Sultan Qaboos University - College of Science
Department of Computer Science
COMP5405 : Software Patterns Project - Spring2022

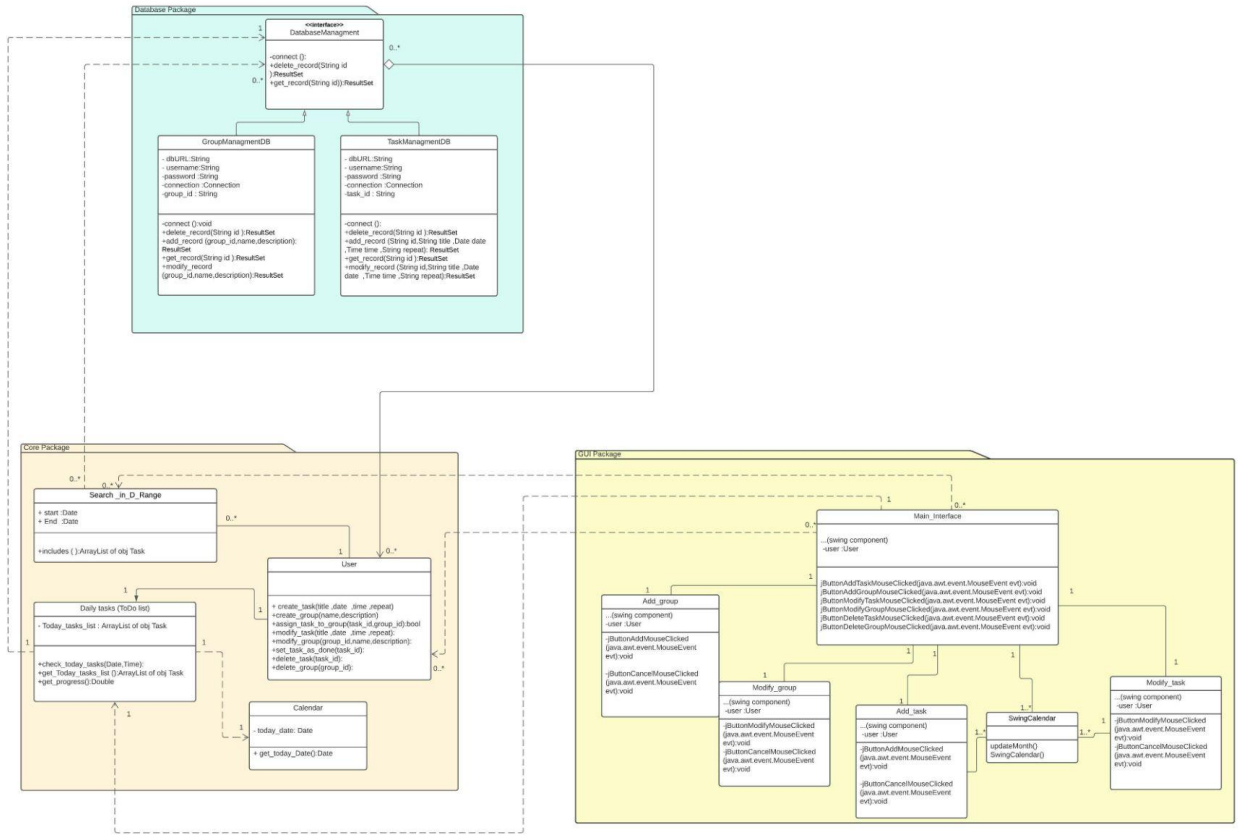
Project Phase 2 : Project Description
Task Manager project

Submitted to: Dr. Yassine Al Jamoussi
Students:

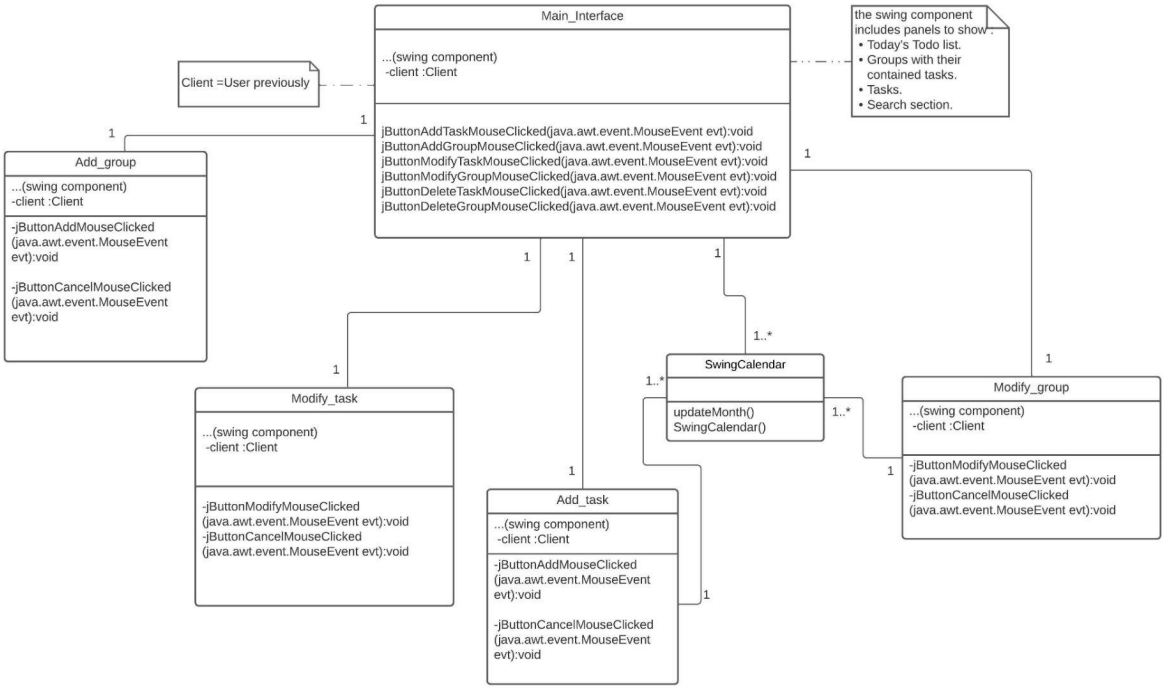
Zayana Al Lamki	120042	**Coordinator**
Fatema Al Ghafri	124394	
Amna Al Nadabi	126558	

Table of Content

Class diagram (including classes related to the GUI and the classes related to the persistence management)



