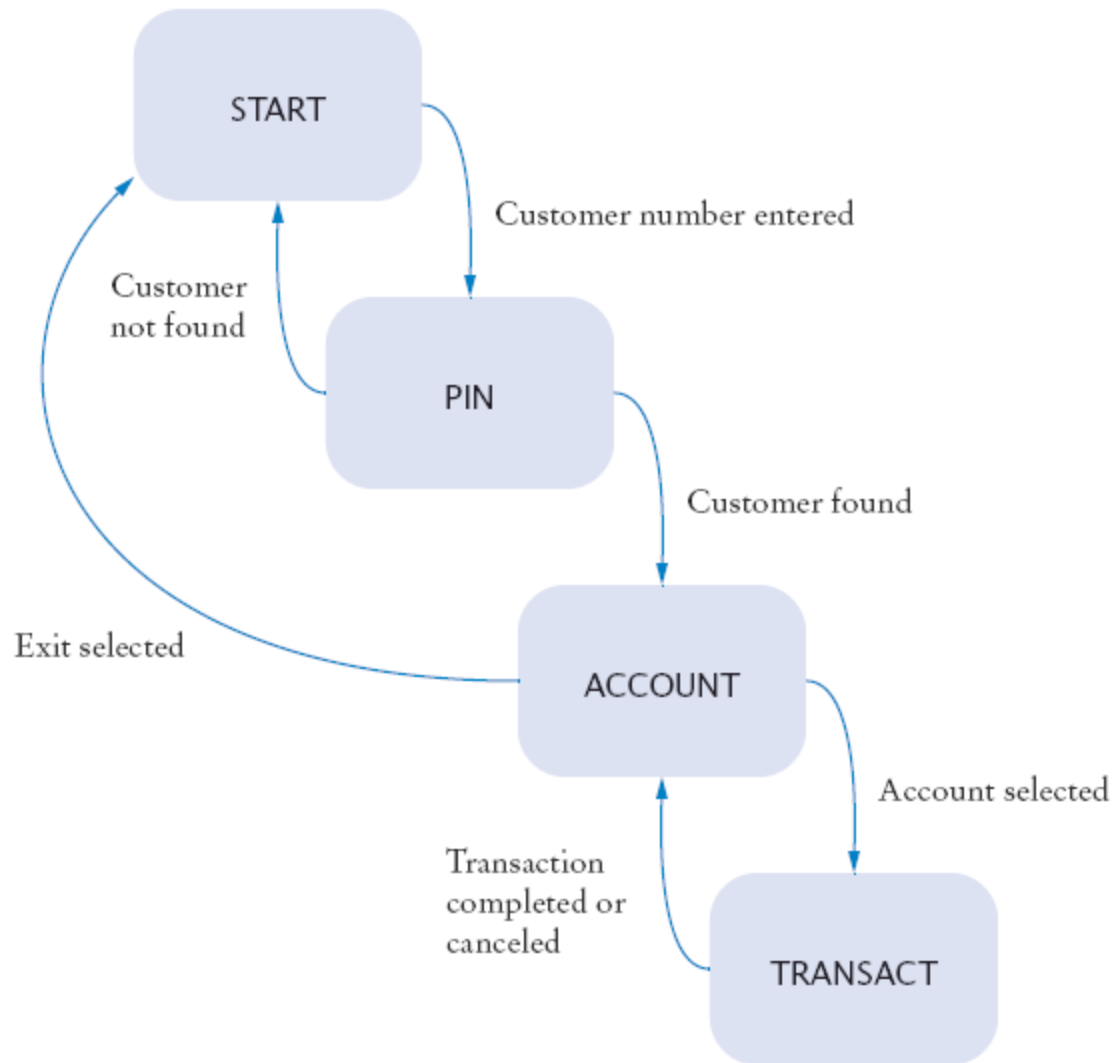
ATM	
<i>manage state</i>	Customer
<i>select customer</i>	Bank
<i>select account</i>	BankAccount
<i>execute transaction</i>	

