Assignment No. 8

Title:

Study and implement GRASP patterns.

Problem statement:

- Identification and implementation of GRASP pattern.
- Apply any two GRASP 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:

- 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

- Responsibility can be:
    - accomplished by a single object.
    - or a group of objects collaboratively accomplish a responsibility.
- GRASP helps us in deciding which responsibility should be assigned to which object/class.
- Identify the objects and responsibilities from the problem domain, and also identify how objects interact with each other.
- Define blueprint for those objects – i.e. class with methods implementing those responsibilities.

Responsibilities are assigned to classes of objects during object design. For example, we may declare that a Manager can add a new Volunteer I.e. “create a new object of volunteers”.

Creator

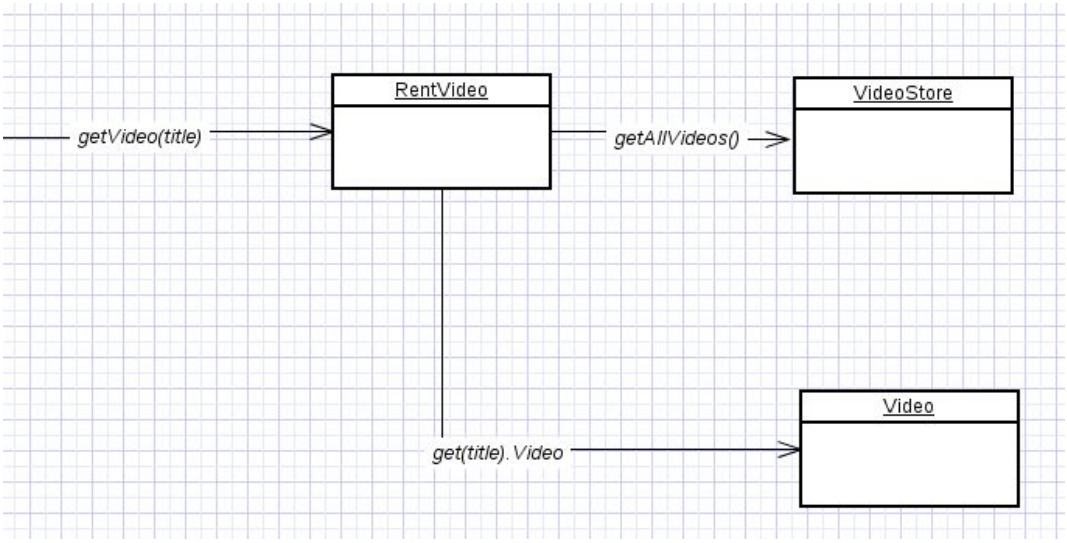
- “Container” object creates “contained” objects.
- Decide who can be creator based on the objects association and their interaction.
- Consider VideoStore and Video in that store.
- VideoStore has an aggregation association with Video. I.e, VideoStore is the container and the Video is the contained object.
- So, we can instantiate video objects in VideoStore class.
- Example,

Expert

- Expert principle says – assign those responsibilities to an object for which an object has the information to fulfill that responsibility.
- They have all the information needed to perform operations, or in some cases they collaborate with others to fulfill their responsibilities.
- Assume we need to get all the videos of a VideoStore.
- Since VideoStore knows about all the videos, we can assign this responsibility of giving all the videos to VideoStore class.
- VideoStore is the information expert.
- Example,