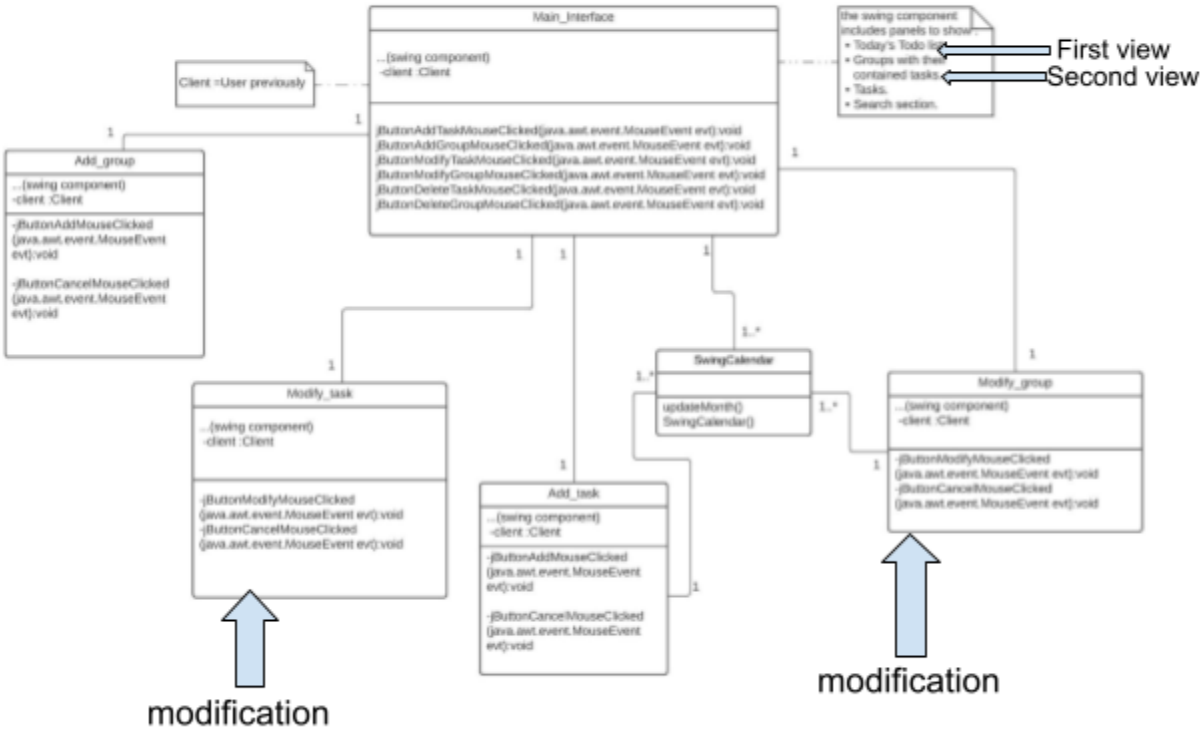
UI Diagram



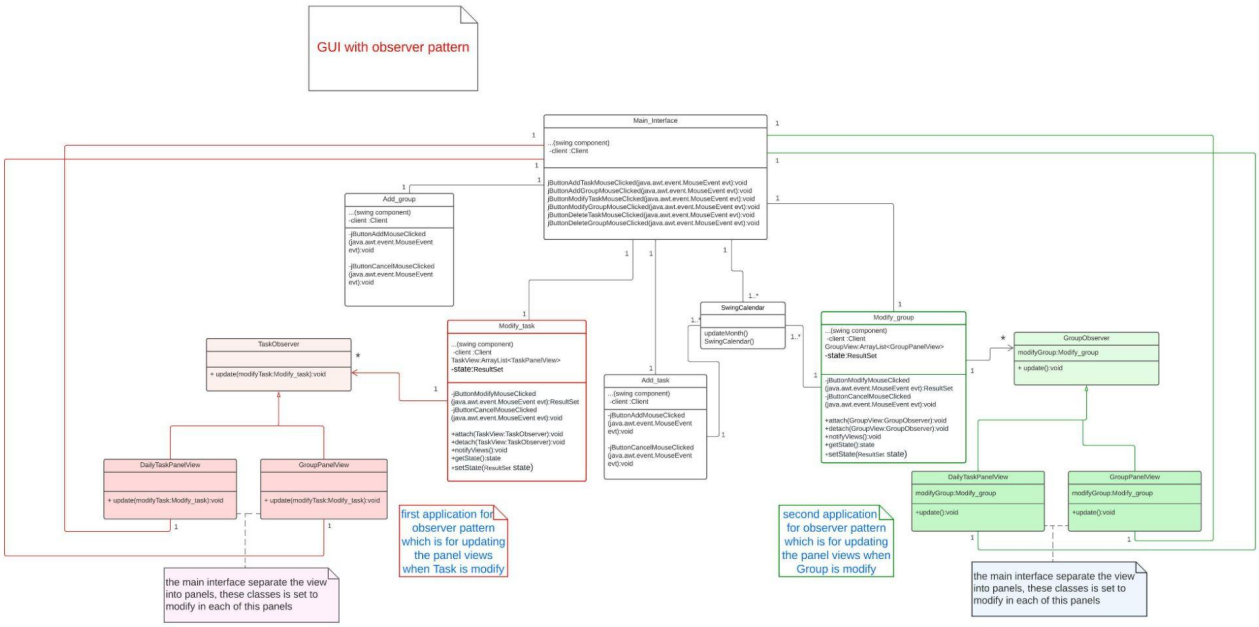
Observer Pattern

The intent of the pattern is to define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

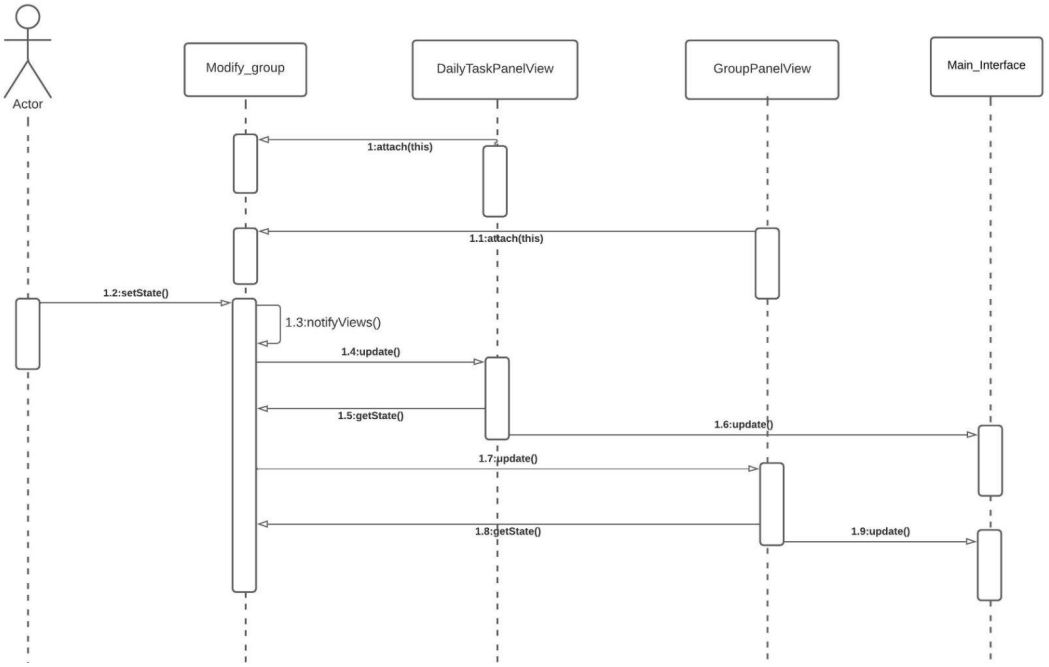
Since our main class has two different panels for showing the list of today's tasks and all the groups that the user created with the tasks associated with each group, we have to update these views each time the user modifies a group or task. By applying the observer pattern in two separate locations for each modification, the two panel views will be notified to update their information with the newly received adjustments. The user should see all his modifications after he did them. See below diagram.