## ATM States

---

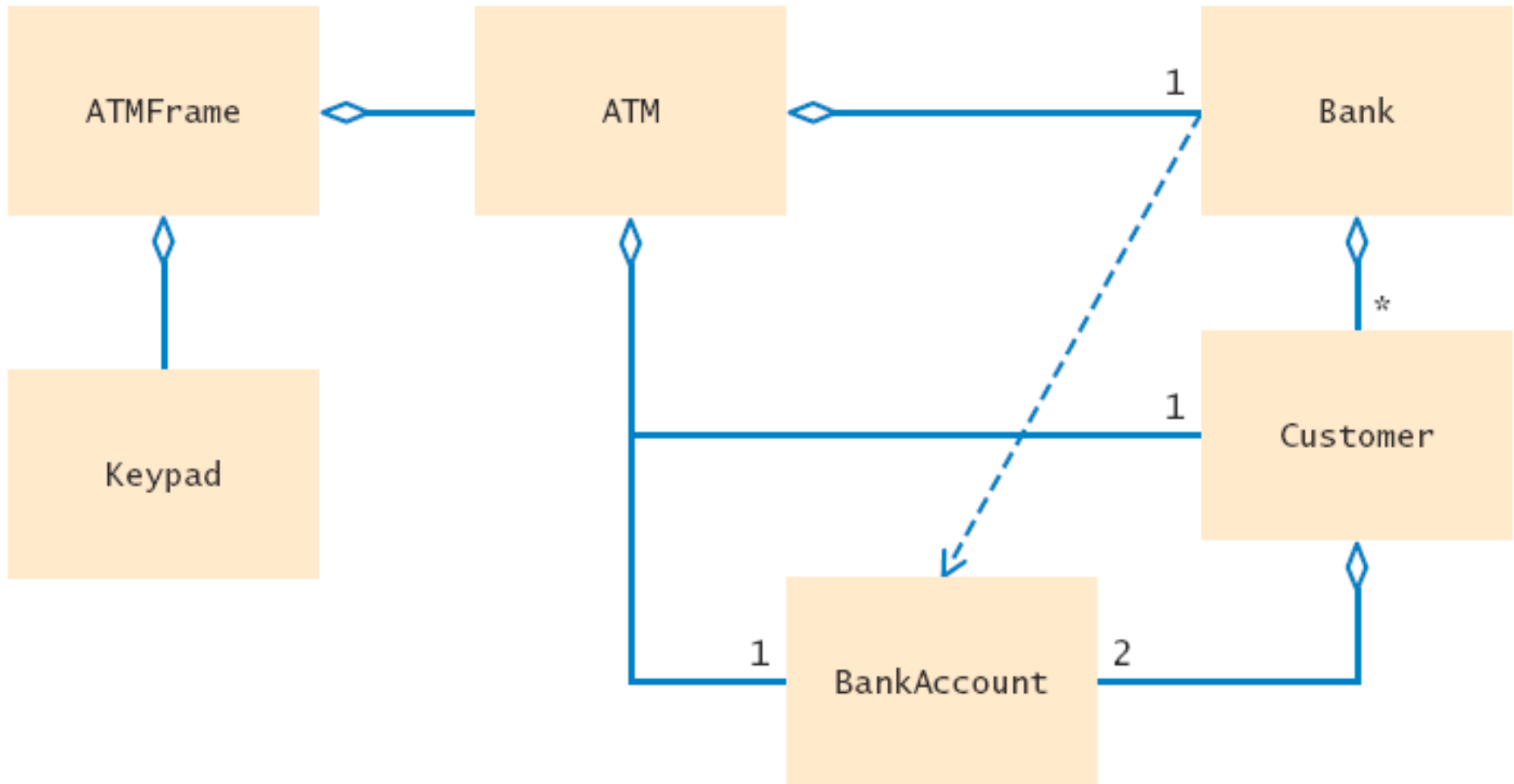
1. START: Enter customer ID
2. PIN: Enter PIN
3. ACCOUNT: Select account
4. TRANSACT: Select transaction

