Assignment no. 8

Title: Study and implement GRASP pattern.

Problem statement:

- Identification and implementation of GRASP pattern for blood bank management System.
- Apply any two GRASP patterns to refine the Design Model for a given problem descriptionusing 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:

- What are GRASP Patterns?

GRASP patterns basically describe fundamental principles of object design and the responsibility assignment. After identification of the requirements and creation of a domain model, we add methods to the software classes, and define the messaging between the objects to fulfil the requirement of the system. The GRASP patterns apply design reasoning in
a methodical, rational, explainable way.

- Responsibilities and Methods

A responsibility is defined "a contract or obligation of a classifier". It is basically the obligation posed on a classifier to do a certain thing. They are related to the obligations of an object in terms of its behavior.
The responsibilities are of the following two types:

1. Knowing
2. Doing

Doing is something in itself, such as creating an object or doing a calculation., Initiating action in other objects, controlling and coordinating activities in other objects, etc.

Whereas, knowing responsibilities of an object include: knowing about private encapsulated data, knowing about related objects, knowing about things it can derive or calculate something.

Responsibilities are assigned to classes of objects during object design. For example, we may declare that a vehicle class can add a new vehicle entry I.e. “create a new object of vehicle”, or say "an admin is responsible for knowledge of the log" (a knowing).

Controller

The Controller pattern assigns the responsibility of dealing with system events to a non-UI class that represent the overall system or a use case scenario. A Controller object is a non-user interface object responsible for receiving or handling a system event. A use case controller should be used to deal with all system events of a use case, and may be used for more than one use case (for instance, for use cases Create User and Delete User, one can have one UserController, instead of two separate use case controllers).

It is defined as the first object beyond the UI layer that receives and coordinates ("controls") a system operation. The controller should delegate to other objects the work that needs to be done; it coordinates or controls the activity. It should not do much work itself. The GRASP Controller can be thought of as being a part of the Application/Service layer (assuming that the application has made an explicit distinction between the App/Service layer and the Domain layer) in an object-oriented system with common layers.

Low Coupling GRASP Pattern

The software industry has the biggest issue: How to support low dependency, low change impact, and increase reuse?

The solution is to assign responsibilities so that coupling remains low. Try to avoid one class to have to know about many others.

Key points about low coupling

- Low dependencies between “artefacts” (classes, modules, components).
- There shouldn’t be too much dependency between the modules, even if there is a dependency it should be via the interfaces and should be minimal.
- Avoid tight-coupling for collaboration between two classes (if one class wants to call the logic of a second class, then the first class needs an object of second class it means the first class creates an object of the second class).
- Strive for loosely coupled design between objects that interact.

High Cohesion GRASP Pattern

High Cohesion is an evaluative pattern that attempts to keep objects appropriately focused, manageable and understandable. High cohesion is generally used in support of Low Coupling. To achieve this, assign responsibilities so that cohesion remains high. Try to avoid classes to do too much or too different things.

The term cohesion is used to indicate the degree to which a class has a single, well-focused responsibility. Cohesion is a measure of how the methods of a class or a module are meaningfully and strongly related and how focused they are in providing a well-defined purpose to the system.

A class is identified as a low cohesive class when it contains many unrelated functions within it. And that is what we need to avoid because big classes with unrelated functions hamper their maintenance. Always make your class small and with precise purpose and highly related functions.

Key points about high cohesion

- The code has to be very specific in its operations.
- The responsibilities/methods are highly related to class/module.
- The term cohesion is used to indicate the degree to which a class has a single, well-focused responsibility. The more focused a class is the higher its cohesiveness-a good thing.
- A class is identified as a low cohesive class when it contains many unrelated functions within it and that is a must to avoid because big classes with unrelated functions hamper their maintenance. Always make your class small and with precise purpose and highly related functions.

Polymorphism

- According to Polymorphism, responsibility for defining the variation of behaviours based on type is assigned to the types for which this variation happens. This is achieved using polymorphic operations.

Pure Fabrication

- A pure fabrication is a class that does not represent a concept in the problemdomain, 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 "Service" in Domain-driven design.

