Assignment 5

Title:
Design and implementation design model from analysis model

Problem Statement:
- Prepare a Design Model from Analysis Model
- Study in detail working on systems/projects.
- Identify Design classes/ Evolve Analysis Model. Use advanced relationships. Draw Design class Model using OCL and UML2.0 Notations. Implement the design model with a suitable object-oriented language.

Objective:
- To Identify Design level Classes.
- To Draw Design level class Model using analysis model.
- To Implement Design Model-class diagram

Relevant Theory:
Creating Design level Class Diagrams
Class diagrams model the static structure of a package or of a complete system. As the blueprints of the system, class diagrams model the objects that make up the system, allowing to display the relationships among those objects and to describe what the objects can do and the services they can provide.

Class diagrams
In UML, class diagrams are one of six types of structural diagram. Class diagrams are fundamental to the object modeling process and model the static structure of a system. Depending on the complexity of a system, you can use a single class diagram to model an entire system, or you can use several class diagrams to model the components of a system.

Class diagrams are the blueprints of your system or subsystem. You can use class diagrams to model the objects that make up the system, to display the relationships between the objects, and to describe what those objects do and the services that they provide.

Class diagrams are useful in many stages of system design. In the analysis stage, a class diagram can help you to understand the requirements of your problem domain and to identify its components. In an object-oriented software project, the class diagrams that you create during the early stages of the project contain classes that often translate into actual software classes and objects when you write code. Later, you can refine your earlier analysis and conceptual models into class diagrams that show the specific parts of your system, user interfaces, logical implementations, and so on. Your class diagrams then become a snapshot that describes exactly how your system works, the relationships between system components at many levels, and how you plan to implement those components.

During the analysis and design phases of the development cycle, you can create class diagrams to perform the following functions:

- Capture and define the structure of classes and other classifiers
- Define relationships between classes and classifiers
- Illustrate the structure of a model by using attributes, operations, and signals
- Show the common classifier roles and responsibilities that define the behavior of the system
- Show the implementation classes in a package
- Show the structure and behavior of one or more classes
- Show an inheritance hierarchy among classes and classifiers
- Show the workers and entities as business object models

During the implementation phase of a software development cycle, you can use class diagrams to convert your models into code

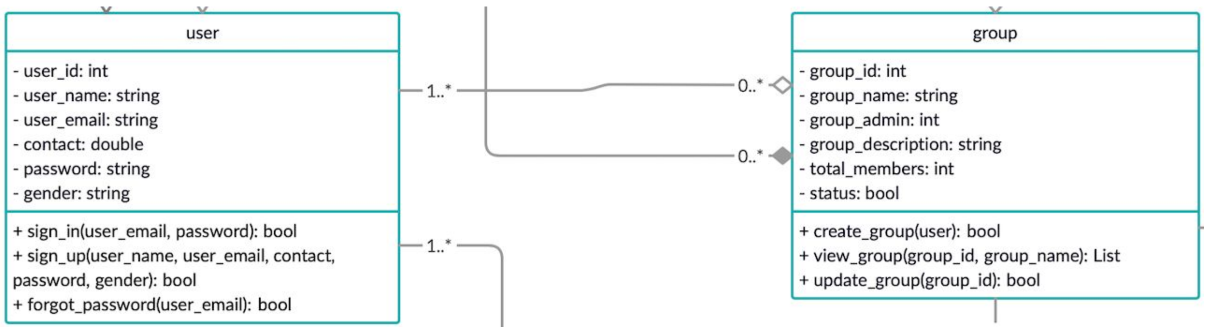
Aggregation relationships

In UML models, an aggregation relationship shows a class as a part of or subordinate to another class.

An aggregation is a special type of association in which objects are assembled or configured together to create a more complex object.

Data flows from the whole classifier, or aggregate, to the part. A part classifier can belong to more than one aggregate classifier and it can exist independently of the aggregate.

