Advanced Java

LESSON 10: STRATEGY PATTERN

Creative Commons

Attribution 4.0 International (CC BY 4.0)

This workforce product was funded by a grant awarded by the U.S. Department of Labor's Employment and Training Administration. The product was created by the grantee and does not necessarily reflect the official position of the U.S. Department of Labor. The U.S. Department of Labor makes no guarantees, warranties, or assurances of any kind, express or implied, with respect to such information, including any information on linked sites and including, but not limited to, accuracy of the information or its completeness, timeliness, usefulness, adequacy, continued availability, or ownership. This is an equal opportunity program. Assistive technologies are available upon request and include Voice/TTY (771 or 800-947-6644).

Agenda

- **Design Patterns**
- **Association, Composition and Aggregation**
- **Strategy Pattern**

Design Patterns

Design Patterns

A general, reusable solution to a commonly occurring problem within a given context in software design.

Creational patterns

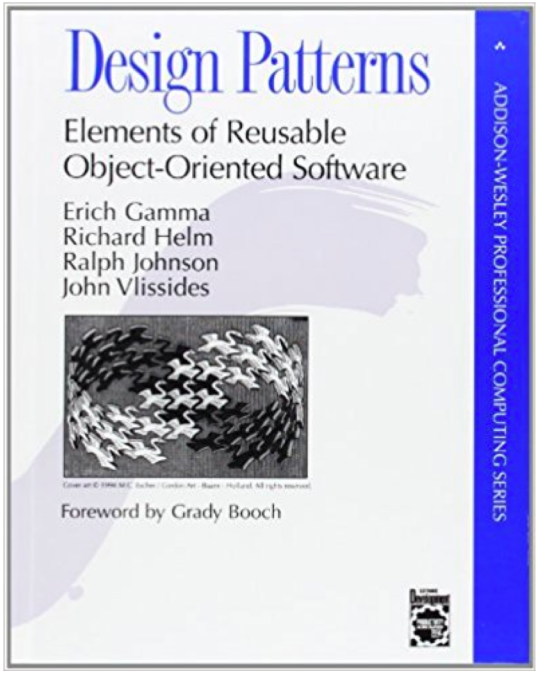
- Singlton
- Factory

Structural patterns

- Decorator
- Façade

Behavioral patterns

- Memento
- Observer
- Strategy



1994

Association, Composition and Aggregation

Association, Composition and Aggregation

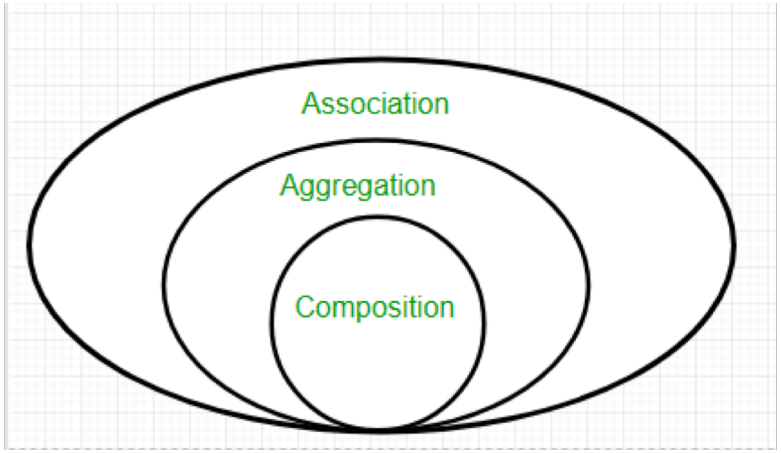
Inheritance is an "is-a" relationship. Composition is a "has-a". You do composition by having an instance of another class C as a field of your class, instead of extending C .

Stack Overflow

Association, Composition and Aggregation

Association

- Association is relation between two separate classes which establishes through their Objects. Association can be one-to-one, one-to-many, many-to-one, many-to-many.