Indirection

- The Indirection pattern supports low coupling (and reuse potential) between two elements by assigning the responsibility of mediation between them to an intermediate object. An example of this is the introduction of a controller component for mediation between data (model) and its representation (view) in the Model-view-controller pattern.

Protected Variations

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

Controller GRASP Pattern

1. The controller pattern assigns the responsibility of dealing with system events to a non-UI class that represents the overall system or a use case scenario. A controller object is a non-user interface object responsible for receiving or handling a system event.
2. Who should be responsible for handling an input system event?
3. A use case controller should be used to deal with all system events of a use case, andmay be used for more than one use case. For instance, for the use cases. “add_slot” and “delete_slot”, one can have a single class called “manage_slots”, instead of two separate use case controllers.
4. The controller is defined as the first object beyond the UI layer that receives and coordinates ("controls") a system operation. The controller should delegate the work that needs to be done to other objects; it coordinates or controls the activity. It should not do much work itself.
5. The GRASP Controller can be thought of as being a part of the application/service layer

(assuming that the application has made an explicit distinction between the application/service layer and the domain layer) in an object-oriented system with common layers in an information system logical architecture.

Benefits

- Either front-end part or backend part could be changed without requiring the change into other, thus reducing the work required.
- The controller is a simple class that mediates between the UI and problem domain classes.
- Handles event requests.
- Specific for output requests.

Creator GRASP pattern

Creation of objects is one of the pillars of OOPS. But the question remains, which class is responsible for creating objects for a particular
class. For e.g. who should create a new vehicle entry if one does not exist?
Or who should be responsible to add a new user to the system?

In general, we assign class B the responsibility to create object A if one, or preferably more, of the following apply:

- Instances of B contain or compositely aggregate instances of A
- Instances of B record instances of A
- Instances of B closely use instances of A
- Instances of B have the initializing information for instances of A and pass it on creation.

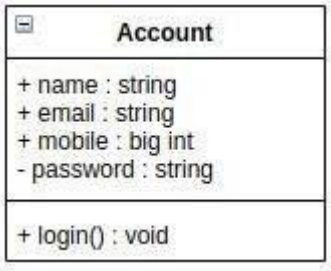
Benefits

Low coupling is supported, which implies lower maintenance dependencies and higher opportunities for reuse.

