



D06

PROGRAMMING with JAVA

Ch17 – Object-Oriented Design

PowerPoint presentation, created by Angel A. Juan - ajuanp@gmail.com,
for accompanying the book "Big Java", by Cay S. Horstmann (2006)

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

- The **software life cycle** encompasses all the activities that take place between the time a software program is first conceived (initial analysis) and the time it is finally retired (obsolescence).
- A **formal process** for software development:
 - Describes phases of the development process
 - Gives guidelines for how to carry out the phases



5-phases model for the development process:

1. Analysis
2. Design
3. Implementation
4. Testing
5. Deployment

Analysis

- In the **analysis phase**, you:
 - Decide what the project is supposed to accomplish
 - Do not think about how the program will accomplish its tasks
- Output of the analysis phase → **requirements document**
 - It describes what the program will be able to do once it is completed
 - It can include a **user manual** (which tells the user how to operate the program)
 - It can include a **performance criteria** (i.e., how many inputs the program must be able to handle in what time, or what its minimum memory and disk storage requirements are)

Design

- In the **design phase**, you:
 - Develop a plan for how you will implement the system
 - Discover the structures that underlie the problem to be solved
 - Decide what classes you need and what their most important methods are
- Output of the design phase → a description of the classes and methods, with **diagrams** that show the relationships among the classes

Implementation

- In the **implementation phase**, you write and compile program code to implement the classes and methods that were discovered in the design phase
- Output of the implementation phase → the completed **program**

Testing & Deployment

- In the **testing phase**, you run tests to verify that the program works correctly
- Output of the testing phase → a **report** describing the tests that you carried out and their results
- Finally, in the **deployment phase**, the users of the program install it and use it for its intended purpose

The Waterfall Model

- The **waterfall model** of software development describes a sequential process of analysis, design, implementation, testing, and deployment
- When rigidly applied, the waterfall model did not work

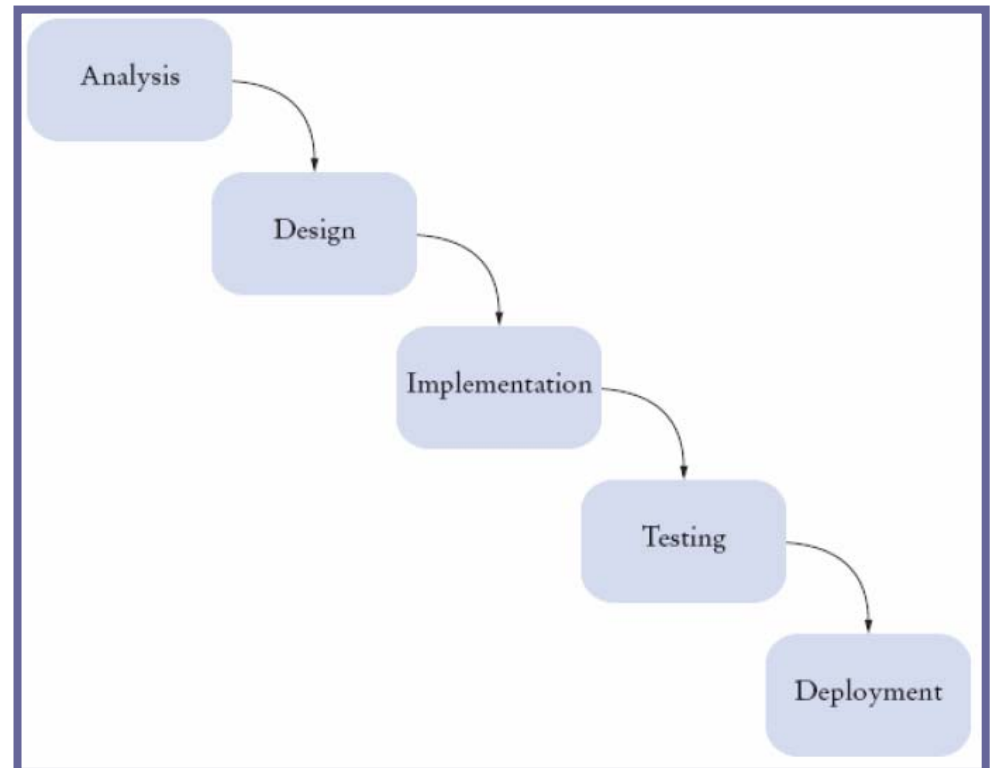


Figure 1:
The Waterfall Method

The Spiral Model

- The **spiral model** breaks the 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