The participants that are involved in the two applications of the pattern are Modify_group and Modify_task which present the **subject and the concrete subject** on the patterns. The **observers** are GroupObserver and TaskObserver and the **concrete observers** for each of the subjects are DailyTaskPanelView and GroupPanelView which are basic classes needed to update the panels on the Main_Interface whenever a modification is done.



The above diagram is presenting the association of the participant classes on the observer pattern in the graphical user interface. The two applications of the pattern are presented with different colors, green for the first application and the red color of the second application. Each of these applications are working similarly and have the same target which is to modify the panel on the main_interfave class, the only difference is that the first one updates when a task is modified and the sound one is when a group is modified.



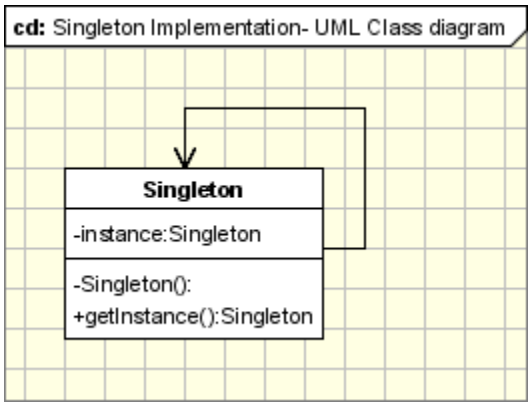
In this sequence diagram is explaining how the observer pattern is applied on this application. The user is setting the state by modifying the group/task then the subject Modify_group will notify the views about the modification, the two concrete observers DailyTaskPanelView and GroupPanelView are updating and getting the state needed to modify and refresh with information on the Main_Interface for a single update each of the two panels using the concrete observers.

Singleton pattern

The intent of this pattern is to Ensure that only one instance of a class is created , and to provide a global point of access to the object. And it is used in core java classes.

We choose the singleton pattern to apply in our class diagram especially in the core package because we find that it is the suitable one for our project.

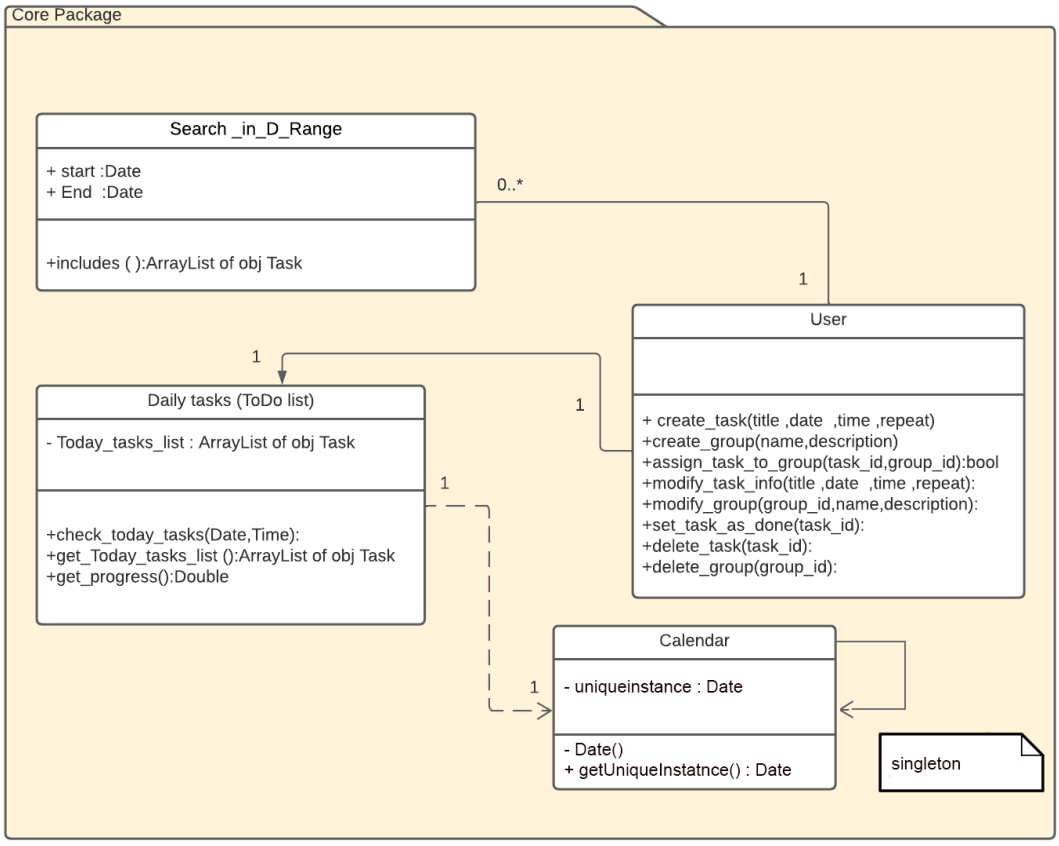
Singleton simply includes one class, which is responsible for ensuring that there is only one instance; it accomplishes this by instantiating itself while also providing a global point of access to that instance. The singleton class does this to ensure that the same instance may be used from everywhere, prohibiting direct use of the singleton constructor.



