**Assignment 9**

**Identification and Implementation of GOF Pattern**

**AIM -** Apply any four 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.

**PROBLEM STATEMENT -**

1. Identification and Implementation of GOF pattern
2. Apply any two GOF patterns 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 -**

1. To Study GOF patterns.
2. To identify the applicability of GOF in the system.
3. Implement a system with GOF pattern.

**THEORY -**

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.

1. **Creational Design Patterns**
2. Abstract​ ​ Factory​ - Allows the creation of objects without specifying their concrete type.
3. Builder​ - Uses to create complex objects.
4. Factory Method​ - ​ Creates objects without specifying the exact class to create.
5. Prototype​ - ​ Creates a new object from an existing object.
6. Singleton​ - ​ Ensures only one instance of an object is created.

**2) Structural Design Patterns**

1. Adapter​ - Allows for two incompatible classes to work together by wrapping an interface around one of the existing classes.
2. Bridge​ - Decouples an abstraction so two classes can vary independently.
3. Composite​ - Takes a group of objects into a single object.
4. Decorator​ - Allows for an object’s behaviour to be extended dynamically at run time.
5. Facade​ - Provides a simple interface to a more complex underlying object.
6. Flyweight​ - Reduces the cost of complex object models.
7. Proxy​ - Provides a placeholder interface to an underlying object to control access, reduce cost, or reduce complexity.

**3) Behavior Design Patterns**

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

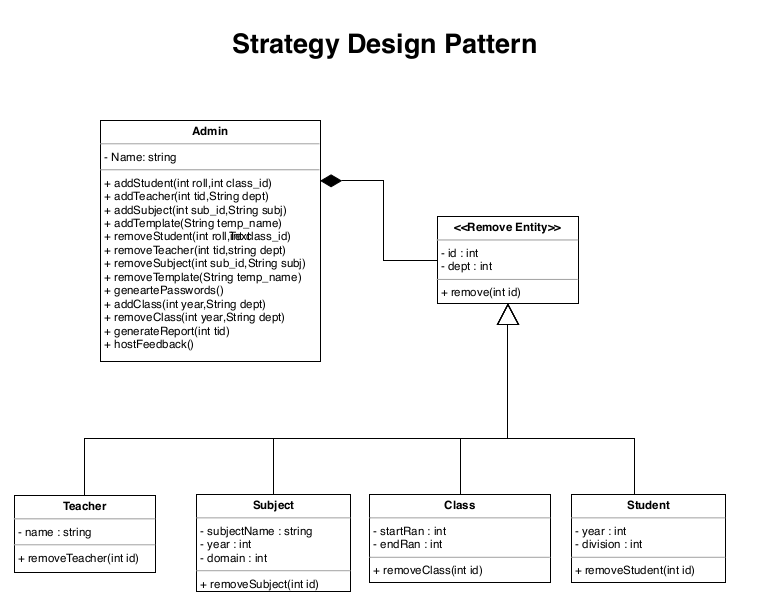
**4) Strategy Design Patterns**

Defines a family of algorithms,

1. Encapsulates each algorithm, and makes the algorithms interchangeable within that family.
2. Strategy lets the algorithm vary independently from clients that use it.

**Example :**

For instance, a class that performs validation on incoming data may use a strategy pattern to select a validation algorithm based on the type of data, the source of the data, user choice, or other discriminating factors. These factors are not known for each case until run-time, and may require radically different validation to be performed. The validation strategies, encapsulated separately from the validating object, may be used by other validating objects in different areas of the system (or even different systems) without code duplication.

****

In this project, we can use the strategy pattern to formulate multiple strategies for all entities. Admin class needs to delete the entities and so we need to implement several methods to delete the respective entities. We can use a single strategy interface to form a common strategy. This interface will be inherited by the subsequent four subclasses. Each of these classes will override the parent method. Then we can only create the object of a single subclass. Thus by calling appropriate subclasses we can make use of a strategy pattern to make our system more modular with respect to software engineering.