For example, a User class can have an aggregation relationship with a Group class, which indicates that the user is part of the group. Aggregations are closely related to compositions.
You can name an association to describe the nature of the relationship between two classifiers; however, names are unnecessary if you use association end names.
As the following figure illustrates, an aggregation association appears as a solid line with an unfilled diamond at the association end, which is connected to the classifier that represents the aggregate. Aggregation relationships do not have to be unidirectional.


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..*
0..*
0..*
group
- group_id: int
- group_name: string
- group_admin: int
- group_description: string
- total_members: int
- status: bool
+ create_group(user): bool
+ view_group(group_id, group_name): List
+ update_group(group_id): bool

Association relationships

In UML models, an association is a relationship between two classifiers, such as classes, that describes the reasons for the relationship and the rules that govern the relationship.
An association represents a structural relationship that connects two classifiers.
Like attributes, associations record the properties of classifiers.
For example, in relationships between classes, you can use associations to show the design decisions that you made about classes in your application that contain data, and to show which of those classes need to share data. You can use an association's navigability feature to show how an object of one class gains access to an object of another class or, in a reflexive association, to an object of the same class.
The name of an association describes the nature of the relationship between two classifiers and should be a verb or phrase.
In the diagram editor, an association appears as a solid line between two classifiers.

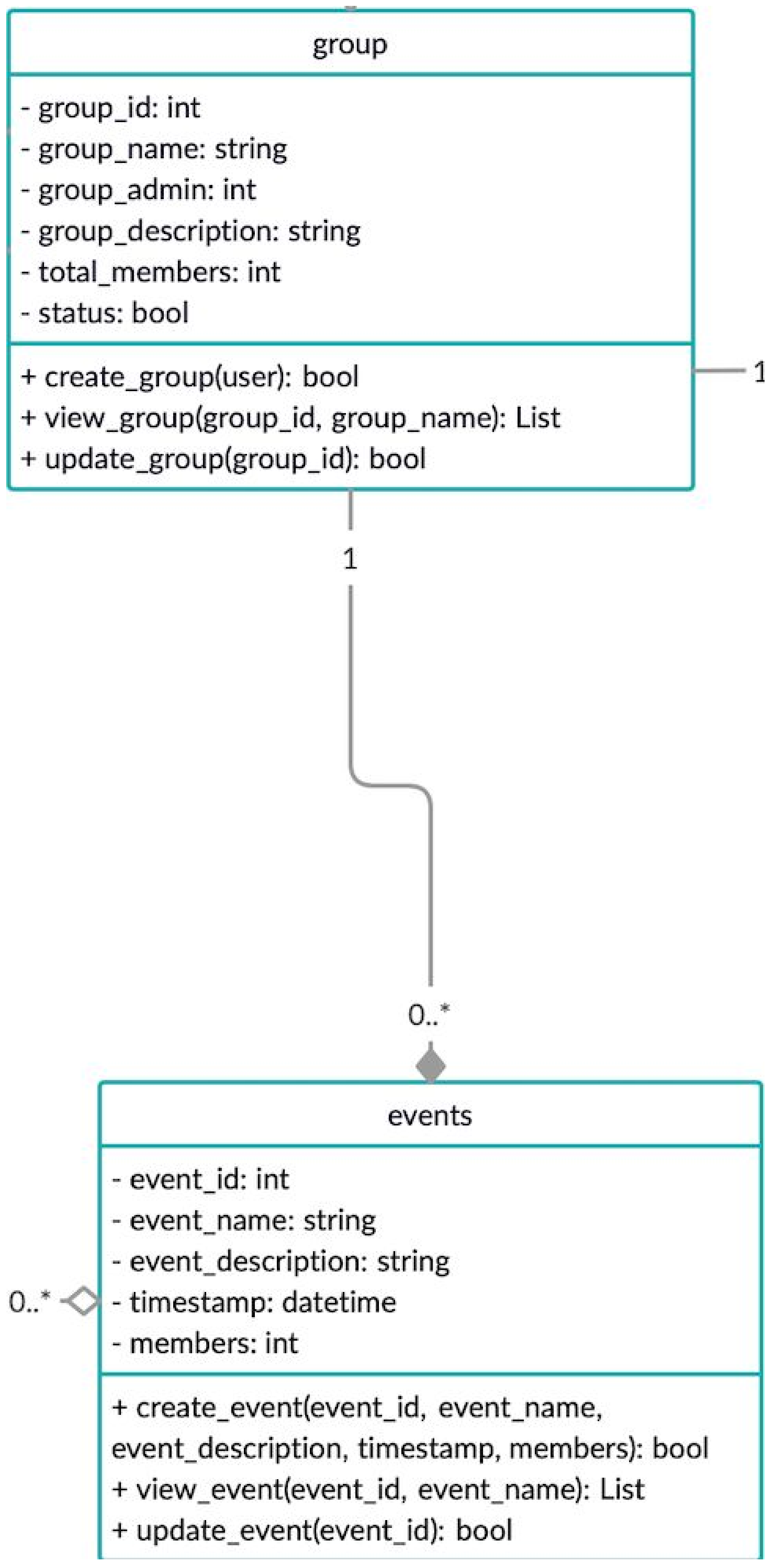
Association classes
In UML diagrams, an association class is a class that is part of an association relationship between two other classes.
You can attach an association class to an association relationship to provide additional information about the relationship. An association class is identical to other classes and can contain operations, attributes, as well as other associations.