# State Diagram for ATM Class



**Figure 10** State Diagram for the ATM Class

# An Automatic Teller Machine – UML Diagrams



**Figure 11** Relationships Between the ATM Classes

## Method Documentation ATM Class

---

```
/**
 * 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) { }
```

## Method Documentation ATM Class (Continued)

---

```
/**
    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) { }
/**
    Sets current account to checking or savings. Sets
    state to TRANSACT.
    (Precondition: state is ACCOUNT or TRANSACT)
    @param account one of CHECKING or SAVINGS
 */
```

***Continued***



## Method Documentation ATM Class (Continued)

---

```
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

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

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

```
public class ATM
{
    ...
    private int state;
    private Customer currentCustomer;
    private BankAccount currentAccount;
}
```

## An Automatic Teller Machine – Implementation

---

- Most methods are very straightforward to implement
- Consider `selectCustomer`:

```
/**  
    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  
*/
```

***Continued***

## An Automatic Teller Machine – Implementation

---

- Description can be almost literally translated to Java instructions:

```
public void selectCustomer(int pin)
{
    assert state == PIN;
    currentCustomer = theBank.findCustomer(customerNumber,
        pin);
    if (currentCustomer == null)
        state = START;
    else
        state = ACCOUNT;
}
```

## ch12/atm/ATM.java

---

```
001: /**
002:     An ATM that accesses a bank.
003: */
004: public class ATM
005: {
006:     /**
007:         Constructs an ATM for a given bank.
008:         @param aBank the bank to which this ATM connects
009:     */
010:     public ATM(Bank aBank)
011:     {
012:         theBank = aBank;
013:         reset();
014:     }
015:
016:     /**
017:         Resets the ATM to the initial state.
018:     */
019:     public void reset()
020:     {
```

***Continued***

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

---

```
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**

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

---

```
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***



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

---

```
062:     public void selectAccount(int account)
063:     {
064:         assert state == ACCOUNT || state == TRANSACT;
065:         if (account == CHECKING)
066:             currentAccount = currentCustomer.getCheckingAccount();
067:         else
068:             currentAccount = currentCustomer.getSavingsAccount();
069:         state = TRANSACT;
070:     }
071:
072:     /**
073:      * Withdraws amount from current account.
074:      * (Precondition: state is TRANSACT)
075:      * @param value the amount to withdraw
076:      */
077:     public void withdraw(double value)
078:     {
079:         assert state == TRANSACT;
080:         currentAccount.withdraw(value);
081:     }
082:
```

***Continued***

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

---

```
083:     /**
084:         Deposits amount to current account.
085:         (Precondition: state is TRANSACT)
086:         @param value the amount to deposit
087:     */
088:     public void deposit(double value)
089:     {
090:         assert state == TRANSACT;
091:         currentAccount.deposit(value);
092:     }
093:
094:     /**
095:         Gets the balance of the current account.
096:         (Precondition: state is TRANSACT)
097:         @return the balance
098:     */
099:     public double getBalance()
100:     {
101:         assert state == TRANSACT;
102:         return currentAccount.getBalance();
103:     }
```

***Continued***

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

---

```
104:
105:     /**
106:         Moves back to the previous state.
107:     */
108:     public void back()
109:     {
110:         if (state == TRANSACT)
111:             state = ACCOUNT;
112:         else if (state == ACCOUNT)
113:             state = PIN;
114:         else if (state == PIN)
115:             state = START;
116:     }
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: }
```

## ch12/atm/Bank.java

---