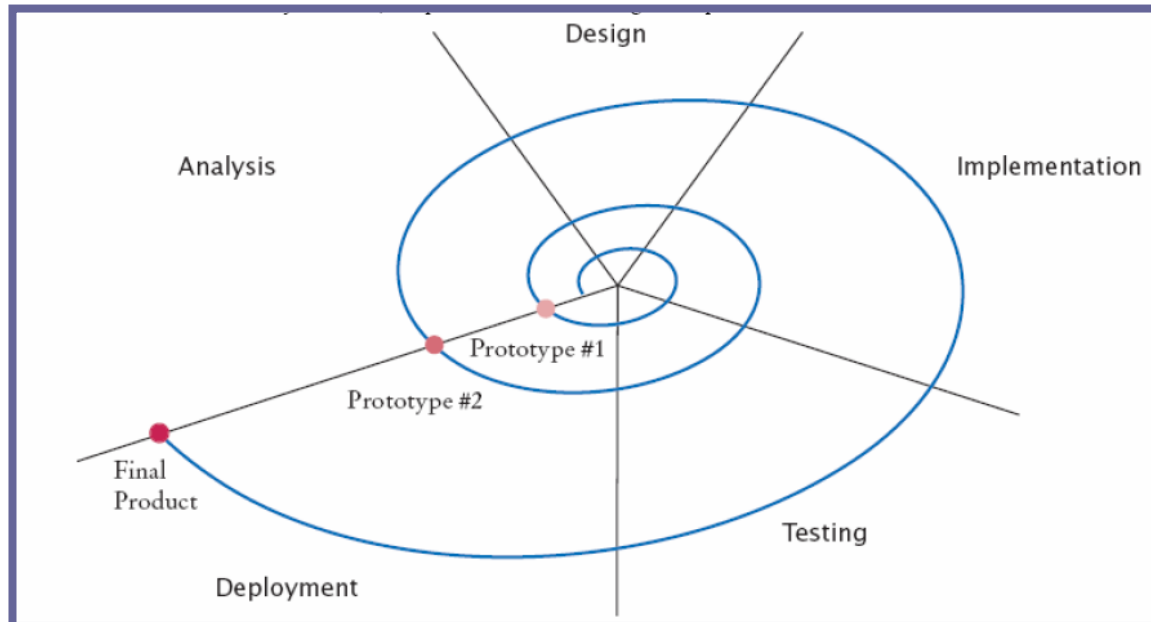


Figure 2:
A Spiral Model

Self Check

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?
2. Does Extreme Programming follow a waterfall or a spiral model?
3. What is the purpose of the "on-site customer" in Extreme Programming?

Answers

1. 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.
2. An "extreme" spiral model, with lots of iterations.
3. To give frequent feedback as to whether the current iteration of the product fits customer needs.

Discovering Classes



When you use the **object-oriented design** process, you carry out the following tasks:

1. Discover **classes**
 2. Determine the **responsibilities** of each class
 3. Describe the **relationship** between the 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
 - When finding classes, keep the following points in mind:
 - A class represents a set of objects with the same behavior
 - Some entities should be represented as objects, others as primitive types
 - Not all classes can be discovered in the analysis phase
 - Some of the classes that you need may already exist, either in the standard library or in a program that you developed previously

Defining Behavior

- Once a set of classes has been identified, define the **behavior** for each class (i.e., find out what methods each object needs to carry out to solve the programming problem)
- Find methods by looking for **verbs** in the task description

Example: Invoice

- **Classes** that come to mind: **Invoice**, **LineItem**, and **Customer**
- It is a good idea to keep a list of **candidate** classes

INVOICE

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

Item	Qty	Price	Total
Toaster	3	\$29.95	\$89.85
Hair Dryer	1	\$24.95	\$24.95
Car Vacuum	2	\$19.99	\$39.98

AMOUNT DUE: \$154.78

**Figure 4:
An Invoice**

CRC Card Method

- The CRC card method describes a **class**, its **responsibilities**, and its **collaborators** :
 - a) Use an index card for each class
 - b) Pick the class that should be responsible for each method (verb)
 - c) Write the responsibility onto the class card
 - d) Indicate what other classes are needed to fulfill responsibility (collaborators)