Low Coupling

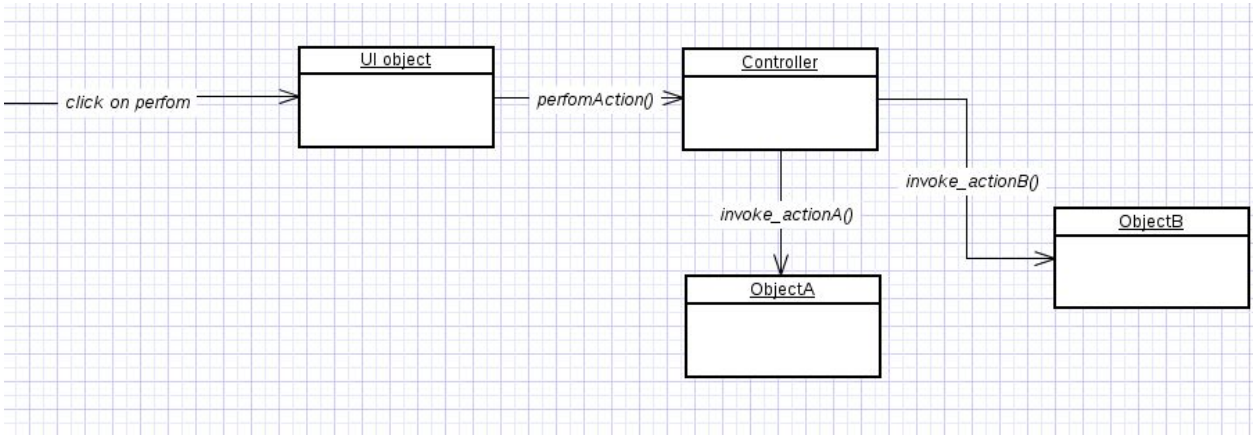
- Coupling – object depending on other object.
- When dependent upon element changes, it affects the dependent also.
- Low Coupling – How can we reduce the impact of change in dependent upon elements on dependent elements.
- Prefer low coupling – assign responsibilities so that coupling remains low.
- Minimizes the dependency hence making system maintainable, efficient and code reusable
- Two elements are coupled, if
    - One element has aggregation/composition association with another element.
    - One element implements/extends another element.
- Example,



here class Rent knows about both VideoStore and Video objects. Rent is depending on both the classes.

Controller

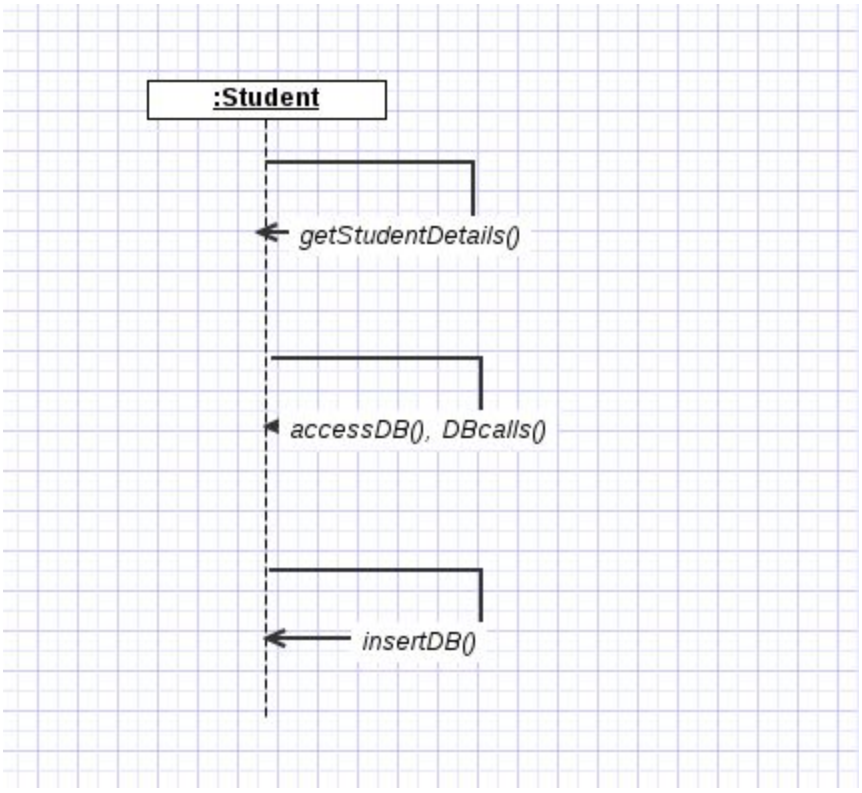
- Deals with how to delegate the request from the UI layer objects to domain layer objects.
- When a request comes from a UI layer object, the Controller pattern helps us in determining what is that first object that receives the message from the UI layer objects.
- This object is called a controller 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.
- We can make an object as Controller, if
    - Object represents the overall system (facade controller)
    - Objects represent a use case, handling a sequence of operations (session controller).
- Benefits
    - Can reuse this controller class.
    - Can use to maintain the state of the use case.
    - Can control the sequence of the activities
- Example,



