Assignment no 8

Aim:
Identify and implement grasp patterns.

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 Patterns

What are GRASP Patterns?

They describe fundamental principles of object design and responsibility assignment, expressed as patterns.

After identifying your requirements and creating a domain model, then add methods to the software classes, and define the messaging between the objects to fulfill the requirements.

The GRASP patterns are a learning aid to help one understand essential object design, and apply design reasoning in a methodical, rational, explainable way. This approach to understanding and using design principles are based on patterns of assigning responsibilities.

Responsibilities and Methods

The UML defines a responsibility as "a contract or obligation of a classifier". Responsibilities are related to the obligations of an object in terms of its behaviour. Basically, these responsibilities are of the following two types:

- knowing
- doing
- doing something itself, such as creating an object or doing a calculation
- initiating action in other objects
- controlling and coordinating activities in other objects

Knowing responsibilities of an object include:

- knowing about private encapsulated data
- knowing about related objects
- knowing about things it can derive or calculate

Responsibilities are assigned to classes of objects during object design. For example, I may declare that "a Sale is responsible for creating SalesLineItems" (a doing), or "a Sale is responsible for knowing its total" (a knowing). Relevant responsibilities related to "knowing" are often inferable from the domain model because of the attributes and associations it illustrates.

Low Coupling GRASP Pattern

Key points about low coupling:

- Low dependencies between “artifacts” (classes, modules, components).
- There shouldn’t be too much of 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 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.
- Inversion Of Control (IoC) / Dependency Injection (DI) - With DI objects are given their dependencies at creation time by some third party (i.e. Java EE CDI, Spring DI…) that coordinates each object in the system. Objects aren’t expected to

create or obtain their dependencies—dependencies are injected into the objects that need them. The key benefit of DI—loose coupling.

High Cohesion GRASP Pattern

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. 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. 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 what we need to avoid because big classes with unrelated functions hamper their maintaining. Always make your class small and with precise purpose and highly related functions.

Controller GRASP Pattern

Problem
Who should be responsible for handling an input system event?
An input system event is an event generated by an external actor. They are associated with system operations of the system in response to system events,
just as messages and methods are related. For example, when a writer using a word processor presses the "spell check" button, he is generating a system event indicating "perform a spell check."
A Controller is a non-user interface object responsible for receiving or handling a system event. A Controller defines the method for the system operation.
Who should be responsible for handling an input event, which object/s beyond the UI layer receives interaction?

Solution
Assign the responsibility for receiving or handling a system event message to a class representing one of the following choices:

• Represents the overall system, device, or subsystem (facade controller).
• Represents a use case scenario within which the system event occurs, often named <UseCaseName>Handler, <UseCaseName>Coordinator, or <Use-CaseName>Session (use-case or session controller).
• Use the same controller class for all system events in the same use case scenario.
• Informally, a session is an instance of a conversation with an actor. Sessions can be of any length but are often organized in terms of use cases (use case sessions).
Corollary: Note that "window," "applet," "widget," "view," and "document" classes are not on this list. Such classes should not fulfill the tasks associated with system events, they typically receive these events and delegate them to a controller.

Benefits
• Either the UI classes or the problem/software domain classes can change without affecting the other side.
• The controller is a simple class that mediates between the UI and problem domain classes, just forwards
• event handling requests
• output requests

Creator GRASP Pattern

Problem
Who should be responsible for creating a new instance of some class? The creation of objects is one of the most common activities in an object-oriented system. Consequently, it is useful to have a general principle for the assignment of creation responsibilities. Assigned well, the design can support low coupling, increased clarity, encapsulation, and reusability. Solution
Assign class B the responsibility to create an instance of class A if one or more of the following is true:
• B aggregates A objects.
• B contains A objects.
• B records instances of A objects.
• B closely uses A objects.
• B has the initializing data that will be passed to A when it is created (thus B is an Expert with respect to creating A).
B is a creator of A objects.

If more than one option applies, prefer a class B which aggregates or contains class A.