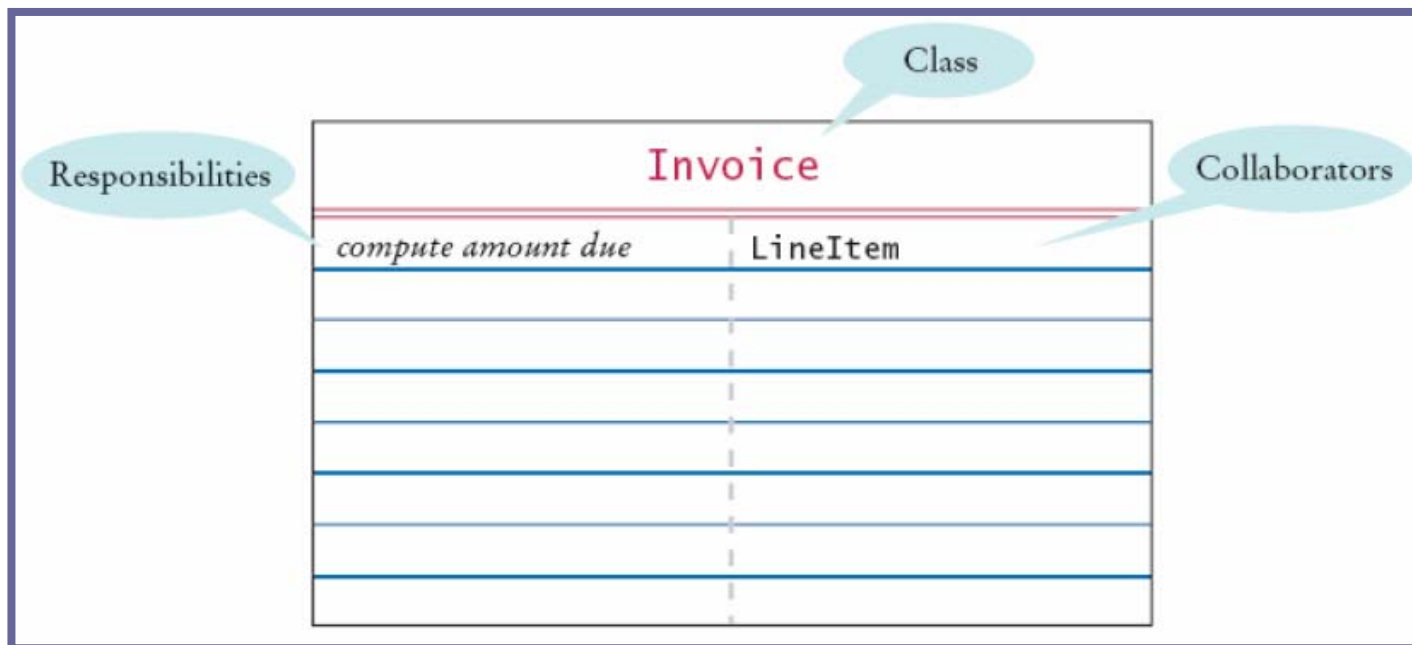


Figure 5:
A CRC Card

Self Check

3. Suppose the invoice is to be saved to a file. Name a likely collaborator.
4. Looking at the invoice in Figure 4, what is a likely responsibility of the **Customer** class?
5. What do you do if a CRC card has ten responsibilities?

Answers

3. **FileWriter**
4. To produce the shipping address of the customer.
5. Reword the responsibilities so that they are at a higher level, or come up with more classes to handle the responsibilities.

Relationships Between Classes

- When designing a program, it is useful to document the **relationships** between classes:
 - If you find classes with common behavior, you can save effort by placing the common behavior into a **superclass**
 - If you know that some classes are not related to each other, you can assign different programmers to implement each of them
- **Inheritance:**
 - *is-a* relationship: every savings account is a bank account, every circle is an ellipse, etc.
 - Relationship between a more general class (**superclass**) and a more specialized class (**subclass**)
 - It is sometimes abused, e.g.: should the class `Tire` be a subclass of a class `Circle`? the *has-a* relationship would be more appropriate

Continued...