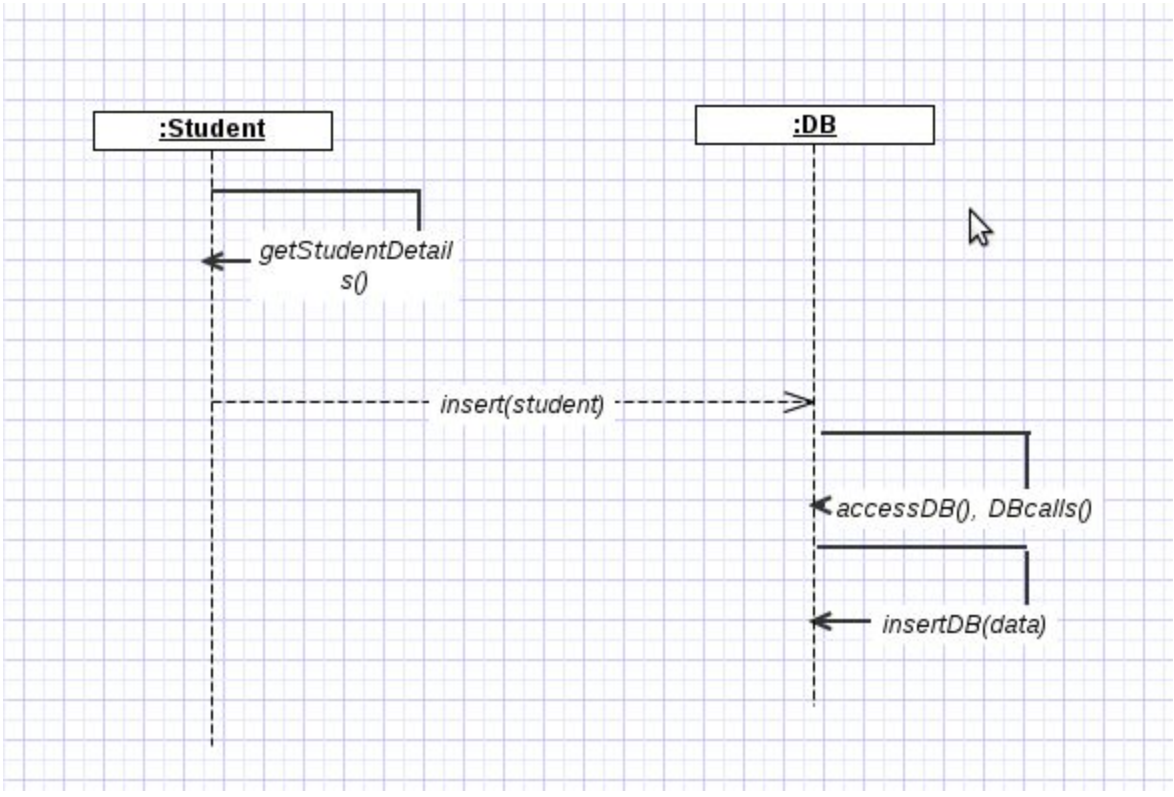
Low & High Cohesion

- Related responsibilities into one manageable unit.
- Prefer high cohesion
- Clearly defines the purpose of the element
- Benefits
    - Easily understandable and maintainable.
    - Code reuse
    - Low coupling