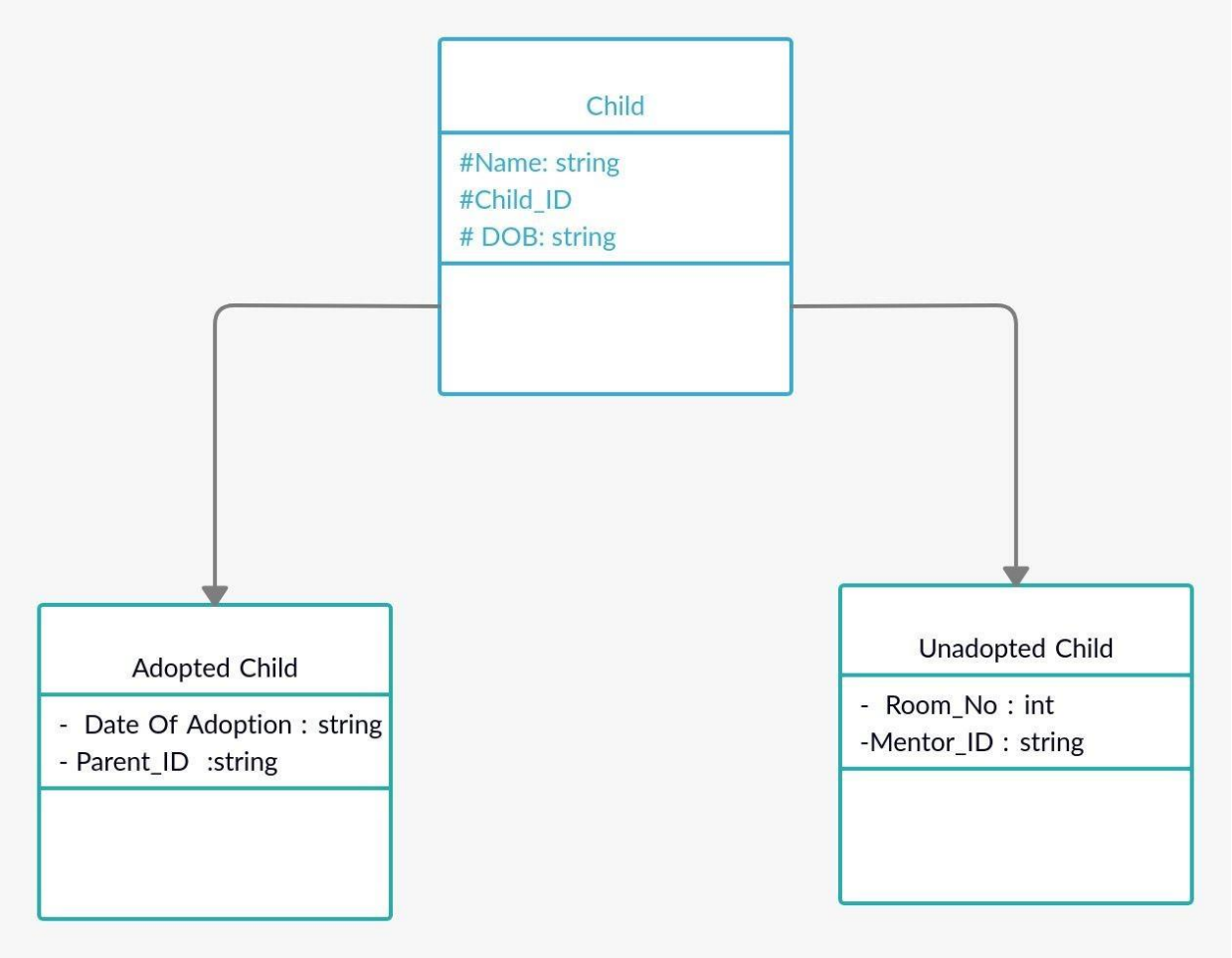
Diagrams:

GRASP Pattern: Low-coupling

Before implementation of GRASP, the login activity was different for mentor and admin. That is mentor and admin classes had separate methods for login.



However, after applying GRASP low coupling pattern, we created one new class, Account, which will store the generalized login credentials mentor and admin. And the login activity is also generalized in the account class for all three types of user. This helps to reduce coupling, which in turn helps increase the reusability as this class can be used in different parts of the system directly.

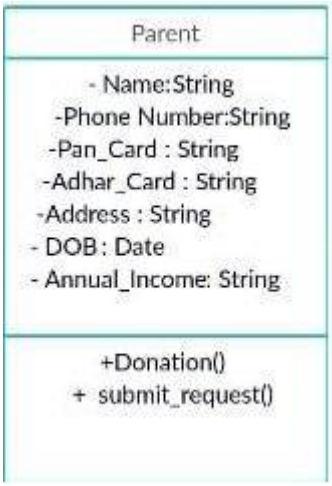


code-

```
import java.util.*;
import java.lang.*;
import java.io.*;


class Child {
    public int name;
    public int date_of_birth;
    public int child_id;
}

class Unadopted_child extends Child {
    public int room_num;
    public int mentor_id;
}

class Adopted_child extends Child {
    public int date_of_adoption;
    public int parent_id;
}
```

GRASP Creator:

Creator is a class responsible for creation of an instance. In our system, creation of a new adoption request is the responsibility given to the parent class, which basically creates a new instance of adoption request whenever parent request.



code-

```
public class Parent extend User {

    Parent(String name, String phone_num, String address,String dob, String adhr_card, String pan_card, String annual_income) {
        this.name = name;
        this.phone_num = phone_num;
        this.adhr_card = adhr_card;
        this.pan_card = pan_card;
        this.dob = dob;
        this.annual_income = annual_income
    }

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

    public void confirm_meeting() {
        // code for meetinh confirmation
    }

    public void apply_for_donation() {
        // code for application submision
    }

}
```

Conclusion:

Thus, in this assignment, we learnt about GRASP design patterns, and the different assignment of responsibilities, and the advantages of the same. Also, we implemented the GRASP design pattern in our problem statement.