# Assignment 9

**Title:** Identify and Implement GOF pattern

## Problem Statement:

- Identification and Implementation of GOF pattern for parking management system.
- Apply any two GOF patterns to refine Design Model for a given problem description using effective UML 2 diagrams and implement them with a suitable object oriented language.

## Objective:

- To Study GOF Patterns.
- To identify applicability of GOF in the system
- Implement a System with patterns.

## Theory:

### Strategy Design Pattern

In Software Engineering, the strategy pattern (also known as the policy pattern) is a software design pattern that enables an algorithm's behavior to be selected at runtime.

The GoF Design Patterns are broken into three categories: Creational Patterns for the creation of objects; Structural Patterns to provide a relationship between objects; and finally, Behavioral Patterns to help define how objects interact.

### I. Creational Design Patterns

- **Abstract Factory**: Allows the creation of objects without specifying their concrete type.
- **Builder**: Uses to create complex objects.
- **Factory Method**: Creates objects without specifying the exact class to create.
- **Prototype**: Creates a new object from an existing object.
- **Singleton**: Ensures only one instance of an object is created.

### II. Structural Design Patterns

- **Adapter**: Allows for two incompatible classes to work together by wrapping an interface around one of the existing classes.

- **Bridge**: Decouples an abstraction so two classes can vary independently.
- **Composite**: Takes a group of objects into a single object.
- **Decorator**: Allows for an object's behavior to be extended dynamically at run time.
- **Facade**: Provides a simple interface to a more complex underlying object.
- **Flyweight**: Reduces the cost of complex object models.
- **Proxy**: Provides a placeholder interface to an underlying object to control access, reduce cost, or reduce complexity.

### III. Behavior Design Patterns

- **Chain of Responsibility**: Delegates command to a chain of processing objects.
- **Command**: Creates objects which encapsulate actions and parameters.
- **Interpreter**: Implements a specialized language.
- **Iterator**: Accesses the elements of an object sequentially without exposing its underlying representation.
- **Mediator**: Allows loose coupling between classes by being the only class that has detailed knowledge of their methods.
- **Memento**: Provides the ability to restore an object to its previous state.
- **Observer**: Is a publish/subscribe pattern which allows a number of observer objects to see an event.
- **State**: Allows an object to alter its behavior when its internal state changes.
- **Strategy**: Allows one of a family of algorithms to be selected on-the-fly at run-time.
- **Template Method**: Defines the skeleton of an algorithm as an abstract class, allowing its subclasses to provide concrete behavior.
- **Visitor**: Separates an algorithm from an object structure by moving the hierarchy of methods into one object.

## Implementation:

### 1. Abstract Factory Design Pattern:

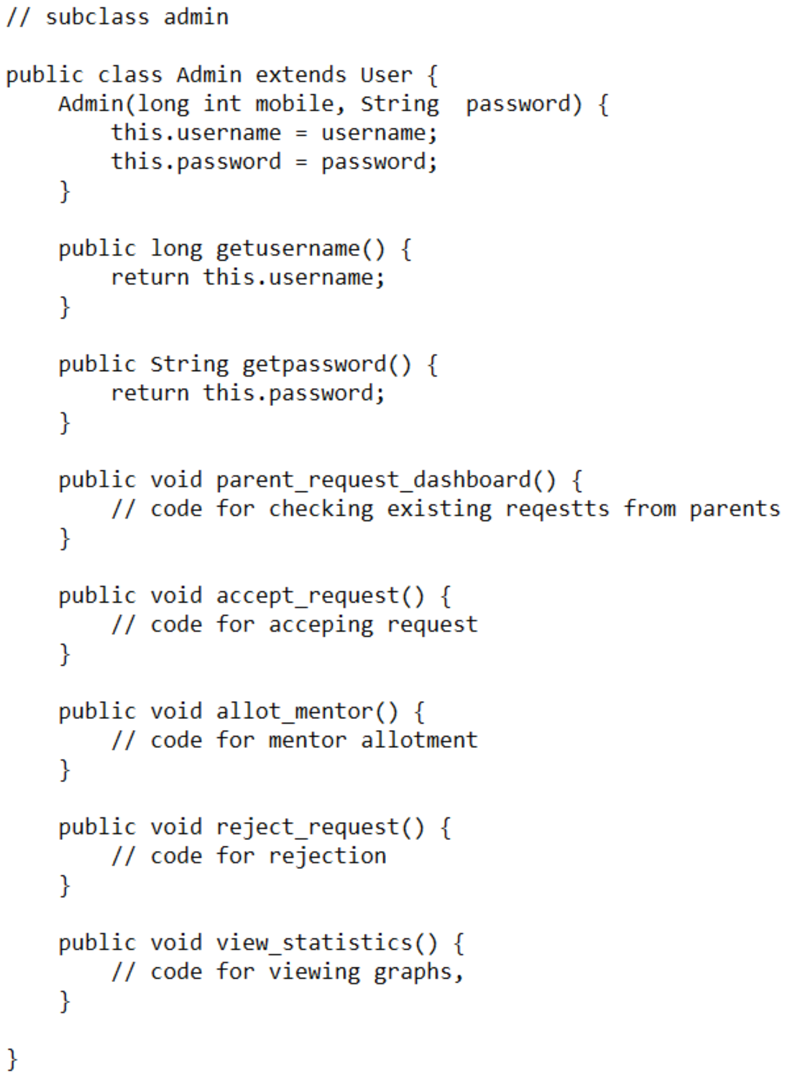
Abstract Factory design pattern comes under the creational pattern as this pattern provides one of the best ways to create an object. In Abstract Factory pattern an interface is responsible for creating a factory of related objects without explicitly specifying their classes. Each generated factory can give the objects as per the Factory pattern.

