Assignment 8
Identification and Implementation of GRASP pattern

Aim:
Apply any four GRASP patterns to refine the Design Model for a given problem description. Using effective UML 2 diagrams and implementing them with a suitable object-oriented language.

Problem Statement:
- Identification and Implementation of GRASP pattern
- Apply any two GRASP pattern to refine the Design Model for a given problem description
- Using effective UML 2 diagrams and implement them with a suitable object-oriented language

Objective:
- To Study GRASP patterns.
- To implement a system using any two GRASP Patterns

Theory:

GRASP or General Responsibility Assignment Software Principles helps guide object-oriented design by clearly outlining WHO does WHAT. Which object or class is responsible for what action or role? GRASP also helps us define how classes work with one another. The key point of GRASP is to have clean, clear, efficient, and understandable code.

Within GRASP there are nine different principles:
1. Creator
2. Controller
3. Information Expert
4. Low Coupling
5. High Cohesion
6. Indirection
7. Polymorphism
8. Protected Variations
9. Pure Fabrication

Creator:
- The Creator defines WHO instantiates WHAT object. In object-oriented design lingo, we need to ask the question of who creates an object A. The solution is that we give class B the role of instantiating (creating an instance of) a class A if:
- B contains A.
- B uses most of A’s features.
- B can initialize A.

Controller:

- Deals with how to delegate the request from the UI layer objects to domain layer objects.
- When a request comes from a UI layer object, the Controller pattern helps us in determining what is that first object that receives the message from the UI layer objects.
- This object is called a controller object which receives requests from UI layer objects and then controls/coordinates with other objects of the domain layer to fulfill the request.
- It delegates the work to other classes and coordinates the overall activity.

Information Expert:

- Information Expert is a principle used to determine where to delegate responsibilities. These responsibilities include methods, computed fields, and so on.
- Using this principle of information expert, a general approach to assigning responsibilities is to look at a given responsibility, determine the information needed to fulfill it, and then determine where that information is stored.
- Information experts will lead to placing the responsibility on the class with the most information required to fulfill it.
- This pattern stated that we need to assign responsibilities to the right expert.

Low Coupling:

- Coupling is a measure of how strongly one element is connected to, has knowledge of, or relies on other elements. Low Coupling is an evaluative pattern that dictates how to assign responsibilities to support.
- Lower dependency between the classes
- Change in one class having a lower impact on other classes
- Higher reuse potential
- It can be described as following the path of least resistance.

High Cohesion:

- It is important to have clean code. Objects need to be manageable, easy to maintain, and have clearly-stated properties and objectives. This is High Cohesion which includes defined purposes of classes, ability to reuse code, and keeping responsibility to one unit. High Cohesion, Low Coupling, and clearly defined responsibilities go together. To achieve High Cohesion, a class should have ONE job.

Indirection:

- To support lower coupling between objects, we look for indirection, which is creating an intersection object between two or more objects so they aren't connected.
- Indirection and polymorphism go hand in hand.

Polymorphism:

- Polymorphism guides us in deciding which object is responsible for handling those varying elements.
- It means that one thing can be performed in different ways.

Protected Variation:

- The protected variations pattern protects elements from the variations on other elements (objects, systems, subsystems) by wrapping the focus of instability with an interface and using polymorphism to create various implementations of this interface.

