Assignment 8
Identify and Implement GRASP pattern

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
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 behaviour.
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 casesCreate 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 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 "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 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.

Controller GRASP Pattern

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.
Who should be responsible for handling an input 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 the use cases. “add_slot” and “delete_slot”, one can have a single class called “manage_slots”, instead of two separate use case controllers.

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. 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.

<u>GRASP PATTERNS:</u>

A. Controller

This principle implementation depends on the high-level design of our system but generally, we need to always define objects which orchestrate our business transaction processing. At first glance, it would seem that the MVC Controller in

Web applications/API's is a great example here (even the name is the same) butfor me, it is not true. Of course, it receives input but it shouldn't coordinate a system operation – it should delegate it to separate service or Command Handler.

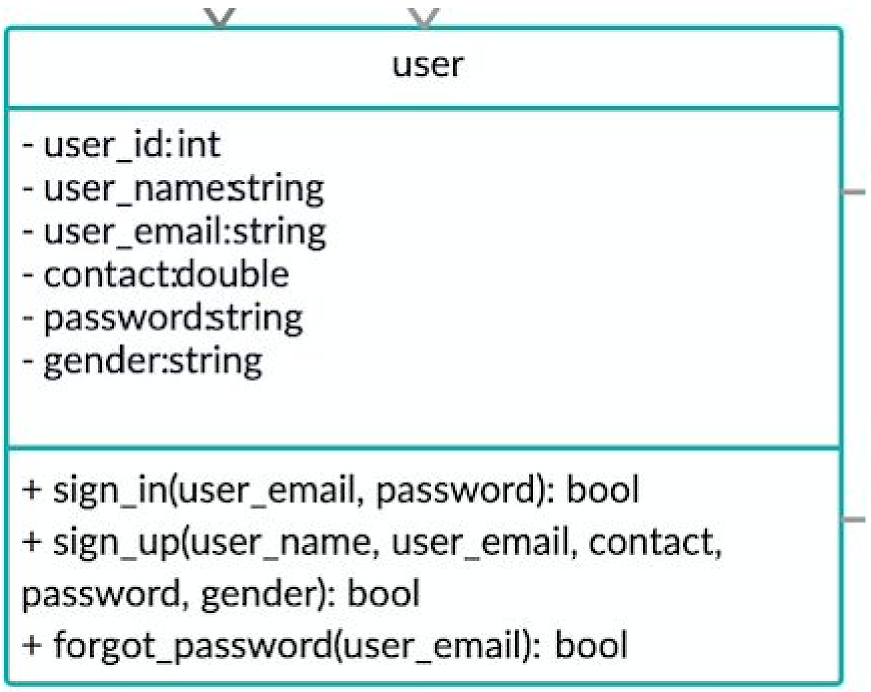
user
- user_id:int
- user_name:string
- user_email:string
- contact:double
- password:string
- gender:string
+ sign_in(user_email, password): bool
+ sign_up(user_name, user_email, contact, password, gender): bool
+ forgot_password(user_email): bool

Fig1. Creator

In this project -

- The user class is the controller.
- It is used to create and handle most of the other objects.

B. Creator

The creation of objects is one of the most common activities in an object-oriented system. Which class is responsible for creating objects is a fundamental property of the relationship between objects of particular classes.