Abstract Factory design pattern provides an approach to code for interface rather than implementation. Abstract Factory pattern is "factory of factories" and can be easily extended to accommodate more products. Abstract Factory pattern is robust and avoids conditional logic of Factory pattern.

**Example:**

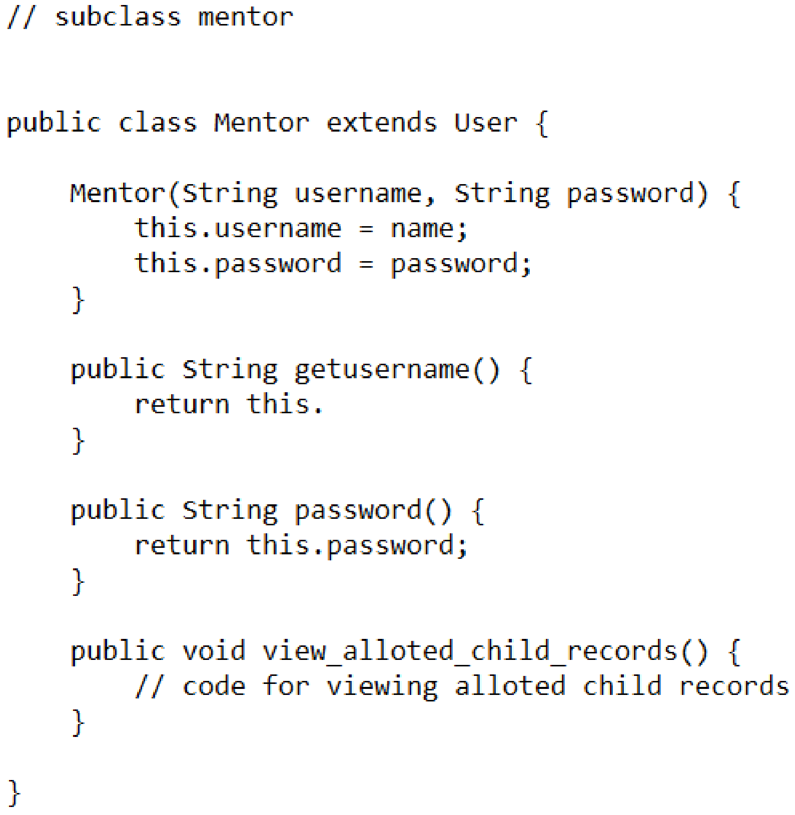
- Abstract factory design pattern is used in this system for User class which is an abstract class.
- It includes four subclasses which are Admin, Donor , parent and mentor
- Then created a factory to generate object of concrete class i.e. UserSelectFactory.
- Use a factory to get objects of concrete class by passing information as shown in Userfactory.