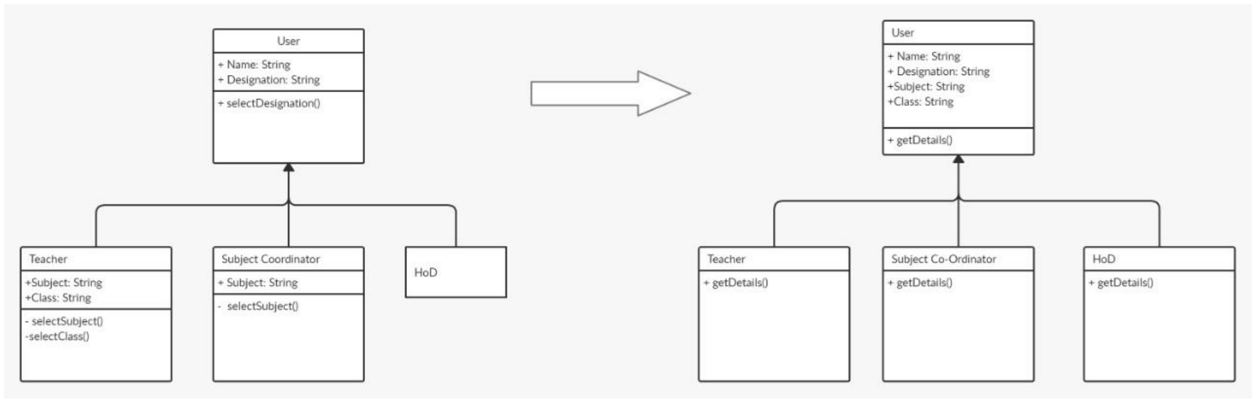
Pure Fabrication:

- A pure fabrication is a class that does not represent a concept in the problem domain, specially made up to achieve low coupling, high cohesion, and the reuse potential thereof derived (when a solution presented by the information expert pattern does not). This kind of class is called a ‘service’ in a domain-driven design.