Composition Association Relationships
A composition association relationship represents a whole–part relationship and is a form of aggregation. A composition association relationship specifies that the lifetime of the part classifier is dependent on the lifetime of the whole classifier.

In a composition association relationship, data usually flows in only one direction (that is, from the whole classifier to the part classifier).

For example, a composition association relationship connects a Group class with an Event class, which means that if you remove the group, the event is also removed.

As the following figure illustrates, a composition association relationship appears as a solid line with a filled diamond at the association end, which is connected to the whole, or composite, classifier.


group
- group_id: int
- group_name: string
- group_admin: int
- group_description: string
- total_members: int
- status: bool
+ create_group(user): bool
+ view_group(group_id, group_name): List
+ update_group(group_id): bool
1
1
0..*
events
- event_id: int
- event_name: string
- event_description: string
- timestamp: datetime
- members: int
+ create_event(event_id, event_name, event_description, timestamp, members): bool
+ view_event(event_id, event_name): List
+ update_event(event_id): bool
0..*

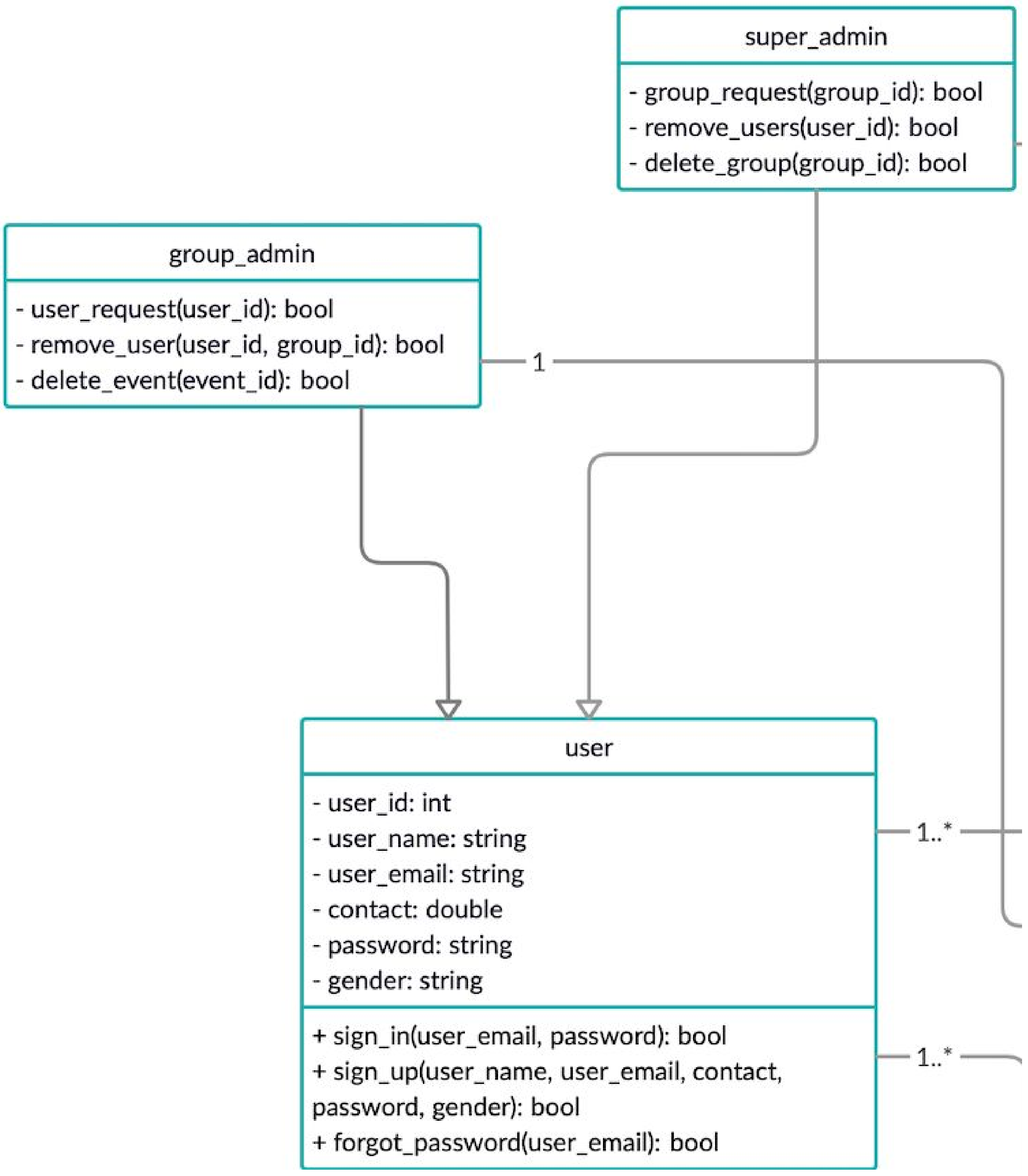
Generalization Relationships

In UML modeling, a generalization relationship is a relationship in which one model element (the child) is based on another model element (the parent).

You can add generalization relationships to capture attributes, operations, and relationships in a parent model element and then reuse them in one or more child model elements. Because the child model elements in generalizations inherit the attributes, operations, and relationships of the parent, you must only define for the child the attributes, operations, or relationships that are distinct from the parent.