**Subclass Admin:**



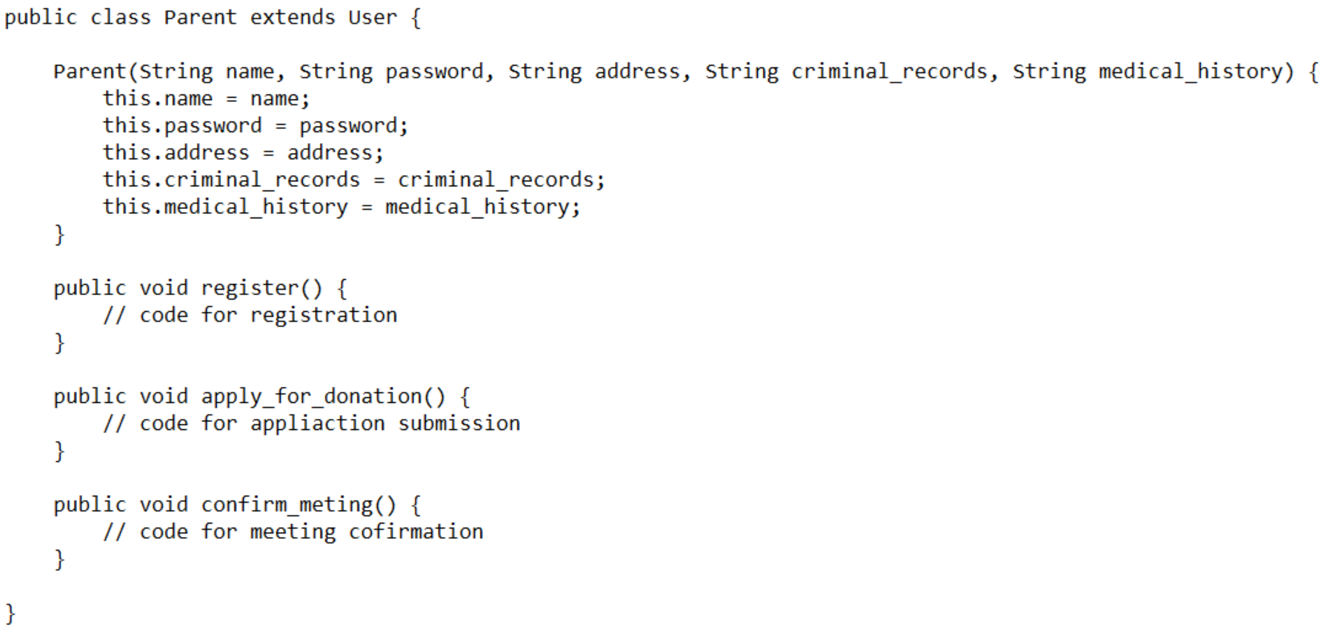
```
// subclass admin

public class Admin extends User {
    Admin(long int mobile, String  password) {
        this.username = username;
        this.password = password;
    }

    public long getusername() {
        return this.username;
    }

    public String getpassword() {
        return this.password;
    }

    public void parent_request_dashboard() {
        // code for checking existing reqestts from parents
    }

    public void accept_request() {
        // code for acceping request
    }

    public void allot_mentor() {
        // code for mentor allotment
    }

    public void reject_request() {
        // code for rejection
    }

    public void view_statistics() {
        // code for viewing graphs,
    }

}
```

## Subclass Mentor



```
// subclass mentor


public class Mentor extends User {

    Mentor(String username, String password) {
        this.username = name;
        this.password = password;
    }

    public String getusername() {
        return this.
    }

    public String password() {
        return this.password;
    }

    public void view_alloted_child_records() {
        // code for viewing alloted child records
    }

}
```

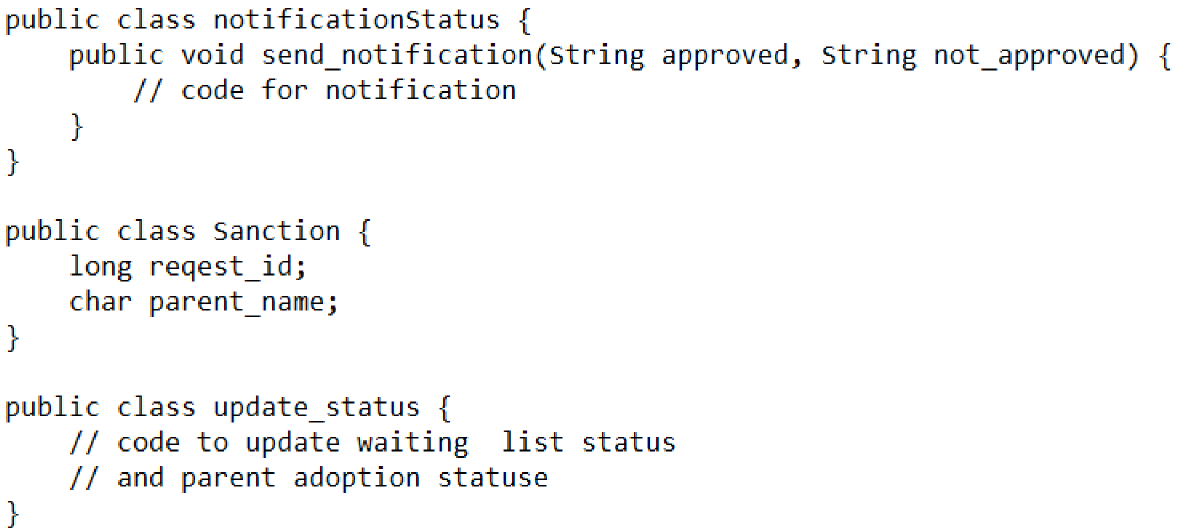
## Subclass Parent



```
public class Parent extends User {

    Parent(String name, String password, String address, String criminal_records, String medical_history) {
        this.name = name;
        this.password = password;
        this.address = address;
        this.criminal_records = criminal_records;
        this.medical_history = medical_history;
    }

    public void register() {
        // code for registration
    }

    public void apply_for_donation() {
        // code for appliaction submission
    }

    public void confirm_meting() {
        // code for meeting cofirmation
    }

}
```

# Observer Design Pattern

- The observer pattern is a software design pattern in which an object, named the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods.
- In our system, the status of sanctioning and updation works in coherence with the observer design pattern.



```
public class notificationStatus {
    public void send_notification(String approved, String not_approved) {
        // code for notification
    }
}

public class Sanction {
    long reqest_id;
    char parent_name;
}

public class update_status {
    // code to update waiting  list status
    // and parent adoption statuse
}
```

**Conclusion:** In this assignment, we identified different GOF patterns for the vehicle parking management system using UML notations and implemented them using Java.