Relationships Between Classes

- **Aggregation:**

- *Has-a* relationship
- Objects of one class **contain** references to objects of another class, e.g.: every car has four tires
- Use an instance variable, e.g.: a tire has a circle as its boundary:

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

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

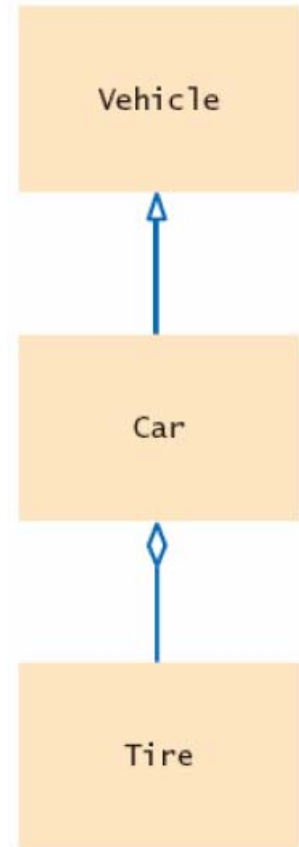


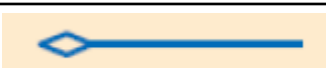



Figure 6:
UML Notation for Inheritance
and Aggregation

Continued...

Relationships Between Classes

- **Dependency:**
 - *Uses* relationship: many of our applications depend on the `Scanner` class to read input
 - Aggregation is a stronger form of dependency (if a class has objects of another class, it certainly uses the other class)
 - Generally, you need aggregation when an object needs to remember another object between method calls

Relationship	Symbol	Line Style	Arrow Tip
Inheritance (is-a)		Solid	Triangle
Interface Implementation		Dotted	Triangle
Aggregation (has-a)		Solid	Diamond
Dependency (uses)		Dotted	Open

Attributes and Methods in UML Diagrams

- An **attribute** is an externally observable property that objects of a class have (e.g., `name` and `price` would be attributes of the `Product` class)
- Usually, attributes correspond to instance variables
- It is useful to indicate class attributes and methods in a **UML class diagram**:

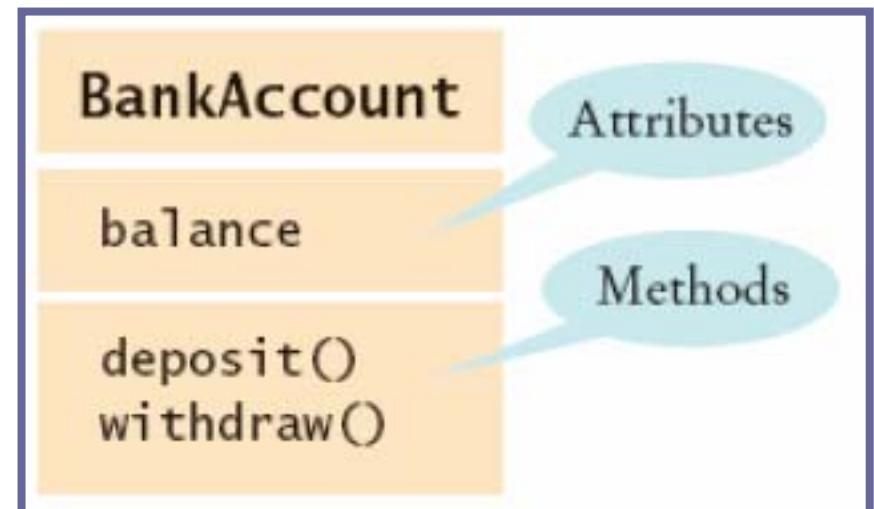


Figure 7:
Attributes and Methods in a
Class Diagram

Multiplicities

- **Multiplicities** can be written at the end(s) of an aggregation relationship to denote how many objects are aggregated:
 - any number (zero or more): *****
 - one or more: **1..***
 - zero or one: **0..1**
 - exactly one: **1**



Figure 8:
An Aggregation Relationship with Multiplicities

Aggregation and Association

- A class is **associated** with another if you can navigate from objects of one class to objects of the other class. E.g., given a `Bank` object, you can navigate to `Customer` objects
- Association is a more general relationship than aggregation
- Use association early in the design phase



Figure 9:
An Association Relationship

Self Check

7. Consider the `Bank` and `BankAccount` classes of Chapter 7. How are they related?
8. Consider the `BankAccount` and `SavingsAccount` objects of Chapter 12. How are they related?
9. Consider the `BankAccountTester` class of Chapter 3. Which classes does it depend on?

Answers

7. Through aggregation. The bank manages bank account objects.
8. Through inheritance.
9. The `BankAccount`, `System`, and `PrintStream` classes.