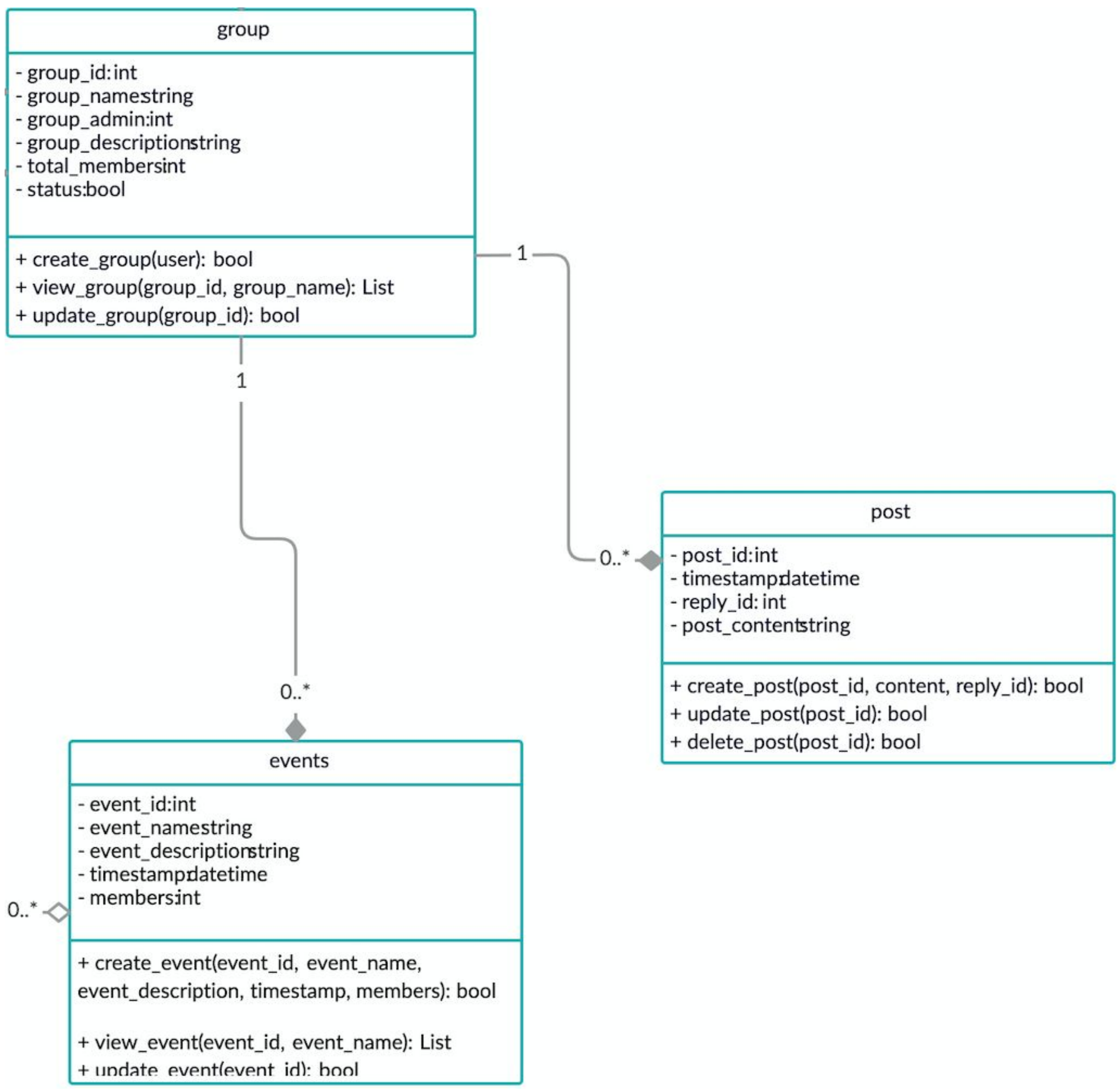
group
- group_id:int
- group_namestring
- group_admin:int
- group_descriptionstring
- total_membersint
- status:bool
+ create_group(user): bool
+ view_group(group_id, group_name): List
+ update_group(group_id): bool
1
1
0..*
0..*
post
- post_id:int
- timestampdatetime
- reply_id: int
- post_contentstring
+ create_post(post_id, content, reply_id): bool
+ update_post(post_id): bool
+ delete_post(post_id): bool
events
- event_id:int
- event_namestring
- event_descriptionstring
- timestampdatetime
- membersint
+ create_event(event_id, event_name, event_description, timestamp, members): bool
+ view_event(event_id, event_name): List
+ update_event(event_id): bool
0..*

Fig2. Creator

In this project -

- There is a composition relationship between group and post, and group and events.
- As group class consists of post and event class, it also in a way creates those classes.
- Event and Post are composed in Group class. So here Group class creates them.

C. 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. Assign a cohesive set of responsibilities to a non-domain entity in order to support high cohesion and low coupling.

