SWE 4201 - OBJECT ORIENTED CONCEPTS I

LAB 06 HANDOUT AND TASKS

Prepared by:
Md. Jubair Ibna Mostafa
Assistant Professor, IUT CSE

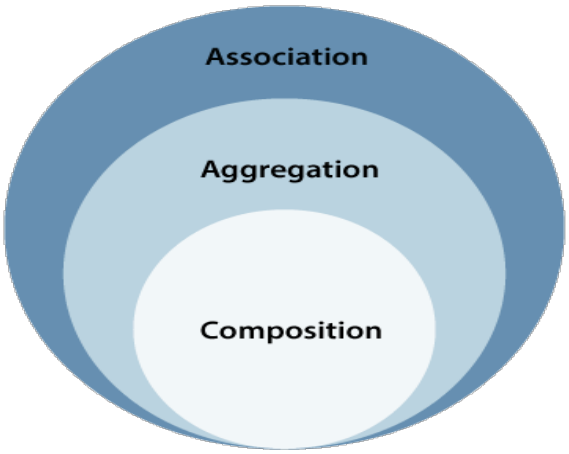
Department of Computer Science and Engineering
Islamic University of Technology

Contents

1 Object Relation

Object relationships define how objects in a system interact and collaborate. Object relationships can be classified into association, aggregation, and composition. Besides, inheritance, dependency, interface implementation, or realization of a contract can also be thought of as object relations.

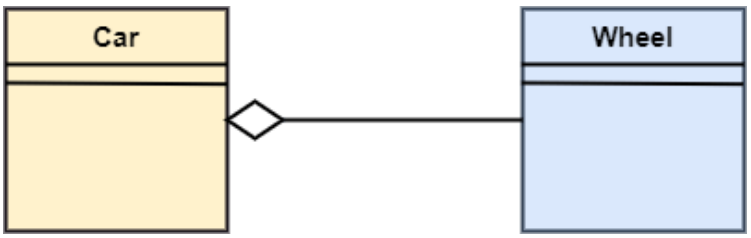
Figure 1: Object Relation



Association: Represents a generic connection or link between two or more classes. Characteristics: Objects of one class are related to objects of another class, often indicating some form of interaction or connection. Example: A Teacher class associated with a Student class.

Aggregation: Represents a whole-part relationship where one class (the whole) contains another class (the part). Characteristics: The part has a more independent lifecycle and can exist without the whole. The diamond-shaped line in UML denotes aggregation. Example: A University class containing Departments. A Car class containing Wheels.

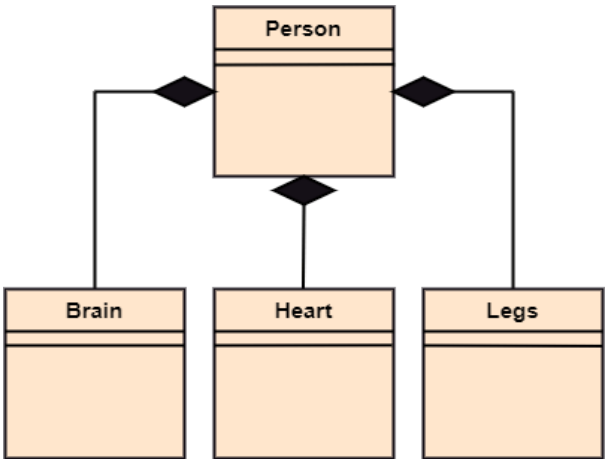
Figure 2: Aggregation



Composition: Similar to aggregation but with a stronger relationship. The part is entirely

dependent on the whole. Characteristics: If the whole is destroyed, the parts are also destroyed. Denoted by a filled diamond in UML. Example: A Person class composed with Brain class, Heart class, and Legs class.