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

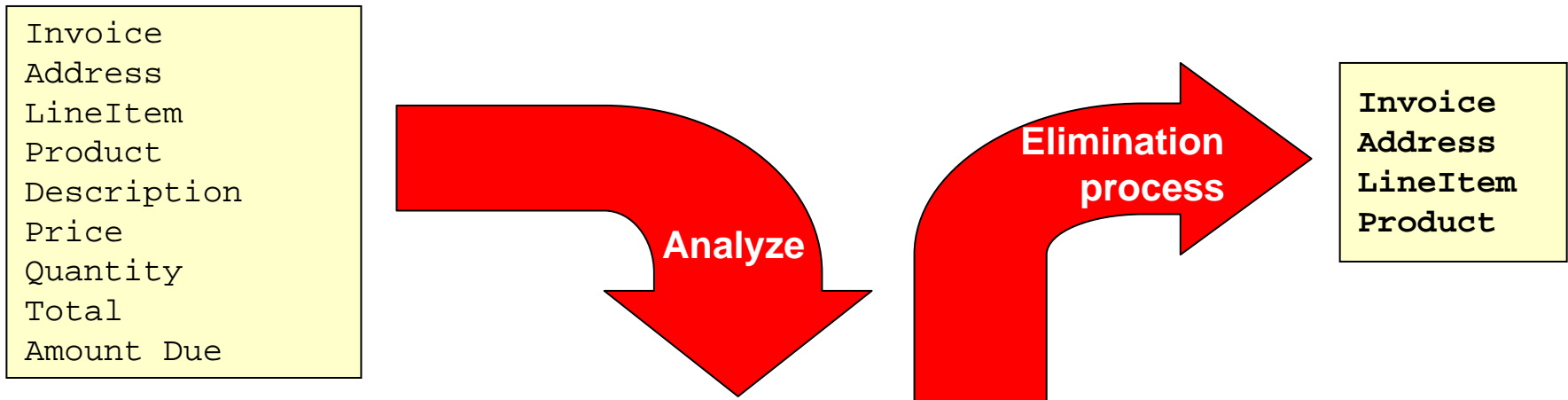
Printing an Invoice – Requirements

- **Task:** print out an invoice
- **Invoice:** describes the charges for a set of products in certain quantities; omit complexities (dates, taxes, etc.)
- **Print invoice:** billing address, all line items, amount due
- **Line item:** description, unit price, quantity ordered, total price
- **Test program:** adds line items to the invoice and then prints it

I N V O I C E			
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    // 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
```


Printing an Invoice – UML Diagrams

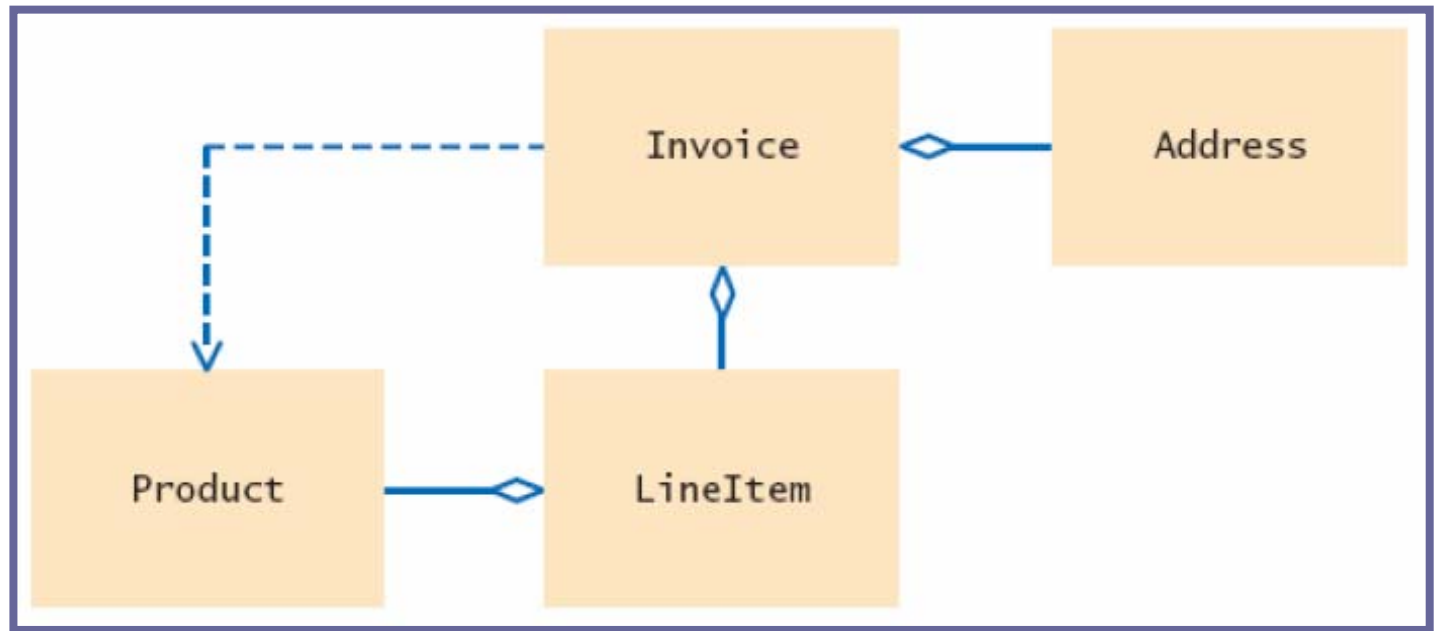


Figure 10:
The Relationships Between the Invoice Classes

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()
    {
    }
}
```