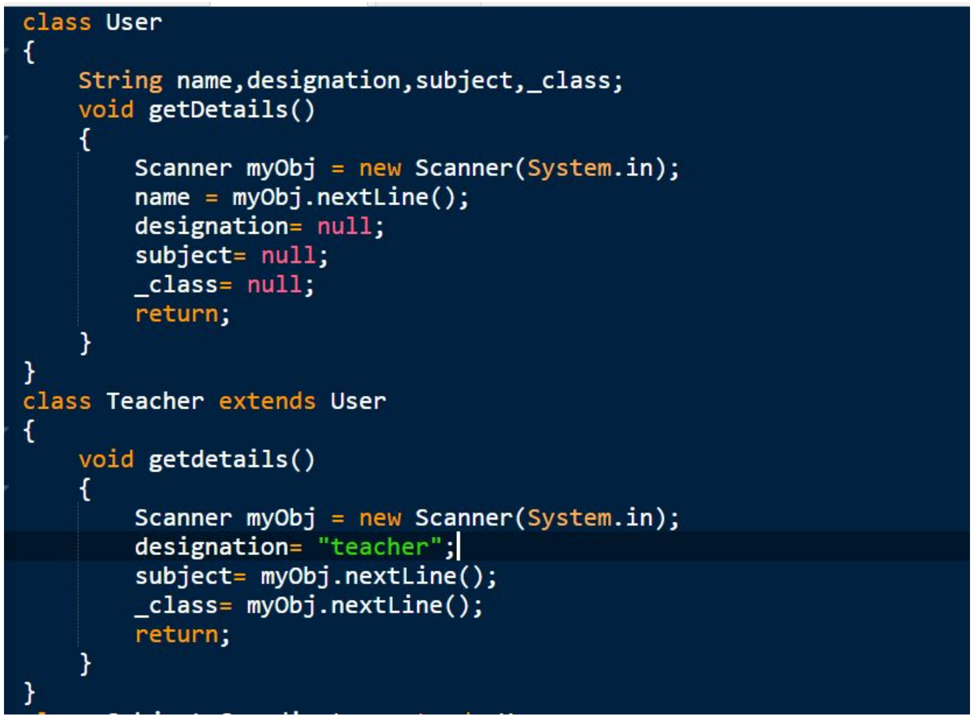
Implementation

Polymorphism

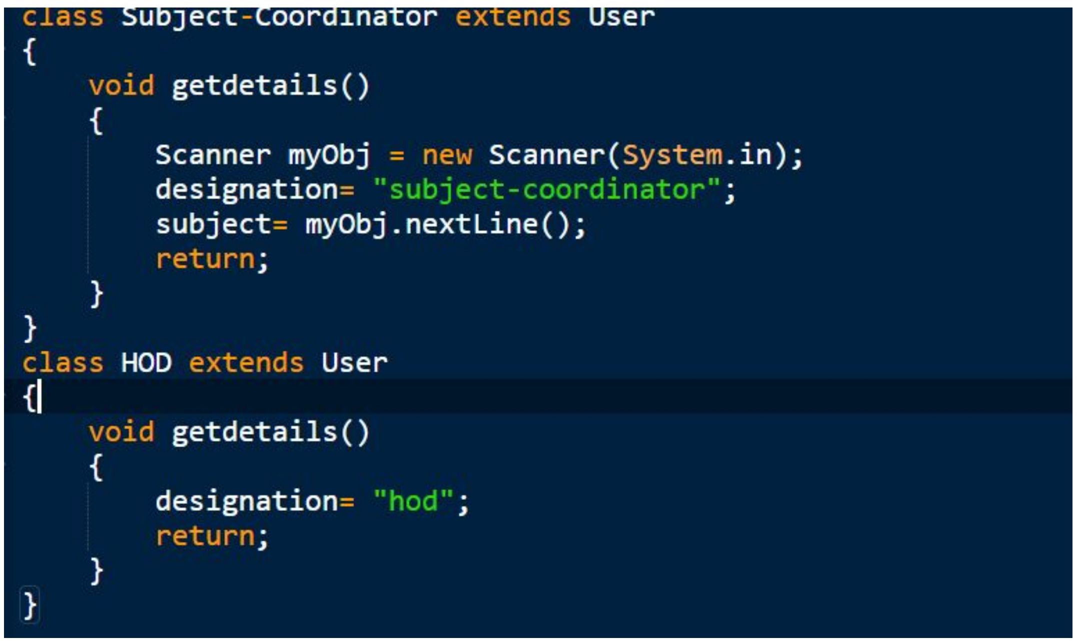
Problem : How to handle details based on different users?
Solutions:When related alternatives or behaviors vary by type (user), assign responsibility for the behavior (using polymorphism operations) to the types for which the behavior varies.



In this project:
● User, Teacher, Subject-Coordinator and HOD are classes which use polymorphism.
● The getDetails() varies by the type of user, so we assign that responsibility to the subclasses Teacher, Subject Coordinator and HOD.



```
class User
{
    String name,designation,subject,_class;
    void getDetails()
    {
        Scanner myObj = new Scanner(System.in);
        name = myObj.nextLine();
        designation= null;
        subject= null;
        _class= null;
        return;
    }
}
class Teacher extends User
{
    void getdetails()
    {
        Scanner myObj = new Scanner(System.in);
        designation= "teacher";
        subject= myObj.nextLine();
        _class= myObj.nextLine();
        return;
    }
}
```

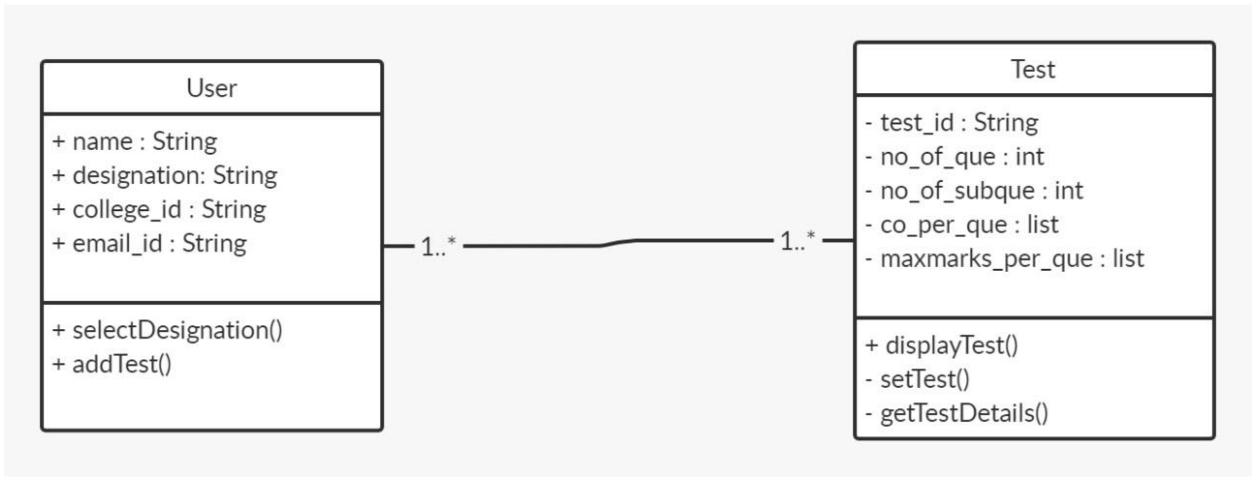


```
class Subject-Coordinator extends User
{
    void getdetails()
    {
        Scanner myObj = new Scanner(System.in);
        designation= "subject-coordinator";
        subject= myObj.nextLine();
        return;
    }
}
class HOD extends User
{
    void getdetails()
    {
        designation= "hod";
        return;
    }
}
```