In our project , in the core package , we have a calendar that returns the date to the user , which always returns a unique instance. So we can use the singleton pattern there. A class using the singleton design pattern will include,

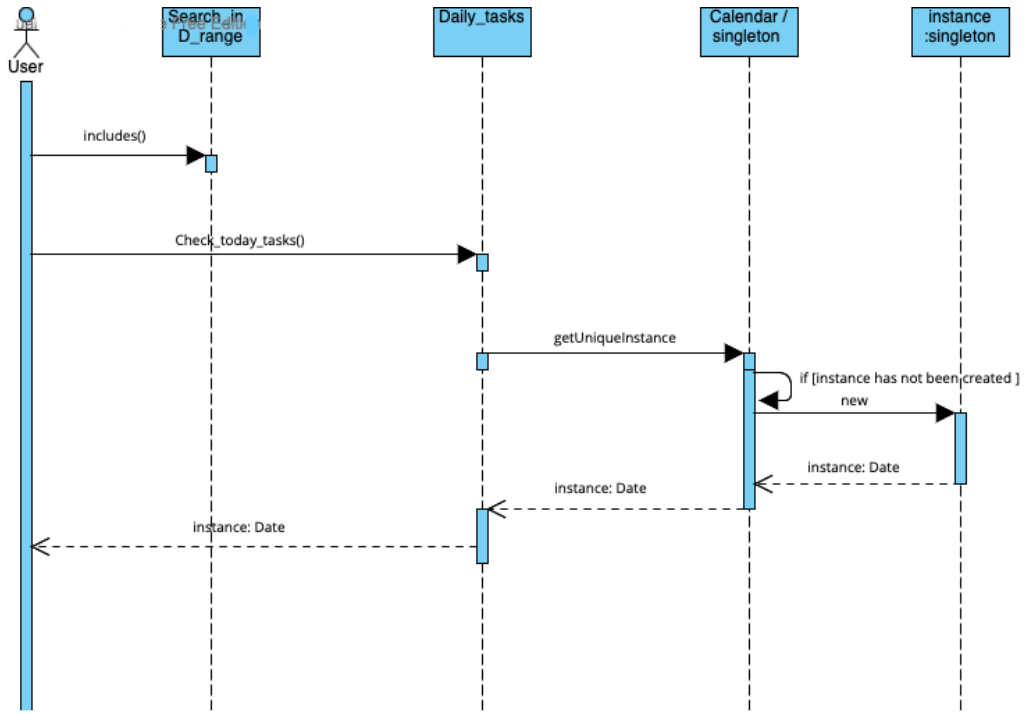
1. A private static variable, holding the instance of the class.
2. A private constructor.
3. A public static method, to return the single instance of the class.

Here is our core package class diagram with the singleton pattern.



The diagram above shows The class diagram of the core package which we add a singleton pattern to it. So the user will ask to search for tasks between two dates and also it asks for the daily tasks that will be there. Daily tasks need to know the date which will be checked from the calendar singleton.