The parent model element can have one or more children, and any child model element can have one or more parents. It is more common to have a single parent model element and multiple child model elements. Generalization relationships do not have names.

As the following figures illustrate, a generalization relationship as a solid line with a hollow arrowhead that points from the child model element to the parent model element.


super_admin
- group_request(group_id): bool
- remove_users(user_id): bool
- delete_group(group_id): bool
group_admin
- user_request(user_id): bool
- remove_user(user_id, group_id): bool
- delete_event(event_id): bool
1
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..*

Visibility in Class Diagram

Use visibility indicates who can access the information contained within a class.

There are four types of visibilities :

1. Private visibility hides information from anything outside the class partition. UML notation for private is +
2. Public visibility allows all other classes to view the marked information. UML notation for public is -.
3. Protected visibility allows child classes to access information they inherited from a parent class. UML notation for protected is #.
4. Package or default visibility allows classes to access information within the same package. UML notation for package visibility is ~ symbol

Attributes in Class

In UML models, attributes represent the information, data, or properties that belong to instances of a classifier.

A classifier can have any number of attributes or none at all. Attributes describe a value or a range of values that instances of the classifier can hold. You can specify an attribute's type, such as an integer or Boolean, and its initial value. You can also attach a constraint to an attribute to define the range of values it holds.