```
01: import java.io.FileReader;
02: import java.io.IOException;
03: import java.util.ArrayList;
04: import java.util.Scanner;
05:
06: /**
07:     A bank contains customers with bank accounts.
08: */
09: public class Bank
10: {
11:     /**
12:         Constructs a bank with no customers.
13:     */
14:     public Bank()
15:     {
16:         customers = new ArrayList<Customer>();
17:     }
18:
19:     /**
20:         Reads the customer numbers and pins
21:         and initializes the bank accounts.
22:         @param filename the name of the customer file
23:     */
```

**Continued**

## ch12/atm/Bank.java (cont.)

---

```
24:     public void readCustomers(String filename)
25:         throws IOException
26:     {
27:         Scanner in = new Scanner(new FileReader(filename));
28:         while (in.hasNext())
29:         {
30:             int number = in.nextInt();
31:             int pin = in.nextInt();
32:             Customer c = new Customer(number, pin);
33:             addCustomer(c);
34:         }
35:         in.close();
36:     }
37:
38:     /**
39:      Adds a customer to the bank.
40:      @param c the customer to add
41:     */
42:     public void addCustomer(Customer c)
43:     {
44:         customers.add(c);
45:     }
```

***Continued***

## ch12/atm/Bank.java (cont.)

---

```
46:
47:     /**
48:         Finds a customer in the bank.
49:         @param aNumber a customer number
50:         @param aPin a personal identification number
51:         @return the matching customer, or null if no customer
52:         matches
53:     */
54:     public Customer findCustomer(int aNumber, int aPin)
55:     {
56:         for (Customer c : customers)
57:         {
58:             if (c.match(aNumber, aPin))
59:                 return c;
60:         }
61:         return null;
62:     }
63:
64:     private ArrayList<Customer> customers;
65: }
66:
67:
```

## ch12/atm/Customer.java

---

```
01: /**
02:     A bank customer with a checking and a savings account.
03: */
04: public class Customer
05: {
06:     /**
07:         Constructs a customer with a given number and PIN.
08:         @param aNumber the customer number
09:         @param aPin the personal identification number
10:     */
11:     public Customer(int aNumber, int aPin)
12:     {
13:         customerNumber = aNumber;
14:         pin = aPin;
15:         checkingAccount = new BankAccount();
16:         savingsAccount = new BankAccount();
17:     }
18:
19:     /**
20:         Tests if this customer matches a customer number
21:         and PIN.
22:         @param aNumber a customer number
23:         @param aPin a personal identification number
```

**Continued**



## ch12/atm/Customer.java (cont.)

---

```
24:     @return true if the customer number and PIN match
25:     */
26:     public boolean match(int aNumber, int aPin)
27:     {
28:         return customerNumber == aNumber && pin == aPin;
29:     }
30:
31:     /**
32:         Gets the checking account of this customer.
33:         @return the checking account
34:     */
35:     public BankAccount getCheckingAccount()
36:     {
37:         return checkingAccount;
38:     }
39:
40:     /**
41:         Gets the savings account of this customer.
42:         @return the checking account
43:     */
44:     public BankAccount getSavingsAccount()
45:     {
```

**Continued**

## ch12/atm/Customer.java (cont.)

---

```
46:         return savingsAccount;
47:     }
48:
49:     private int customerNumber;
50:     private int pin;
51:     private BankAccount checkingAccount;
52:     private BankAccount savingsAccount;
53: }
```

## ch12/atm/ATMSimulator.java

---

```
01: import java.io.IOException;
02: import java.util.Scanner;
03:
04: /**
05:     A text-based simulation of an automatic teller machine.
06: */
07: public class ATMSimulator
08: {
09:     public static void main(String[] args)
10:     {
11:         ATM theATM;
12:         try
13:         {
14:             Bank theBank = new Bank();
15:             theBank.readCustomers("customers.txt");
16:             theATM = new ATM(theBank);
17:         }
18:         catch(IOException e)
19:         {
20:             System.out.println("Error opening accounts file.");
21:             return;
22:         }
```

## ch12/atm/ATMSimulator.java (cont.)

---