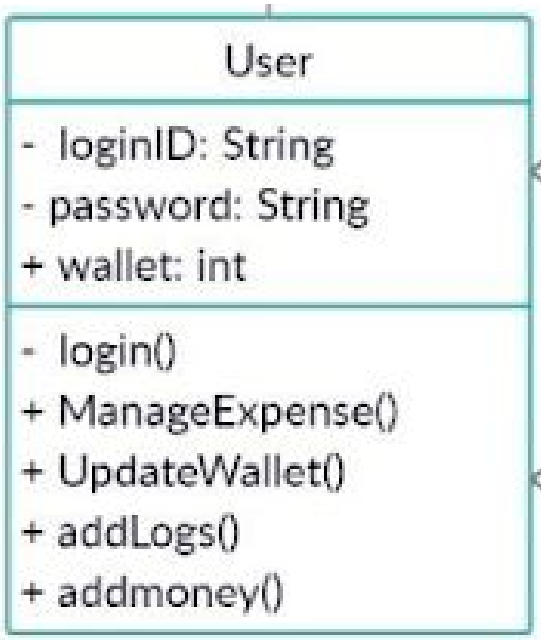
Benefits

• Low coupling (described next) is supported, which implies lower maintenance dependencies and higher opportunities for reuse. Coupling is probably not increased because the created class is likely already visible to the creator class, due to the existing associations that motivated its choice as creator.

Grasp Diagram for MoneyMize:

CREATOR PATTERN :-

Here, GroupEvent and PersonalEvent object is created by the owner(User object) so we assign Creator to user.

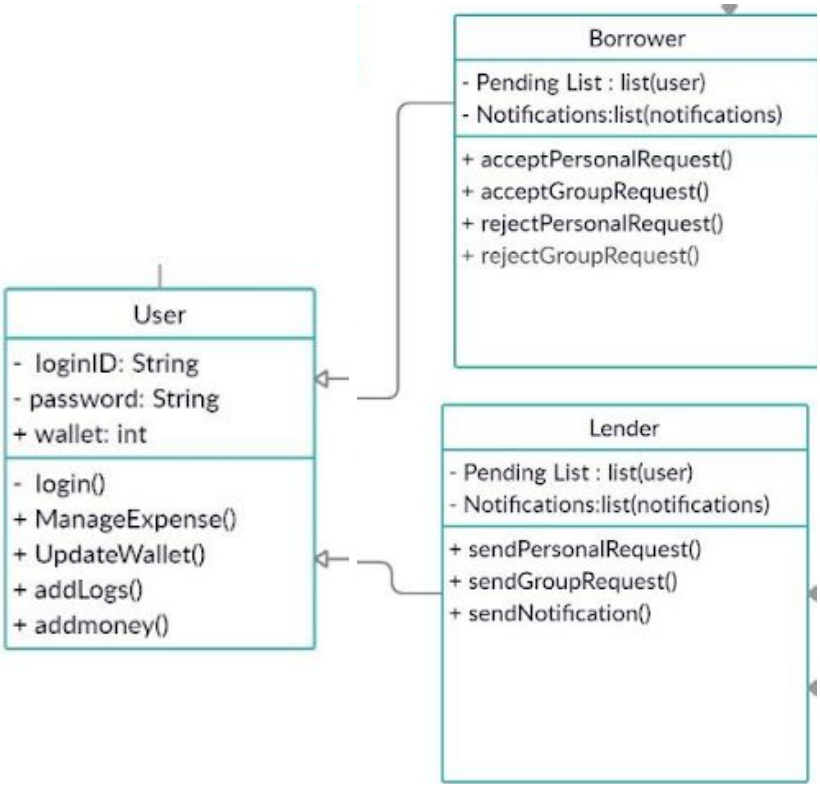


LOW COUPLING :-

Group Event class contains event owner and participants as composite objects. Hence, adhere to LOW COUPLING.

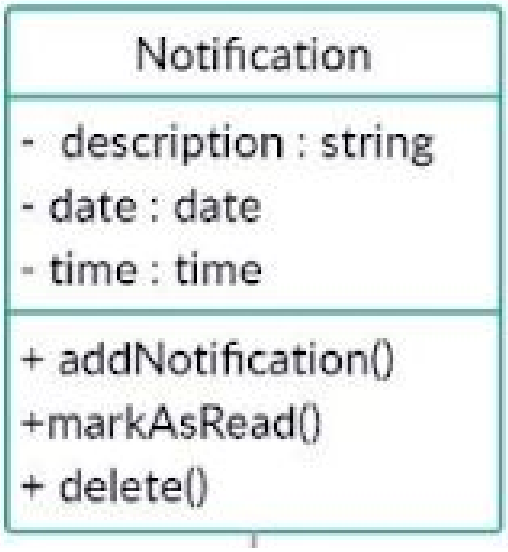
HIGH COHESION :-

Send Request, Accept Request, Reject Request are separated from user class to achieve HIGH COHESION..



INFORMATION EXPERT:-

Notification class has all the information about all the events / transactions happening within the system. Therefore we assign Information Expert to Notification.



Conclusion:

We successfully Identified and implemented grasp patterns.