Attribute names are short nouns or noun phrases that describe the attribute. The UML syntax for an attribute name incorporates information in addition to its name, such as the attribute's visibility, type, and initial value as shown in the following example.

visibility «stereotype» name : type-expression = initial-value

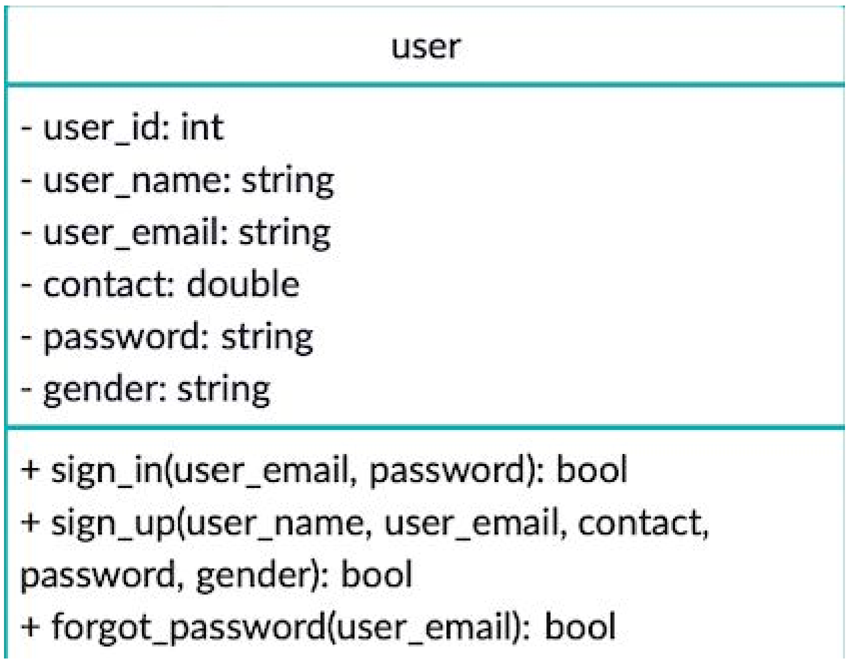
Operations in Class

In UML models, operations represent the services or actions that instances of a classifier might be requested to perform.

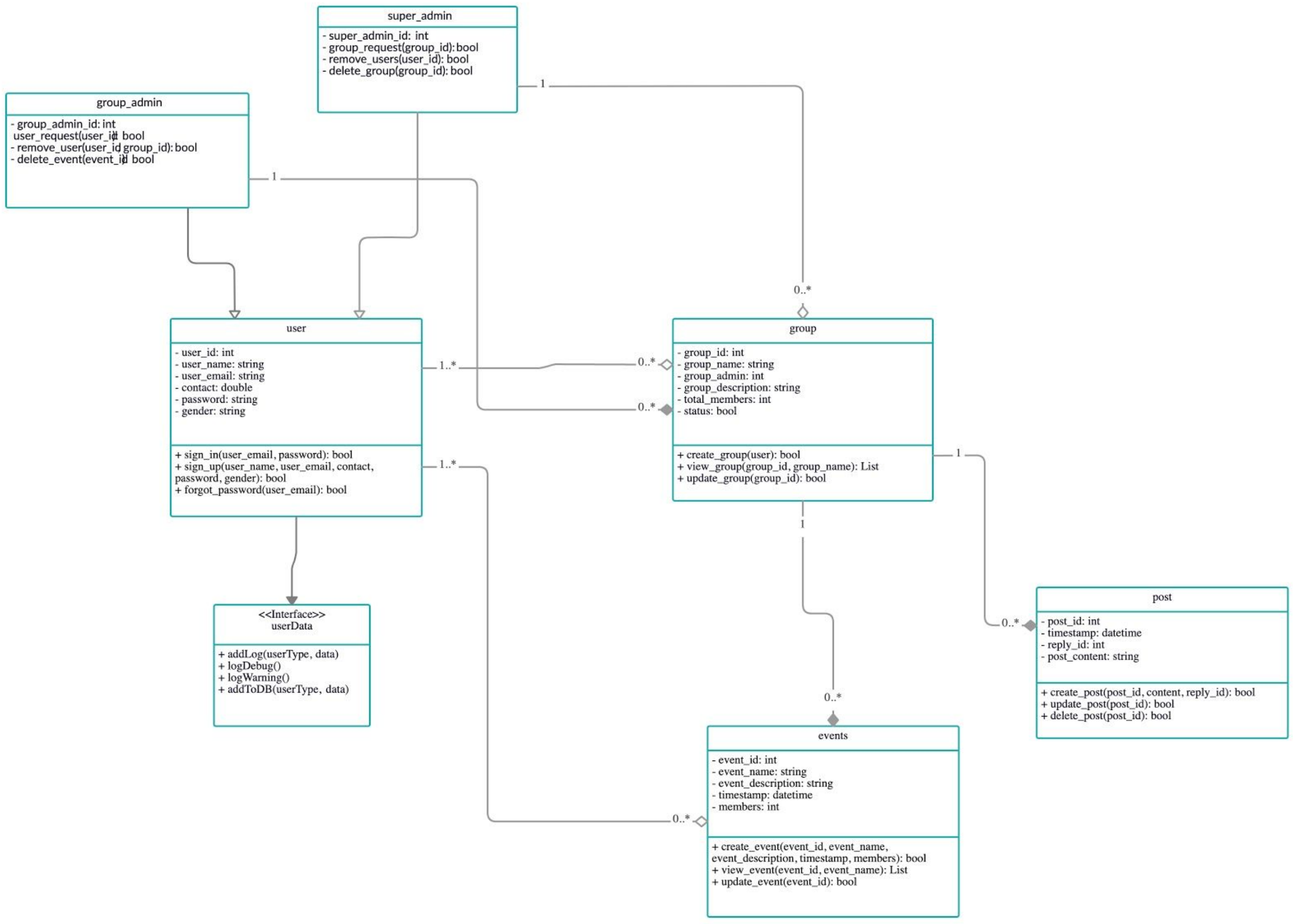
A classifier can have any number of operations or none at all. Operations define the behavior of an instance of a classifier