```
23:
24:     Scanner in = new Scanner(System.in);
25:
26:     while (true)
27:     {
28:         int state = theATM.getState();
29:         if (state == ATM.START)
30:         {
31:             System.out.print("Enter customer number: ");
32:             int number = in.nextInt();
33:             theATM.setCustomerNumber(number);
34:         }
35:         else if (state == ATM.PIN)
36:         {
37:             System.out.print("Enter PIN: ");
38:             int pin = in.nextInt();
39:             theATM.selectCustomer(pin);
40:         }
41:         else if (state == ATM.ACCOUNT)
42:         {
43:             System.out.print("A=Checking, B=Savings, C=Quit: ");
44:             String command = in.next();
```

## ch12/atm/ATMSimulator.java (cont.)

---

```
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:         }
```

## ch12/atm/ATMSimulator.java (cont.)

---

```
66:         else if (command.equalsIgnoreCase("B"))
67:         {
68:             System.out.print("Amount: ");
69:             double amount = in.nextDouble();
70:             theATM.withdraw(amount);
71:             theATM.back();
72:         }
73:         else if (command.equalsIgnoreCase("C"))
74:             theATM.back();
75:         else
76:             System.out.println("Illegal input!");
77:     }
78: }
79: }
80: }
81:
```

## ch12/atm/ATMSimulator.java (cont.)

---

```
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
. . .
```

## ch12/atm/ATMViewer.java

---

```
01: import java.io.IOException;
02: import javax.swing.JFrame;
03: import javax.swing.JOptionPane;
04:
05: /**
06:     A graphical simulation of an automatic teller machine.
07: */
08: public class ATMViewer
09: {
10:     public static void main(String[] args)
11:     {
12:         ATM theATM;
13:
14:         try
15:         {
16:             Bank theBank = new Bank();
17:             theBank.readCustomers("customers.txt");
18:             theATM = new ATM(theBank);
19:         }
20:         catch(IOException e)
21:         {
```

***Continued***



## ch12/atm/ATMViewer.java (cont.)

---

```
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**

## ch12/atm/ATMFrame.java (cont.)

---

```
023:         pad = new Keypad();
024:
025:         display = new JTextArea(4, 20);
026:
027:         aButton = new JButton("  A  ");
028:         aButton.addActionListener(new AButtonListener());
029:
030:         bButton = new JButton("  B  ");
031:         bButton.addActionListener(new BButtonListener());
032:
033:         cButton = new JButton("  C  ");
034:         cButton.addActionListener(new CButtonListener());
035:
036:         // Add components
037:
038:         JPanel buttonPanel = new JPanel();
039:         //buttonPanel.setLayout(new GridLayout(3, 1));
040:         buttonPanel.add(aButton);
041:         buttonPanel.add(bButton);
042:         buttonPanel.add(cButton);
043:
044:         setLayout(new FlowLayout());
045:         add(pad);
046:         add(display);
```

**Continued**

## ch12/atm/ATMFrame.java (cont.)

---

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

**Continued**

## ch12/atm/ATMFrame.java (cont.)

---

```
070:         + "\nEnter amount and select transaction\n"
071:         + "A = Withdraw\nB = Deposit\nC = Cancel");
072:     }
073:
074:     private class AButtonListener implements ActionListener
075:     {
076:         public void actionPerformed(ActionEvent event)
077:         {
078:             int state = theATM.getState();
079:             if (state == ATM.START)
080:                 theATM.setCustomerNumber((int) pad.getValue());
081:             else if (state == ATM.PIN)
082:                 theATM.selectCustomer((int) pad.getValue());
083:             else if (state == ATM.ACCOUNT)
084:                 theATM.selectAccount(ATM.CHECKING);
085:             else if (state == ATM.TRANSACT)
086:             {
087:                 theATM.withdraw(pad.getValue());
088:                 theATM.back();
089:             }
090:             showState();
091:         }
092:     }
```