Creator

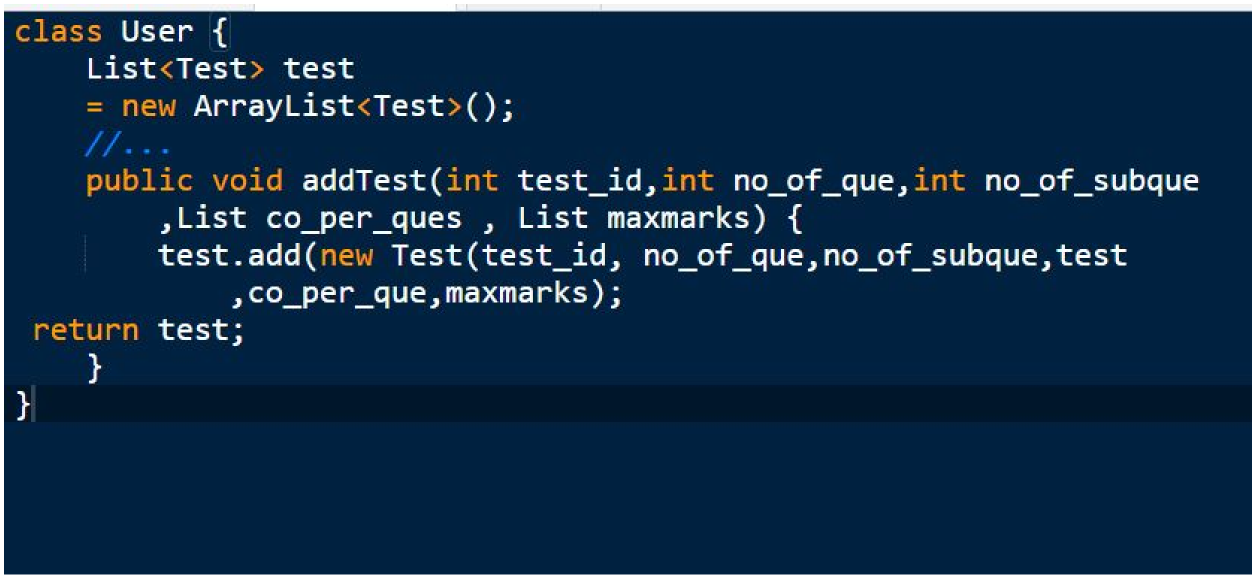
Problem : Who creates the object of the class test?

Solutions:Assign class user the responsibility to create object A if one of these is true



In this project:

- User class is decided to be the creator based on the object association and interaction with the test class. And thus the user is the creator of test.



```
class User {
    List<Test> test
    = new ArrayList<Test>();
    //...
    public void addTest(int test_id,int no_of_que,int no_of_subque
        ,List co_per_ques , List maxmarks) {
        test.add(new Test(test_id, no_of_que,no_of_subque,test
            ,co_per_que,maxmarks);
 return test;
    }
}
```

Conclusion : Thus in this assignment we have successfully Identified and implemented GRASP patterns.