Geeksforgeeks.org

https://www.geeksforgeeks.org/association-composition-aggregation-java/

Association, Composition and Aggregation

Aggregation

It is a special form of Association where:

- It represents Has-A relationship.
- It is a unidirectional association i.e. a one way relationship. For example, department can have students but vice versa is not possible and thus unidirectional in nature.
- In Aggregation, both the entries can survive individually which means ending one entity will not effect the other entity

Geeksforgeeks.org

Association, Composition and Aggregation

Aggregation vs Composition

1. **Dependency:** Aggregation implies a relationship where the child can exist independently of the parent. For example, Bank and Employee, delete the Bank and the Employee still exist. whereas Composition implies a relationship where the child cannot exist independent of the parent. Example: Human and heart, heart don't exist separate to a Human
2. **Type of Relationship:** Aggregation relation is "has-a" and composition is "part-of" relation.
3. **Type of association:** Composition is a strong Association whereas Aggregation is a weak Association.

Geeksforgeeks.org

Association, Composition and Aggregation

Before Refactoring

```
public class Dessert {
    private String name;
    private int calories;
    private boolean isSweat;

    public Dessert(String name, int calories, boolean isSweat) {
        this.name = name;
        this.calories = calories;
        this.isSweat = isSweat;
    }
    public String getName() {
        return name;
    }
    public int getCalories() {
        return calories;
    }
    public boolean isSweat() {
        return isSweat;
    }
}
```

Association, Composition and Aggregation

Aggregation – "Has A"

```
public class Food {
    private String name;
    private int calories;
    public Food(String name,
                int caloriest) {
        this.name = name;
        this.calories = calories;
    }
    public String getName() {
        return name;
    }
    public int getCalories() {
        return calories;
    }
}
```

```
public class Dessert {
    private boolean isSweat;
    private Food food;
    public Dessert(String name,
            int calories, boolean isSweat) {
        food = new Food(name, calories);
        this.isSweat = isSweat;
    }
    public String getName() {
        return food.getName();
    }
    public int getCalories() {
        return food.getCalories();
    }
    public boolean isSweat() {
        return isSweat;
    }
}
```

In-Class Activity

1. Download, "CompisitionExample" project from Blackboard.
2. Read the "ReadMe.txt" file and follow the instructions.

Strategy Pattern

Strategy Pattern

Law of Demeter (LoD) or **principle of least knowledge**

A design guideline for developing software, particularly object-oriented programs. In its general form, the LoD is a specific case of loose coupling.

- Each unit should have only limited knowledge about other units: only units "closely" related to the current unit.
- Each unit should only talk to its friends; don't talk to strangers.
- Only talk to your immediate friends.

Strategy Pattern

In computer programming, the **Strategy Pattern** (also known as the **Policy Pattern**) is a behavioral software design pattern that enables selecting an algorithm at runtime.

The strategy pattern

1. defines a family of algorithms,
2. encapsulates each algorithm, and
3. makes the algorithms interchangeable within that family.