**Continued**

## ch12/atm/ATMFrame.java (cont.)

---

```
093:
094:     private class BButtonListener implements ActionListener
095:     {
096:         public void actionPerformed(ActionEvent event)
097:         {
098:             int state = theATM.getState();
099:             if (state == ATM.ACCOUNT)
100:                 theATM.selectAccount(ATM.SAVINGS);
101:             else if (state == ATM.TRANSACT)
102:             {
103:                 theATM.deposit(pad.getValue());
104:                 theATM.back();
105:             }
106:             showState();
107:         }
108:     }
109:
110:     private class CButtonListener implements ActionListener
111:     {
112:         public void actionPerformed(ActionEvent event)
113:         {
114:             int state = theATM.getState();
115:             if (state == ATM.ACCOUNT)
116:                 theATM.reset();
```

**Continued**

*Big Java* by Cay Horstmann

Copyright © 2008 by John Wiley & Sons. All rights reserved.

## ch12/atm/ATMFrame.java (cont.)

---

```
117:         else if (state == ATM.TRANSACT)
118:             theATM.back();
119:             showState();
120:         }
121:     }
122:
123:     private JButton aButton;
124:     private JButton bButton;
125:     private JButton cButton;
126:
127:     private KeyPad pad;
128:     private JTextArea display;
129:
130:     private ATM theATM;
131:
132:     private static final int FRAME_WIDTH = 300;
133:     private static final int FRAME_HEIGHT = 400;
134: }
```

## File KeyPad.java

---

```
001: import java.awt.BorderLayout;
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.JPanel;
007: import javax.swing.JTextField;
008:
009: /**
010:     A component that lets the user enter a number, using
011:     a button pad labeled with digits.
012: */
013: public class KeyPad extends JPanel
014: {
015:     /**
016:         Constructs the keypad panel.
017:     */
018:     public KeyPad()
019:     {
020:         setLayout(new BorderLayout());
021:
022:         // Add display field
023:
```

**Continued**



## File KeyPad.java (cont.)

---

```
024:         display = new JTextField();
025:         add(display, "North");
026:
027:         // Make button panel
028:
029:         buttonPanel = new JPanel();
030:         buttonPanel.setLayout(new GridLayout(4, 3));
031:
032:         // Add digit buttons
033:
034:         addButton("7");
035:         addButton("8");
036:         addButton("9");
037:         addButton("4");
038:         addButton("5");
039:         addButton("6");
040:         addButton("1");
041:         addButton("2");
042:         addButton("3");
043:         addButton("0");
044:         addButton(".");
045:
046:         // Add clear entry button
```

***Continued***

## File KeyPad.java (cont.)

---

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

**Continued**

## File KeyPad.java (cont.)

---

```
070:     private void addButton(final String label)
071:     {
072:         class DigitButtonListener implements ActionListener
073:         {
074:             public void actionPerformed(ActionEvent event)
075:             {
076:
077:                 // Don't add two decimal points
078:                 if (label.equals("."))
079:                     && display.getText().indexOf(".") != -1)
080:                     return;
081:
082:                 // Append label text to button
083:                 display.setText(display.getText() + label);
084:             }
085:         }
086:
087:         JButton button = new JButton(label);
088:         buttonPanel.add(button);
089:         ActionListener listener = new DigitButtonListener();
090:         button.addActionListener(listener);
091:     }
092:
```

**Continued**

## File KeyPad.java (cont.)

---

```
093:     /**
094:         Gets the value that the user entered.
095:         @return the value in the text field of the keypad
096:     */
097:     public double getValue()
098:     {
099:         return Double.parseDouble(display.getText());
100:     }
101:
102:     /**
103:         Clears the display.
104:     */
105:     public void clear()
106:     {
107:         display.setText("");
108:     }
109:
110:     private JPanel buttonPanel;
111:     private JButton clearButton;
112:     private JTextField display;
113: }
114:
```

## 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.

## Self Check 12.13

---

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.