Object-Oriented Design

Objectives

° To become familiar with the process of program development
° To the relationship types: association, aggregation, composition, strong inheritance, and weak inheritance
° To declare classes to represent the relationships among the classes
° To design systems by identifying the classes and discovering the relationships among these classes
° To design classes that follow the class-design guidelines
Software Development Process

- Software development is an engineering process.

Requirement Specification

- A formal process that seeks to understand the problem and document in detail what the software system needs to do. This phase involves close interaction between users and designers.

Most of the examples in this book are simple, and their requirements are clearly stated. In the real world, however, problems are not well defined. We need to study a problem carefully to identify its requirements.
**System Analysis**

Seeks to analyze the business process in terms of data flow, and to identify the system’s input and output. Part of the analysis entails modeling the system’s behavior. The model is intended to capture the essential elements of the system and to define services to the system.

**System Design**

The process of designing the system’s components. This phase involves the use of many levels of abstraction to decompose the problem into manageable components, identify classes and interfaces, and establish relationships among the classes and interfaces.
**Implementation**

The process of translating the system design into programs. Separate programs are written for each component and put to work together. This phase requires the use of a programming language like Java. The implementation involves coding, testing, and debugging.

**Discovering Class Relationship**

- There are four common relationships among classes
  - Association
  - Aggregation
  - Composition
  - Inheritance
Relationship between Classes - Association

- **Association** represents a general binary relationship that describes an activity between two classes.
  - A student taking a course is an association between the Student class and the Course class.

- **An Association** is implemented using data fields and methods.
  ```java
  public class Student {
      /** Data fields */
      private Course[] courseList;
      /** Constructors */
      /** Methods */
  }

  public class Course {
      /** Data fields */
      private Student[]classList;
      private Faculty faculty
      /** Constructors */
      /** Methods */
  }

  public class Faculty {
      /** Data fields */
      private Course[] courseList;
      /** Constructors */
      /** Methods */
  }
  ```

Aggregation and Composition

- **Aggregation** is a special form of association, which represents an ownership relationship between two classes. Aggregation models the has-a relationship.
  - A person has an address

- If an object is exclusively owned by an aggregated object, the relationship between the object and its aggregated object is referred to as **composition**.
  - A person own a name
Representing Aggregation in Classes

*An aggregation relationship is usually represented as a data field in the aggregated class.*

```java
public class Name {
    /** Data fields */
    public class Person {
        /** Data fields */
        private Name name;
        /** Constructors */
        /** Methods */
    }
    /** Constructors */
    /** Methods */
}
```

Inheritance

*Inheritance models the *is-an-extension-of* relationship between two classes.*

```java
public class Faculty extends Person {
    /** Data fields */
    /** Constructors */
    /** Methods */
}
```

(A) Faculty (B) Person
Weak Inheritance Relationship

A weak is-an-extension-of relationship can be represented using interfaces. For example, the weak is-an-extension-of relationship “students are comparable based on their grades” can be represented by implementing the Comparable interface, as follows:

```java
public class Student extends Person implements Comparable {
    /** Data fields, Constructors, and */
    /** Methods */

    /** Implement the compareTo method */
    public int compareTo(Object object) {
        // ...
    }
}
```

Class Design

The OOP key is to model the application in terms of cooperative objects.

1. Identify classes for the system.
   - Class abstraction: to decompose the problem to a set of related classes

2. Describe attributes and methods in each class.
   - Method abstraction: to design individual classes

3. Establish relationships among classes
   - Inheritance vs. Association

4. Create classes.
Case Study: Borrowing Loans

The case study models borrowing loans to demonstrate how to identify classes, discover the relationships between classes, and apply class abstraction in OOP.

What are classes we need? What are their relationships?

- Person
- Borrower
- Name
- Address
- Loan

Programming is an art more than a science!

Example 11.1 Borrowing Loans

Defined in Example 7.15
Example 11.1 Borrowing Loans, cont.

