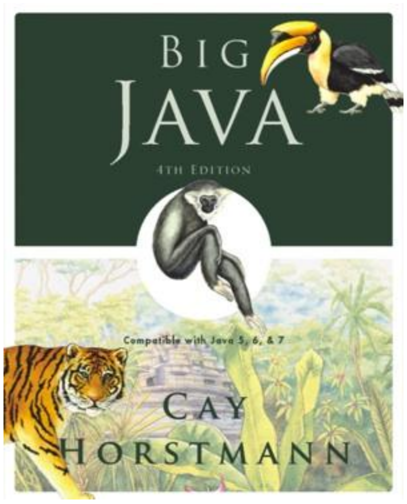


Chapter 12 – Object-Oriented Design

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 supposed 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
- Program Run: Completed program

Testing

- Run tests to verify the program works correctly
- Program Run: 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

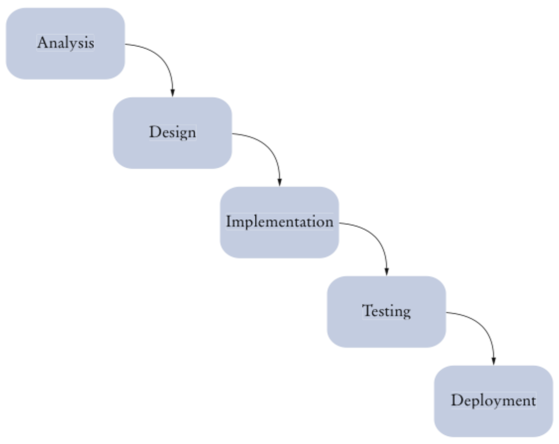


Figure 1 The Waterfall Model

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

The Spiral Model

- Problem: Can lead to many iterations, and process can take too long to complete

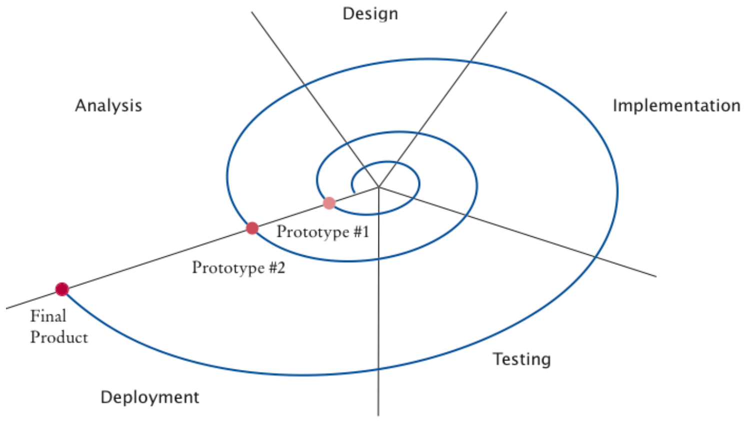


Figure 2 A Spiral Model

Activity Levels in the Rational Unified Process

Development process methodology by the inventors of UML

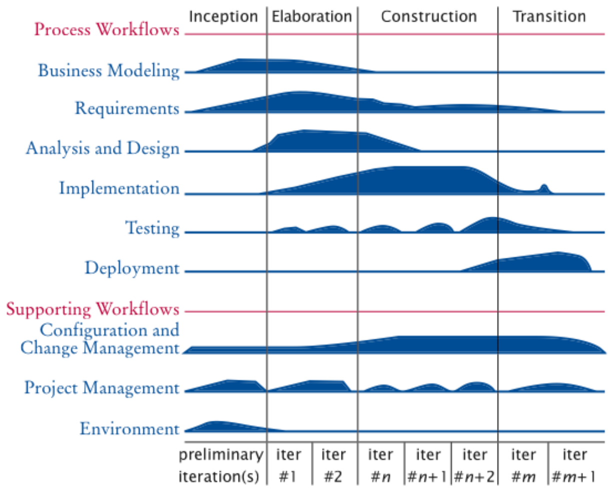


Figure 3 Activity Levels in the Rational Unified Process Methodology

Extreme Programming

- Strives for simplicity
- Removes formal structure
- Focuses on best practices

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.