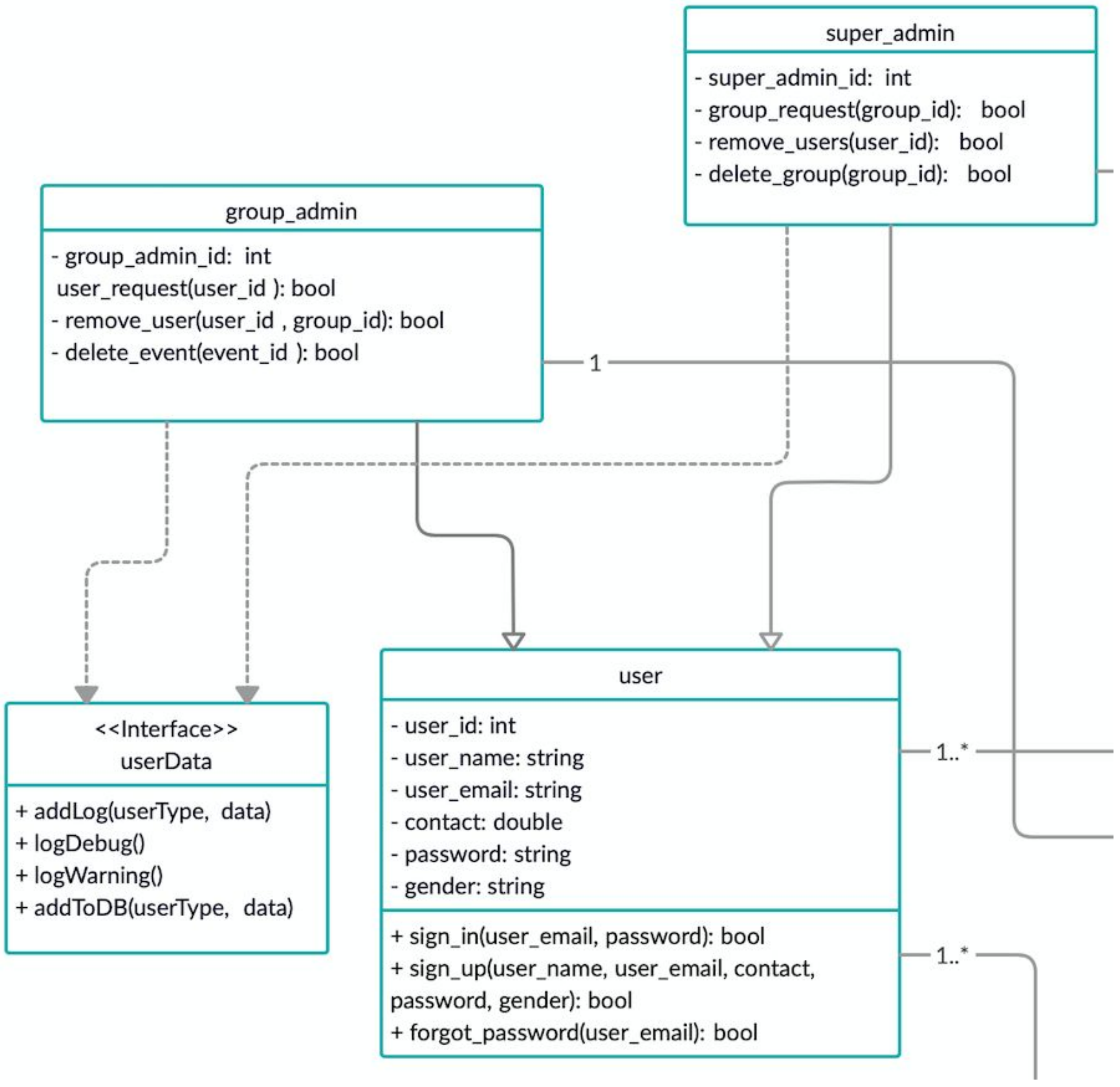
super_admin
- super_admin_id: int
- group_request(group_id): bool
- remove_users(user_id): bool
- delete_group(group_id): bool
group_admin
- group_admin_id: int
user_request(user_id): bool
- remove_user(user_id , group_id): bool
- delete_event(event_id): bool
1
<<Interface>>
userData
+ addLog(userType, data)
+ logDebug()
+ logWarning()
+ addToDB(userType, data)
user
- user_id: int
- user_name: string
- user_email: string
- contact: double
- password: string
- gender: string
+ sign_in(user_email, password): bool
+ sign_up(user_name, user_email, contact, password, gender): bool
+ forgot_password(user_email): bool
1..*
1..*

Fig3. Pure Fabrication

In this project -

- Group Admin and Super Admin have to store their logs and data in the database.
- If we put this responsibility in their respective classes, there will be many database and log related operations; thus making group admin, super admin and user incohesive.
- So, we created a “UserData” interface which is responsible to perform all database operations.
- Creating a new interface to perform database related operations benefits high cohesion, low coupling and reusability.

D. Polymorphism

The Polymorphism GRASP pattern deals with how a general responsibility gets distributed to a set of classes or interfaces. For example, the responsibility of calculating grades depends on the type of marking scheme for a particular student.

If polymorphism is not used, and instead the code tests the type of the object, then that section of code will grow as more types are added to the system. This section of code becomes more coupled (i.e., it knows about more types) and less cohesive (i.e., it is doing too much).