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()
    {
    }

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

Printing an Invoice – Implementation

- In the **UML diagram**, look for **associated** classes, they yield instance fields:
 - 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;
}
```

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

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

Printing an Invoice – 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;
}
```

File InvoiceTester.java

```
01: /**
02:     This program tests the invoice classes by printing
03:     a sample invoice.
04: */
05: public class InvoiceTester
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);
```

Continued...

File InvoiceTester.java

```
17:  
18:     System.out.println(samsInvoice.format());  
19: }  
20: }  
21:  
22:  
23:
```

File 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:
```

Continued...

File Invoice.java

```
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:     */
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:     {
```

Continued...

File Invoice.java

```
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:
45:         r = r + String.format("\nAMOUNT DUE: $%8.2f",
46:             getAmountDue());
47:         return r;
48:     }
49:
```

Continued...

File Invoice.java

```
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: }
```

File LineItem.java

```
01: /**
02:     Describes a quantity 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:
```

Continued...

File LineItem.java

```
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()
```

Continued...

File LineItem.java

```
31:     {
32:         return String.format("%-30s%8.2f%5d%8.2f",
33:             theProduct.getDescription(),
34:             theProduct.getPrice(),
35:             quantity, getTotalPrice());
36:     }
37:     private int quantity;
38:     private Product theProduct;
39: }
```

Continued...

File 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:
```

Continued...

File Product.java

```
17:     /**
18:         Gets the product description.
19:         @return the description
20:     */
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:     }
```

Continued...

File Product.java

```
34:  
35:     private String description;  
36:     private double price;  
37: }  
38:
```

File 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:     {
```

Continued...

File Address.java

```
17:     name = aName;
18:     street = aStreet;
19:     city = aCity;
20:     state = aState;
21:     zip = aZip;
22: }
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:
```

Continued...

File Address.java

```
34:     private String name;  
35:     private String street;  
36:     private String city;  
37:     private String state;  
38:     private String zip;  
39: }  
40:
```

Self Check

10. Which class is responsible for computing the amount due? What are its collaborators for this task?
11. Why do the format methods return `String` objects instead of directly printing to `System.out`?

Answers

10. The `Invoice` class is responsible for computing the amount due. It collaborates with the `LineItem` class.
11. 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.

Chapter Summary

- The **life cycle** of software encompasses all activities from initial analysis until obsolescence
- A **formal process** for software development describes phases of the development process and gives guidelines for how to carry out the phases
- The **waterfall model** of software development describes a sequential process of analysis, design, implementation, testing, and deployment
- The **spiral model** of software development describes an iterative process in which design and implementation are repeated
- **Extreme Programming** is a development methodology that strives for simplicity by removing formal structure and focusing on best practices
- In object-oriented design, you **discover** classes, determine the **responsibilities** of classes, and describe the **relationships** between classes

Continued...

Chapter Summary

- A **CRC card** describes a class, its responsibilities, and its collaborating classes
- **Inheritance** (the *is-a* relationship) is sometimes inappropriately used when the *has-a* relationship would be more appropriate
- **Aggregation** (the *has-a* relationship) denotes that objects of one class contain references to objects of another class
- **Dependency** is another name for the *uses* relationship
- There are **UML notations** for inheritance, interface implementation, aggregation, and dependency
- Use **javadoc** comments (with the method bodies left blank) to record the behavior of classes