- Example,
    - Low Cohesion

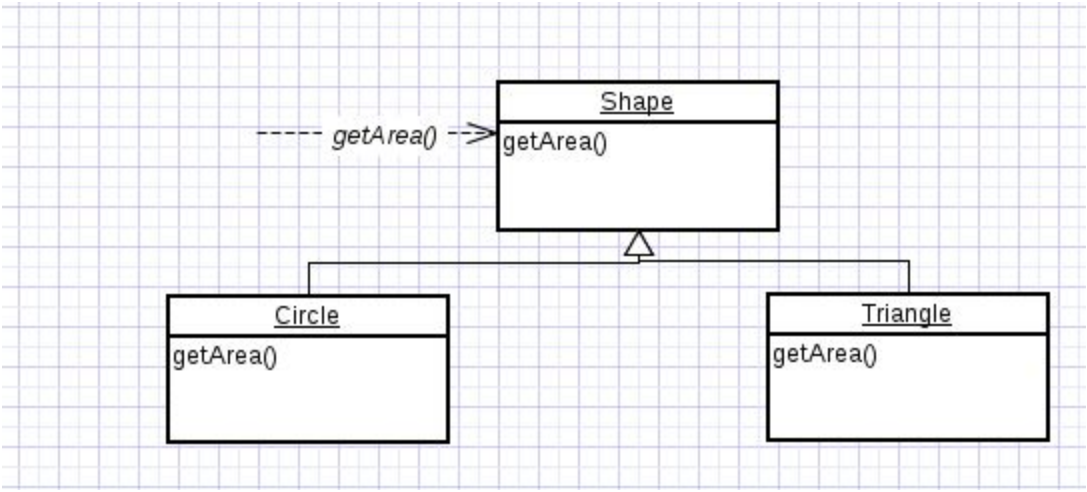


    - High Cohesion



Polymorphism

- Polymorphism guides us in deciding which object is responsible for handling those varying elements.
- Benefits: handling new variations will become easy.
- the getArea() varies by the type of shape, so we assign that responsibility to the subclasses.



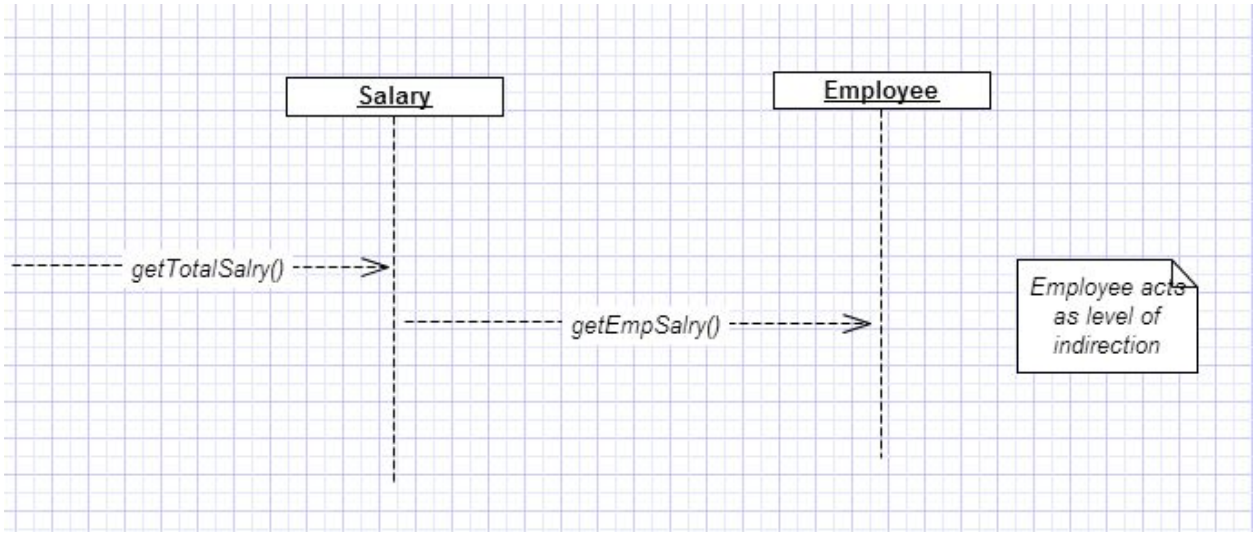
- By sending a message to the Shape object, a call will be made to the corresponding sub class object – Circle or Triangle.