Figure 3: Composition



1.1 Aggregation

```
class Department {
    private String name;

    public Department(String name) {
        this.name = name;
    }

    public String getName(){
        return this.name;
    }
}

class University {
    private String name;
    private Department department;

```

```
17      public University(String name, Department department) {
18          this.name = name;
19          this.department = department;
20      }
21
22      public Department getDepartment() {
23          return department;
24      }
25  }
```

Listing 1: An Example of Aggregation

1.2 Composition

```
1   class Department {
2       private String name;
3
4       public Department(String name) {
5           this.name = name;
6       }
7
8       public String getName(){
9           return this.name;
10      }
11  }
12
13  class University {
14      private String name;
15      private Department department;
16
17      public University(String name, String departmentName) {
18          this.name = name;
19          this.department = new Department(departmentName);
20      }
21
```

```
22      public Department getDepartment() {
23          return department;
24      }
25  }
```

Listing 2: An Example of Aggregation

2 DELEGATION

Delegation is a design pattern in object-oriented programming where an object passes on a task or responsibility to another object. In this pattern, an object, known as the delegator, doesn't handle the task itself but delegates it to another object, known as the delegate, which is specialized in performing that task. This allows for better code organization, reuse, and flexibility. Delegation is often used to achieve composition over inheritance, promoting a more modular and maintainable code structure.

In delegation, the delegator maintains a reference to the delegate and invokes its methods, forwarding the request. The delegator may choose to perform some actions before or after delegating to the delegate, effectively controlling the behavior of the delegated task. This pattern is beneficial when you want to separate concerns, promote code reuse, and create a more flexible and extensible architecture.

Delegation is commonly used in scenarios where a class wants to utilize the functionality of another class without inheriting from it. It encourages a "has-a" relationship instead of an "is-a" relationship, fostering a more modular and loosely coupled design.

Example of Delegation

```
1  // Delegator Interface
2  interface Printer {
3      void print(String document);
4  }
5
6  // Delegate - CanonPrinter
7  class CanonPrinter implements Printer {
```

```
8       @Override
9       public void print(String document) {
10          System.out.println("Printing document using Canon Printer: " +
        document);
11      }
12  }
13
14  // Delegate - EpsonPrinter
15  class EpsonPrinter implements Printer {
16      @Override
17      public void print(String document) {
18          System.out.println("Printing document using Epson Printer: " +
        document);
19      }
20  }
21
22  // Delegate - HpPrinter
23  class HpPrinter implements Printer {
24      @Override
25      public void print(String document) {
26          System.out.println("Printing document using HP Printer: " +
        document);
27      }
28  }
29
30  // Delegator
31  class PrinterController implements Printer {
32      private Printer printer;
33
34      public void setPrinter(Printer printer) {
35          this.printer = printer;
36      }
37
38      @Override
39      public void print(String document) {
```

```
40          if (printer != null) {
41              printer.print(document);
42          } else {
43              System.out.println("No printer selected. Please set a
    printer.");
44          }
45      }
46  }
47
48  // Client Code
49  public class DelegationExample {
50      public static void main(String[] args) {
51          CanonPrinter canonPrinter = new CanonPrinter();
52          EpsonPrinter epsonPrinter = new EpsonPrinter();
53          HpPrinter hpPrinter = new HpPrinter();
54
55          PrinterController printerController = new PrinterController();
56          printerController.setPrinter(canonPrinter);
57          printerController.print("Sample Document");
58
59          // Change the printer to HpPrinter
60          printerController.setPrinter(hpPrinter);
61          printerController.print("Another Document");
62      }
63  }
```

Listing 3: An Example of Delegation

In Listing 3 represents the delegation process for printing task where PrintController is the delegator, and other classes are delegates.

3 TASKS

3.1 Scenario 1 Marks 10

You are developing a system for an online marketplace that connects sellers and buyers for various products. The system should model entities such as Seller, Product, Customer, and Review. you can think of some attributes for every class. For example, a seller has a name and a list of products. A product has a name, price, description, and category. A sellTo method receives a customer object. A customer has a name and a list of reviews and a purchase method receives a product object. A review has a message and rating value. You can add additional methods if you want.

Your task is to create different types of relationships (association, aggregation, composition, dependency).

3.2 Scenario 2 Marks 10

Logging refers to the practice of recording or capturing relevant information about the execution of a program or system. This information, known as log data or logs, typically includes events, messages, warnings, errors, and other details that can provide insights into the software's behavior.

Your task is to create multiple loggers with different logging strategies, such as logging into the console, logging into a file, and sending logs via email. Every type of logger has one contract that is they can log a message.

The challenge lies in designing the system to accommodate new logging strategies in the future without modifying the existing logging classes or interfaces. Your solution should promote flexibility and extensibility, allowing for the seamless integration of additional loggers.

Use the delegation concept to solve the problem.

4 REFERENCE

- https://www.javatpoint.com/uml-association-vs-aggregation-vs-composition