The sequence diagram of the singleton pattern :



This sequence diagram is explaining how the singleton pattern is applied on our project especially in the core diagram. The user will ask to check the daily tasks then Daily_tasks will need to find the date of the day so it will send it to the calendar class which implements the singleton pattern on it and it will return the date.

Bridge pattern

Bridge is a structural design pattern that lets you split a large class or a set of closely related classes into two separate hierarchies: abstraction and implementation which can be developed independently of each other.

Note : we think the name of User class is not meaningful ,but we kept it in the class diagram so you don't think that we included a new class and to avoid confusion .In the pattern class diagram the name is changed to Basic Control along with other changes .

As it is shown in the class diagram we have two classes (implementations) for DatabaseManagment which are GroupManagmentDB and TaskManagmentDB. Also we can see that different classes in GUI packages have a User object which indicates that User is being the connection between GUI package and Database package since User has an object of type DatabaseManagment. The problem we are facing with this design is that when a certain class in GUI make use of one method on User class is has the access to all types of DatabaseManagment .As an example to explain more Add_group on GUI package has a User object which will allow the Add_group to access all the method on user like delete group and others .On the other hand user class gives the access to both types of management systems which should be not allowed .That is why we added the abstraction Basic control and used bridge to make them independent of each other .Now the client of each GUI class will link the abstraction object (basic control) to only one implementation object (GroupManagmentDB or TaskManagmentDB).Now after applying bridge Add_group will have Client object which will connect it to one implementation only GroupManagmentDB throughout the abstraction Basic_Control .

Using bridge patterns now the control classes are independent to the concrete implementation of the management systems which means we can extend more types of control and get implementations of new kinds of management systems without affecting each other .

Another reason to use bridge is that the implementation of DatabaseManagment will change on the runtime since the user may add a task or add a group and other features which require to use the suitable management implementation for it .Using bridge in our case satisfied Open Closed Principle where we can extend more abstractions and there are closed to modification .

Participants :

Abstraction: (BasicControl):

Implementation: (DatabaseManagment):

the database management systems on a persistent layer .

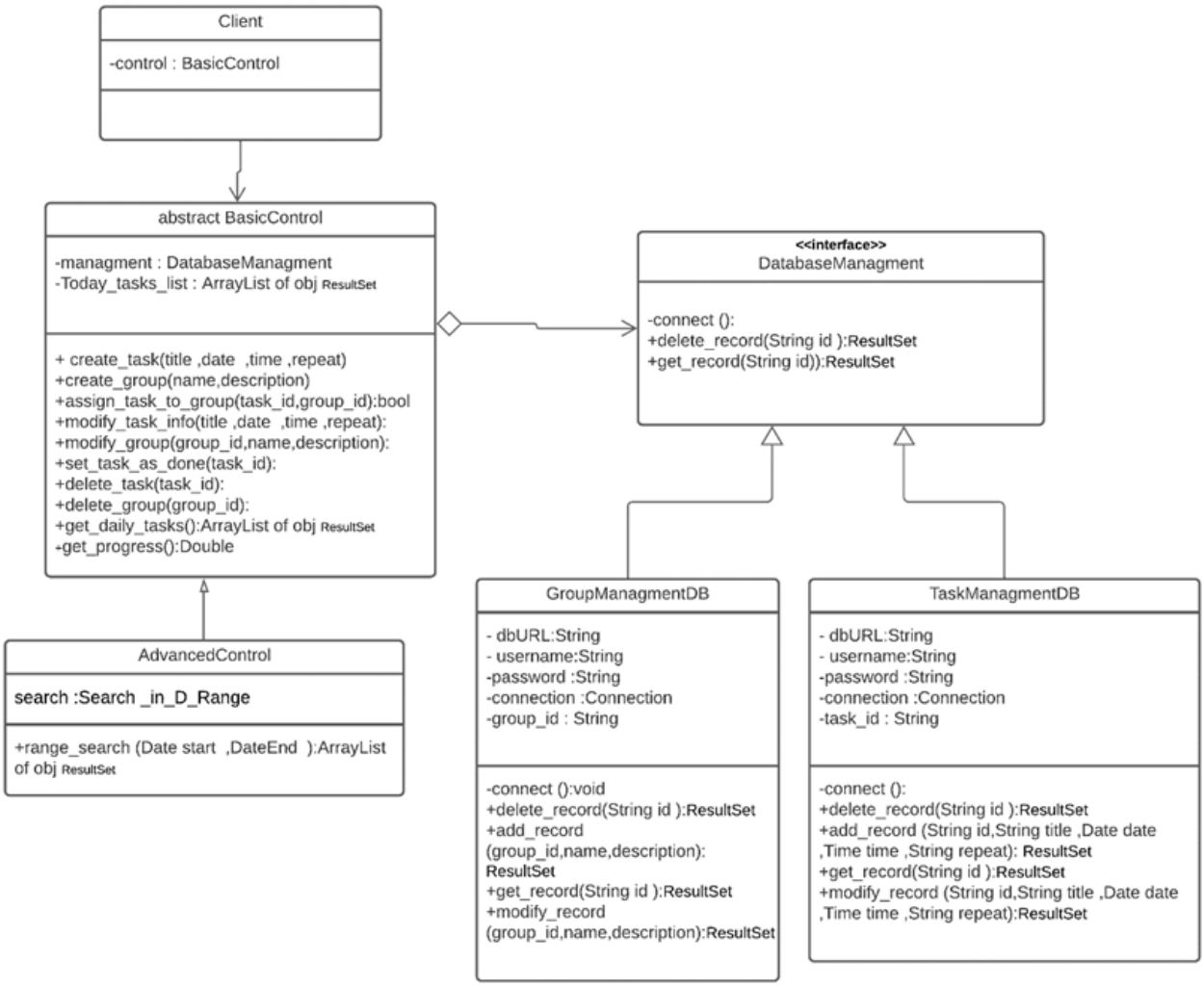
Concrete Implementations :(GroupManagmentDB and TaskManagmentDB):