Wikipedia.org

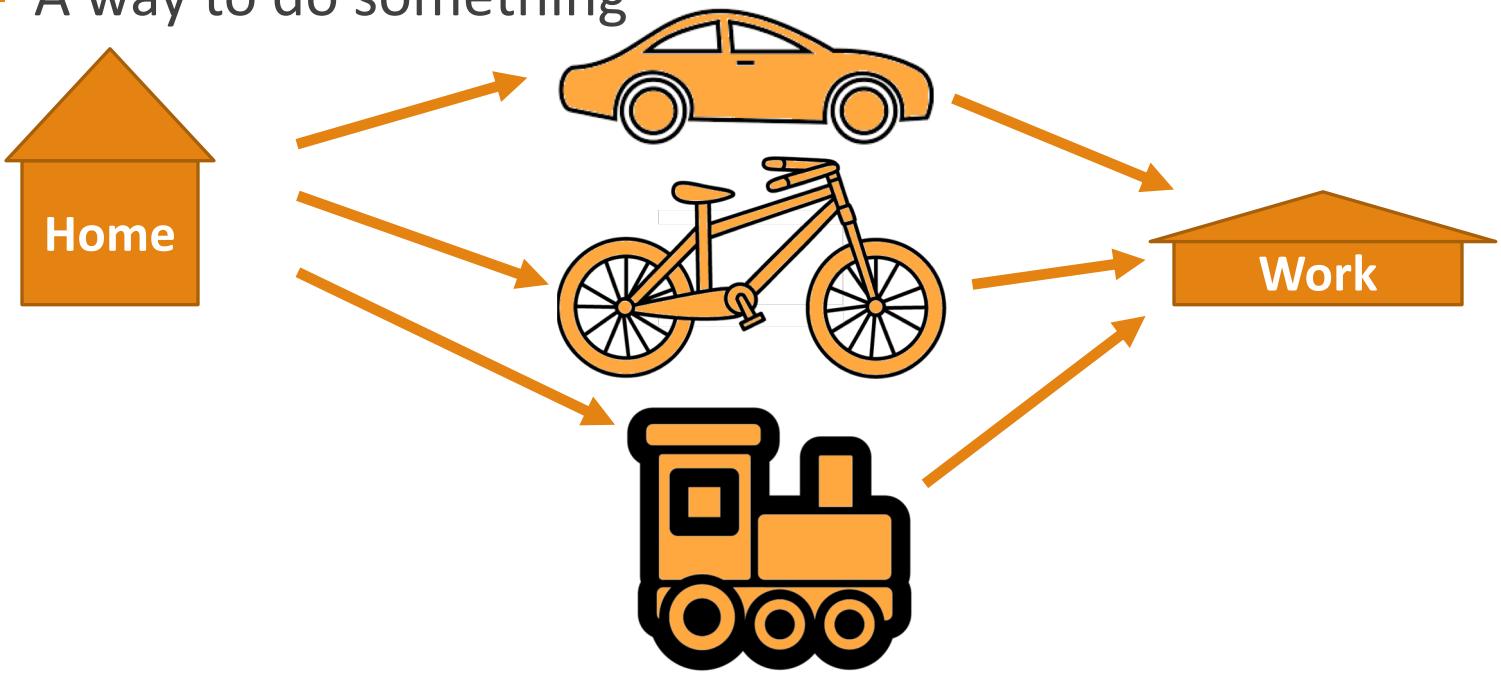
Strategy Pattern

Strategy – "Get to Work"

- Strategy #1 – Drive
- Strategy #2 – Bike
- Strategy #3 –Train

What is a Strategy?

- Plan of Action to achieve a specific Goal
- An algorithm
- A way to do something



Strategy Pattern

```
public interface TravelStrategy {
    public void goToWork();
}

public class DriveStartegy implements TravelStrategy {
    @Override
    public void goToWork() {
        System.out.println("Drove to work");
    }
}

public class BikedStrategy implements  TravelStrategy {
    @Override
    public void goToWork() {
        System.out.println("Biked to work");
    }
}

public class TrainStrategy implements TravelStrategy {
    @Override
    public void goToWork() {
        System.out.println("Road Train to Work.");
    }
}
```

Strategy Pattern

```
public class Day {
    private String dayOfWeek;
    private TravelStrategy travel;

    public Day(String dayOfWeek, TravelStrategy travel) {
        this.dayOfWeek = dayOfWeek;
        this.travel = travel;
    }
    public String getDayOfWeek() {
        return dayOfWeek;
    }
    public TravelStrategy getTravel() {
        return travel;
    }
    @Override
    public String toString() {
        return "Day{" +
                "dayOfWeek='" + dayOfWeek + '\'' +
                ", travel=" + travel +
                '}';
    }
}
```

Strategy Pattern

```
import java.util.ArrayList;
import java.util.List;

public class Main {

    public static void main(String[] args) {
        List<Day> week = new ArrayList<>();
        week.add(new Day("Monday", new BikedStrategy()));
        week.add(new Day("Tuesday", new TrainStrategy()));
        week.add(new Day("Wesnesday",
                                    new TrainStrategy()));
        week.add(new Day("Thursday", new DriveStartegy()));
        week.add(new Day("Friday", new DriveStartegy()));

        for (Day d: week) {
            d.getTravel().goToWork();
        }
    }
}
```

In-Class Activity

1. Download, "StoreStrategyLab" project from Blackboard.
2. Read the "ReadMe.txt" file and follow the instructions.