You can add operations to identify the behavior of many types of classifiers in your model. In classes, operations are implementations of functions that an object might be required to perform. Well-defined operations perform a single task.

Example of Visibility, Attributes and Operations in a Class


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

Class Diagram


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
1
0..*
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..*
0..*
0..*
1..*
group
- group_id: int
- group_name: string
- group_admin: int
- group_description: string
- total_members: int
- status: bool
+ create_group(user): bool
+ view_group(group_id, group_name): List
+ update_group(group_id): bool
1
1
<<Interface>>
userData
+ addLog(userType, data)
+ logDebug()
+ logWarning()
+ addToDB(userType, data)
post
- post_id: int
- timestamp: datetime
- reply_id: int
- post_content: string
+ create_post(post_id, content, reply_id): bool
+ update_post(post_id): bool
+ delete_post(post_id): bool
0..*
0..*
events
- event_id: int
- event_name: string
- event_description: string
- timestamp: datetime
- members: int
+ create_event(event_id, event_name, event_description, timestamp, members): bool
+ view_event(event_id, event_name): List
+ update_event(event_id): bool
0..*

Object Constraint Language for the Class Diagram

Context user::
Invariant: self.login == true

Context super_admin:: group_request(group_id: int): bool
Pre: group_id > 0
Post: group>exists()

Context super_admin:: remove_user(user_id: int): bool
Pre: user_id > 0 and user->exists()
Post: user->delete()

Context super_admin:: delete_group(group_id: int): bool
Pre : group_id > 0 and group->exists()
Post: group->delete()

Context group_admin:: user_request(user_id: int): bool
Pre: user_id > 0
Post: user->exists()

Context group_admin:: remove_user(user_id: int): bool
Pre: user->exists()
Post: user->delete()

Context group_admin:: delete_event(event_id: int): bool
Pre: event_id > 0 and event->exists()
Post: event>delete()

Context group::
Inv: user->exists()

Context group:: create_group(user): bool
Pre: user->exists()
Post: group->exists()

Conclusion

Thus in this assignment, we implemented the class diagram and OCL document for our problem statement.