Refined Abstractions : (AdvancedControl):

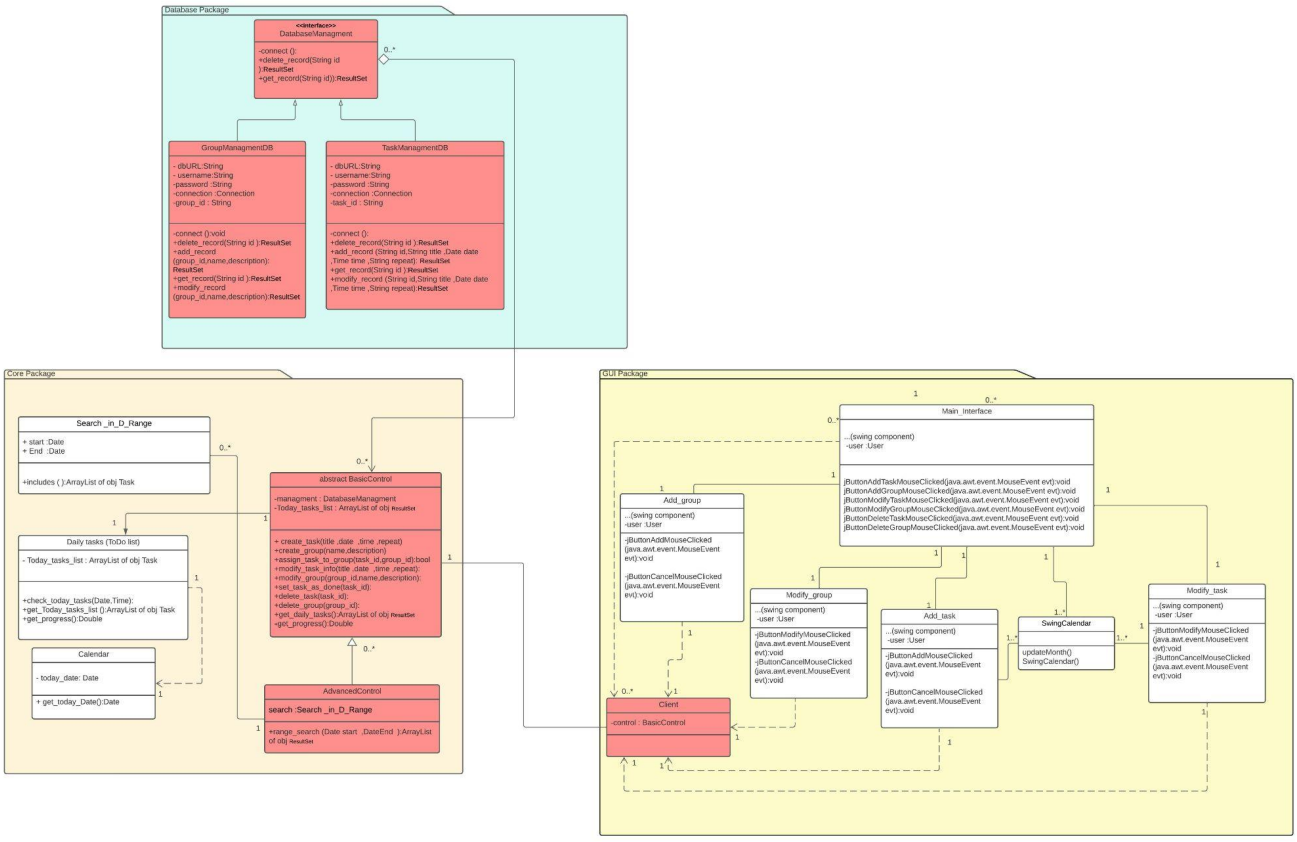
Client : (Client):

the client does not work with the implementation directly; it is connected to the control .