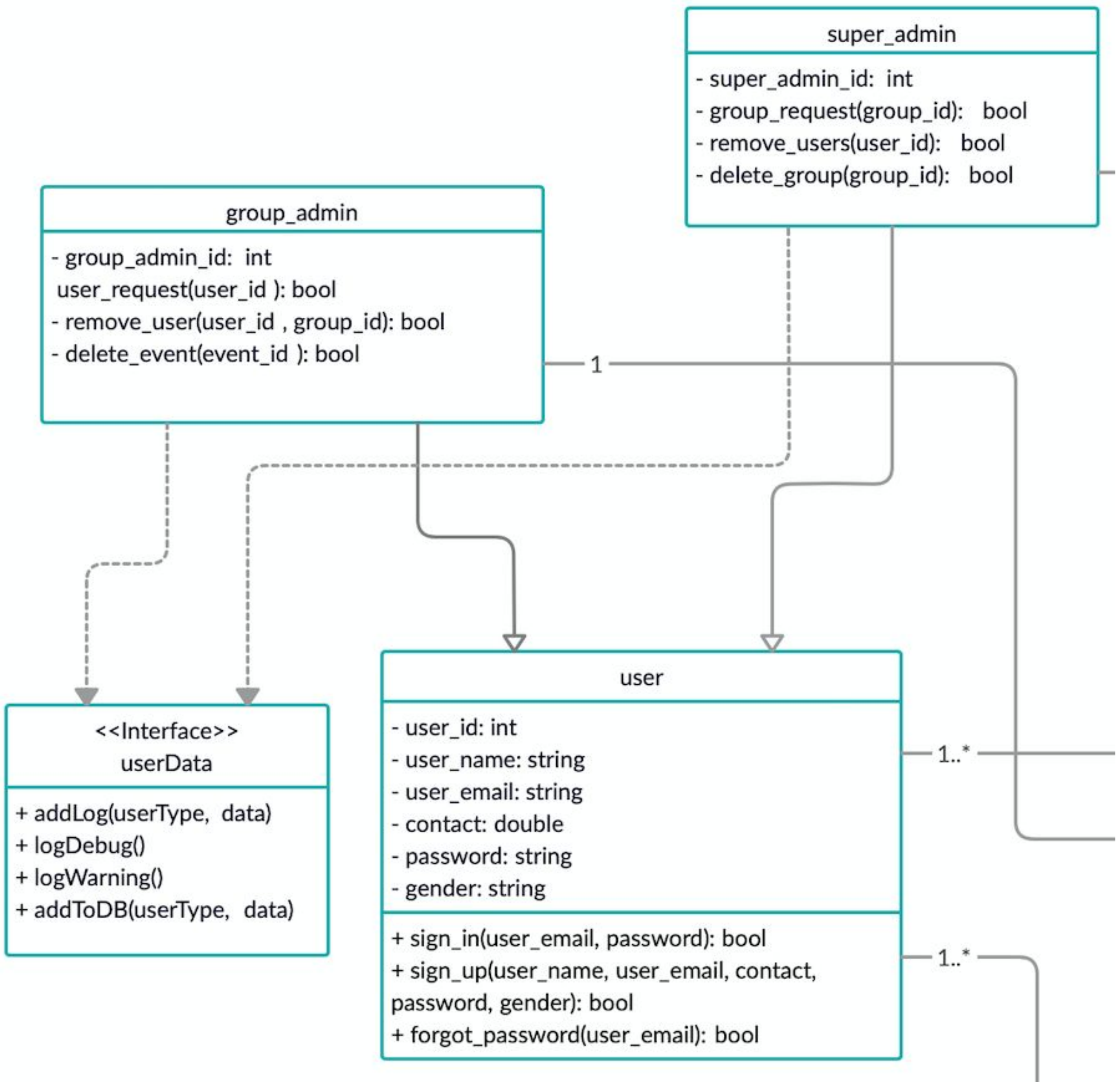
super_admin
- super_admin_id: int
- group_request(group_id): bool
- remove_users(user_id): bool
- delete_group(group_id): bool
group_admin
- group_admin_id: int
user_request(user_id): bool
- remove_user(user_id , group_id): bool
- delete_event(event_id): bool
1
<<Interface>>
userData
+ addLog(userType, data)
+ logDebug()
+ logWarning()
+ addToDB(userType, data)
user
- user_id: int
- user_name: string
- user_email: string
- contact: double
- password: string
- gender: string
+ sign_in(user_email, password): bool
+ sign_up(user_name, user_email, contact, password, gender): bool
+ forgot_password(user_email): bool
1..*
1..*

Fig4. Polymorphism

In this project -

- User, Super admin and Group admin are the classes which use polymorphism.
- The different methods vary according to the type of the user.

CONCLUSION:

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