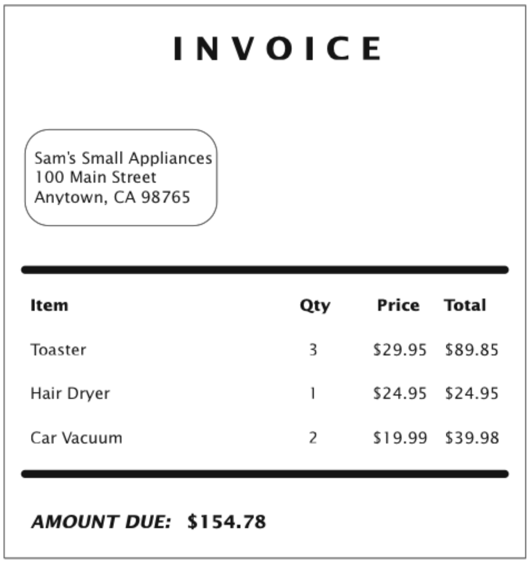
Object-Oriented Design

1. Discover classes
2. Determine responsibilities of each class
3. 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

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

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 **c**lass, its **r**esponsibilities, and its **c**ollaborators
- 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

Continued

CRC Card

- Indicate what other classes are needed to fulfill responsibility (collaborators)

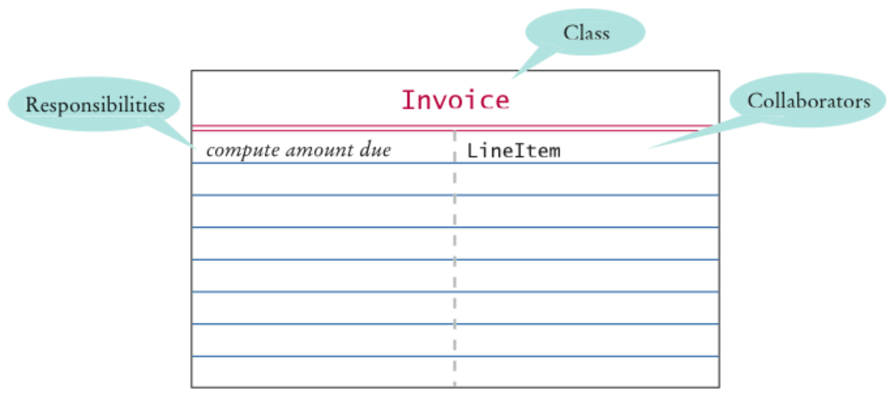


Figure 5 A CRC Card

Self Check 12.4

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

Answer: `PrintStream`

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`?*
    - *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:*

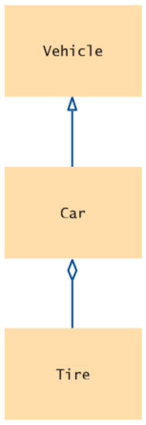
```
class Tire
{
   ...
   private String rating;
   private Circle boundary;
}
```

- 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.

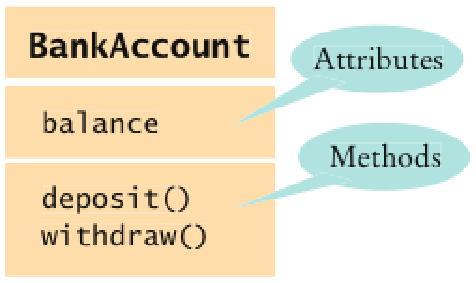
Self Check 12.9

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

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

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

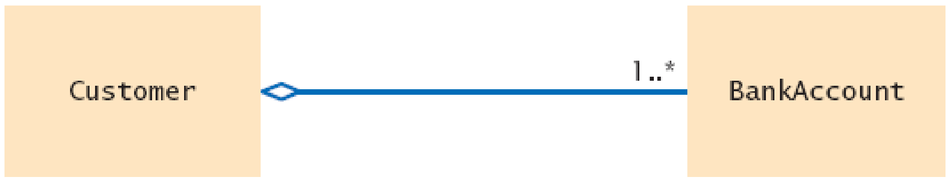
Attributes and Methods in UML Diagrams



Attributes and Methods in a Class Diagram

Multiplicities

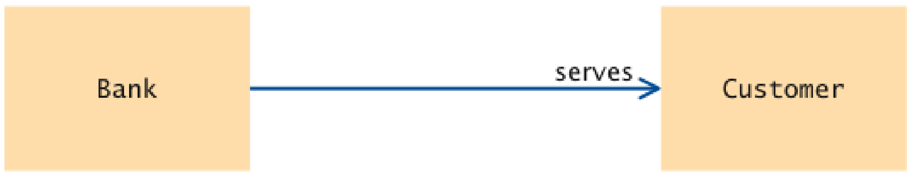
- any number (zero or more): *
- one or more: 1..*
- zero or one: 0..1
- exactly one: 1



An Aggregation Relationship with Multiplicities

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

Case Study: 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

Case Study: Sample Invoice

```
                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
```

Case Study: Printing an Invoice — CRC Cards

- Discover classes
- Nouns are possible classes:

```
Invoice
Address
LineItem
Product
Description
Price
Quantity
Total
Amount Due
```