Pure Fabrication

- Fabricated class/ artificial class – assign set of related responsibilities that doesn't represent
- any domain object.
- Provides a highly cohesive set of activities.
- Behavioral decomposed – implements some algorithm.
- Examples: Adapter, Strategy
- Benefits: High cohesion, low coupling and can reuse this class.
- Suppose we Shape class, if we must store the shape data in a database.
- If we put this responsibility in Shape class, there will be many database related operations thus making Shape incohesive.
- So, create a fabricated class DBStore which is responsible to perform all database operations.
- Similarly logInterface which is responsible for logging information is also a good example for Pure Fabrication.

Indirection

- Indirection introduces an intermediate unit to communicate between the other units, so that the other units are not directly coupled.
- Benefits: low coupling
- Example: Adapter, Facade, Observer
- Here polymorphism illustrates indirection
- Class Employee provides a level of indirection to other units of the system.
- Example,



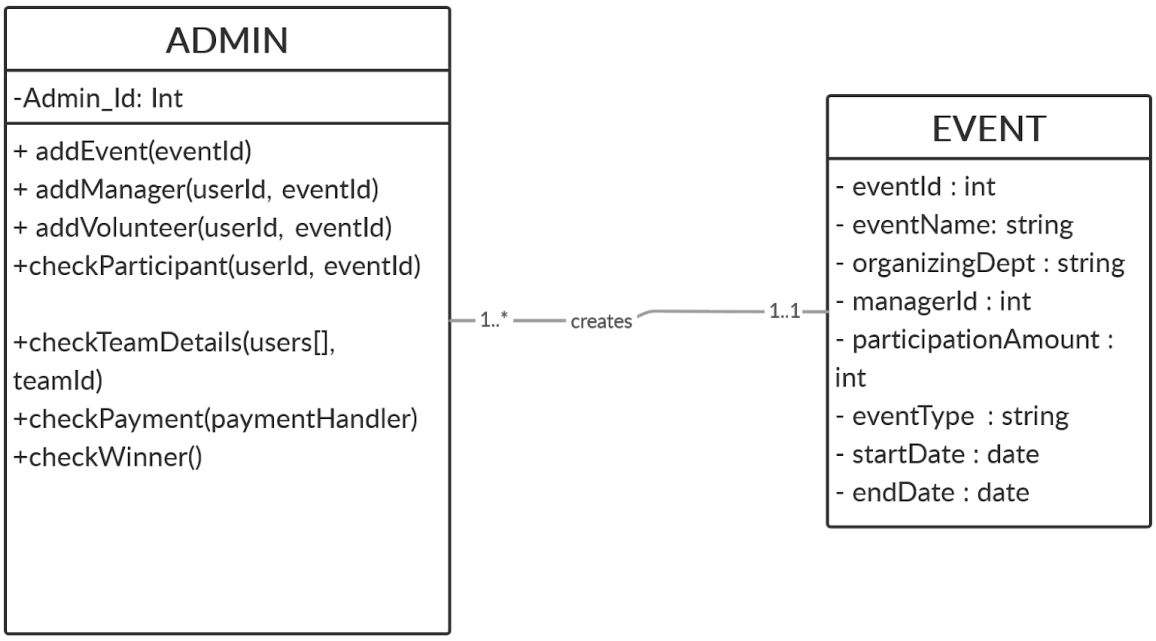
Protected Variation

- It provides a well defined interface so that there will be no effect on other units.
- Provides flexibility and protection from variations.
- Provides more structured design.
- Example: polymorphism, data encapsulation, interfaces

Diagrams :

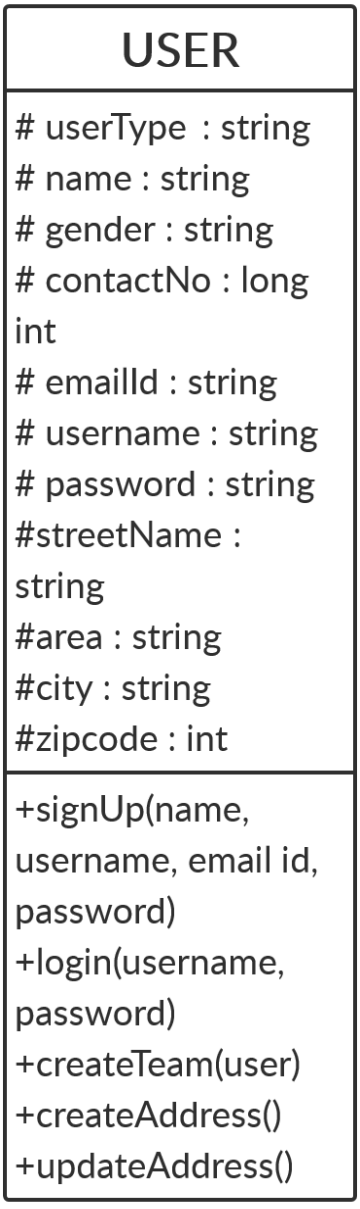
GRASP Creator:

Creator is a class responsible for creation of an instance. In our system, creation of a new team is the responsibility given to the User class, which basically creates a new instance of event whenever required.

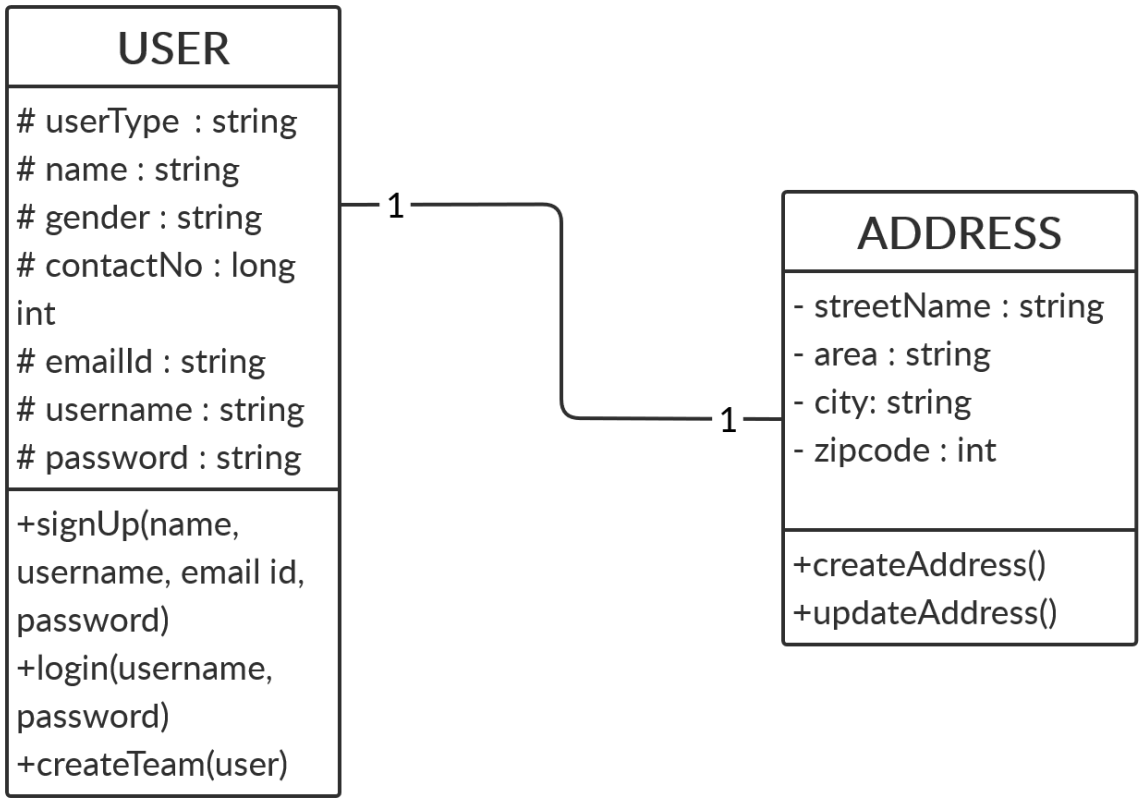


GRASP Pattern : Low Coupling

Before implementation of GRASP, the responsibility of adding and updating the address was the responsibility given to the User class itself.



However, after applying GRASP low coupling pattern, we created one new class, Address, which will store the generic address for a specific location, and the responsibilities of creation and updating the address has been delegated to that class.
This helps reduce coupling, which in turn helps increase the reusability as this class can be used in different parts of the system directly.



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.