The following is a test program that uses the classes Name, Person, Address, Borrower, and Loan.

```java
import javax.swing.JOptionPane;
public class BorrowLoan {
    /** Main method */
    public static void main(String[] args) {
        // Create a name
        Name name = new Name("John", 'D', "Smith");
        // Create an address
        Address address = new Address("100 Main", "Savannah", "GA", "31419");
        // Create a loan
        Loan loan = new Loan(5.5, 15, 250000);
        // Create a borrower
        Borrower borrower = new Borrower(name, address);
        borrower.setLoan(loan);
        // Display loan information
        JOptionPane.showMessageDialog(null, borrower.toString());
    }
}
```

Example 11.2 The Rational Class

```
java.lang.Number
+byteValue(): byte
+shortValue(): short
+intValue(): int
+longValue(): long
+floatValue(): float
+doubleValue(): double

java.lang.Comparable
compareTo(Object): int

Rational
+numerator: long
+denominator: long

-Rational()
+Rational(numerator: long, denominator: long)
+getNumerator(): long
+getDenominator(): long
+add(secondRational: Rational): Rational
+multiply(secondRational: Rational): Rational
-subtract(secondRational: Rational): Rational
+divide(secondRational: Rational): Rational
+toString(): String
+gcd(n: long, d: long): long

Add, Subtract, Multiply, Divide
```
Class Design Guidelines

- Designing a Single Class.
- Using Modifiers public, protected, private and static
- Using Inheritance (and Polymorphism) or Aggregation
- Using Interfaces or Abstract Classes

Designing a Class

- A class should describe a single entity or a set of similar operations. A single entity with too many responsibilities can be broken into several classes to separate responsibilities.
  - The String class, StringBuffer class, and StringBuilder class all deal with strings, for example, but have different responsibilities.
- Classes are usually designed for use by many different customers. To make a class useful in a wide range of applications, the class should provide a variety of ways for customization through properties and methods.
- Provide a public no-arg constructor, and, override the equals method and the toString method defined in the Object class whenever possible.
- Follow standard Java programming style and naming conventions. Choose informative names for classes, data fields, and methods. Always place the data declaration before the constructor, and place constructors before methods. Always provide a constructor and initialize variables to avoid programming errors.
Encapsulation: Using Visibility Modifiers

- A class should use the `private` modifier to hide its data from direct access by clients.
  - You can use `public` get methods and set methods to provide users with access to the private data, but only to private data you want the user to see or to modify. A class should also hide methods not intended for client use.
- Make the fields or method `protected` if they are intended for extenders of the class.
  - The constructors in an abstract classes should always be `protected`.
- A property that is shared by all the instances of the class should be declared as a `static` property.

Using Inheritance or Aggregation

- In general, the difference between inheritance and aggregation is the difference between the `is-an-extension-of` relationship and the `has-a` relationship.
- For example, an apple is fruit; thus, you would use inheritance to model the relationship between the classes Apple and Fruit.
- A person has a name; thus, you would use aggregation to model the relationship between the classes Person and Name.
- Sometimes, the choice between inheritance and aggregation is not obvious, e.g., Circle and Cylinder, both choices are fine.
  - If *polymorphism* is desirable, you need to use the inheritance design. If you don’t care about polymorphism, the aggregation design gives more flexibility because the classes are less dependent using aggregation than using inheritance.
Using Interfaces or Abstract Classes

Both interfaces and abstract classes can be used to generalize common features. How do you decide whether to use an interface or an abstract class?

In general, a strong is-an-extension-of relationship that clearly describes a parent-child relationship should be modeled using classes.

- For example, since an orange is a fruit, their relationship should be modeled using class inheritance.

A weak is-an-extension-of relationship, also known as an is-kind-of relationship, indicates that an object possesses a certain property. A weak is-an-extension-of relationship can be modeled using interfaces.

- For example, all strings are comparable, so the String class implements the Comparable interface.

- A circle or a rectangle is a geometric object, for example, so Circle can be designed as a subclass of GeometricObject. Circles are different and comparable based on their radius, for example, so Circle can implement the Comparable interface.

 Interfaces are more flexible than abstract classes, because a subclass can extend only one superclass, but implement any number of interfaces. However, interfaces cannot contain concrete methods.

Reading

- Chapter 11 of the textbook: 11.1 – 11.6
- Review chapters 1-11, and 17