Case Study: Printing an Invoice — CRC Cards

- Analyze classes:

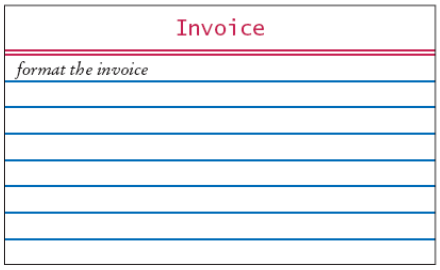
```
Invoice
Address
LineItem     // Records the product and the quantity
Product
Description  // variable of the Product class
Price        // variable of the Product class
Quantity     // Not an attribute of a Product
Total        // Computed – not stored anywhere
Amount Due   // Computed – not stored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

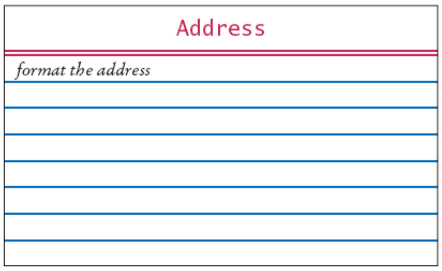
- Classes after a process of elimination:

```
Invoice
Address
LineItem
Product
```

CRC Cards for Printing Invoice

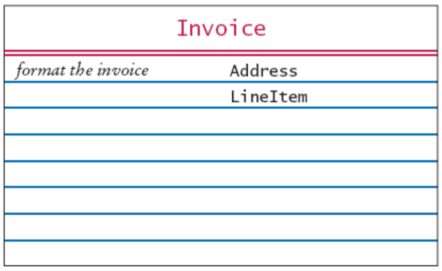
`Invoice` and `Address` must be able to format themselves:





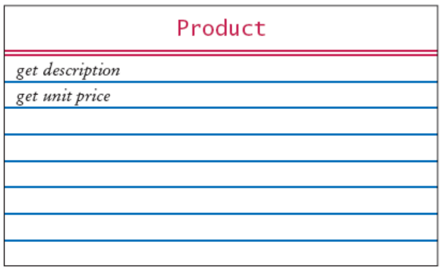
CRC Cards for Printing Invoice

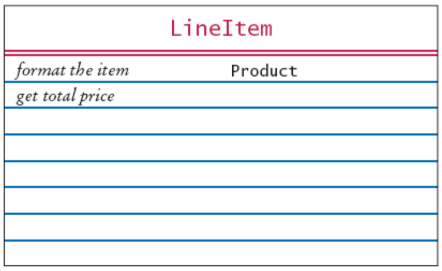
Add collaborators to invoice card:



CRC Cards for Printing Invoice

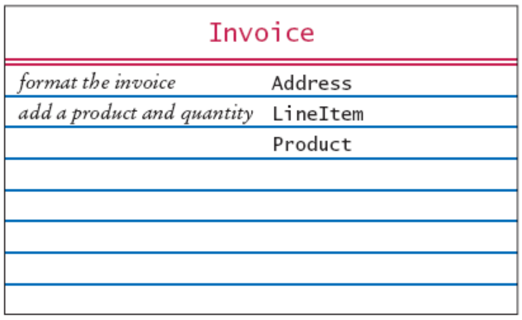
`Product` and `LineItem` CRC cards:





CRC Cards for Printing Invoice

Invoice must be populated with products and quantities:



Invoice	
format the invoice	Address
add a product and quantity	LineItem
	Product

Printing an Invoice — UML Diagrams

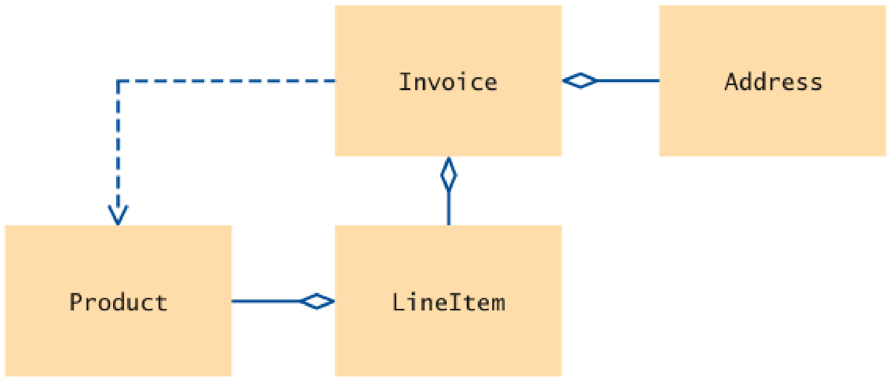


Figure 7 The Relationships Between the Invoice Classes

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*