The client code should pass an implementation object to the abstraction’s constructor to associate one with the other.



The pattern class diagram structure isolated from all other classes



Participants in the bridge pattern are in dark pink

Description :

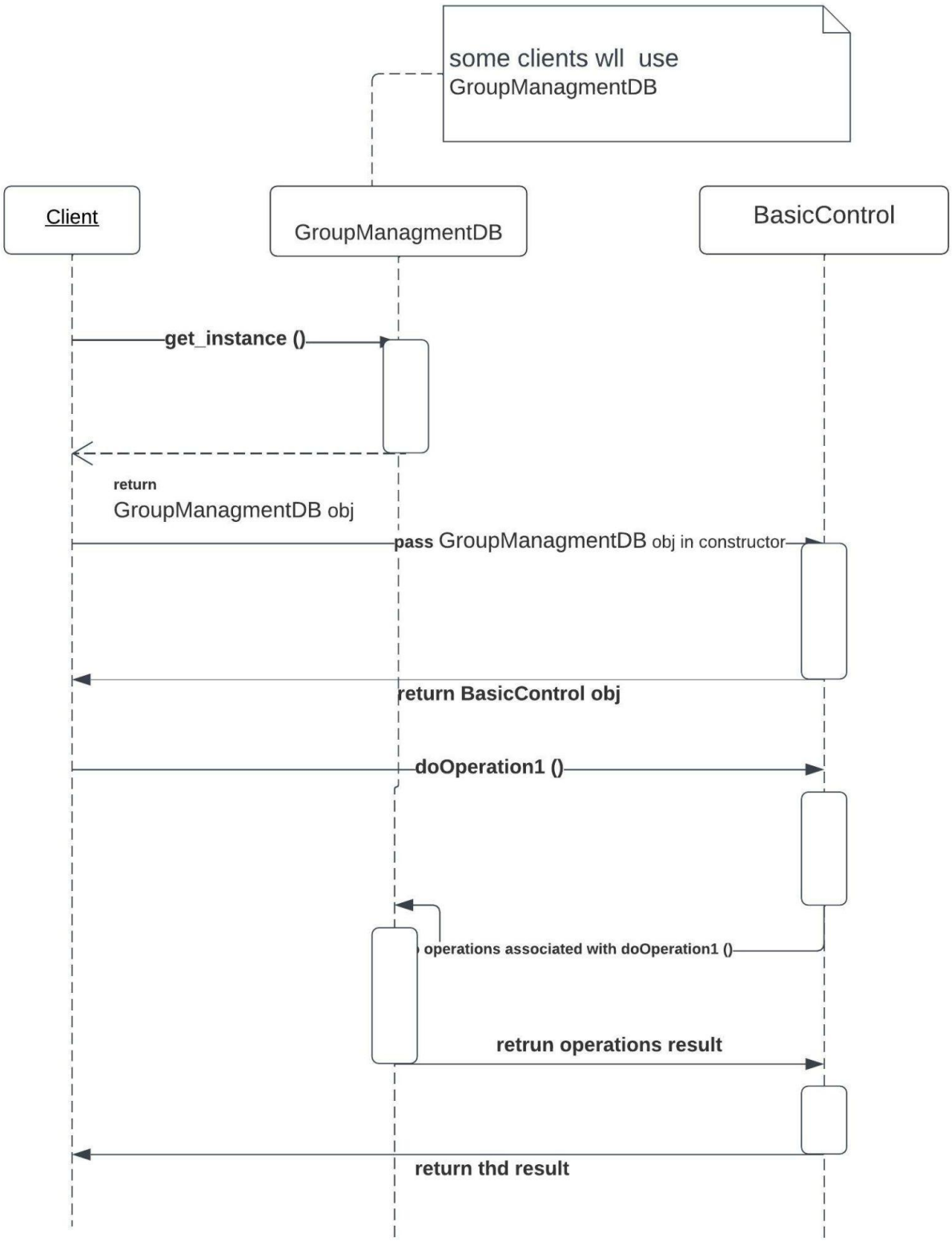
As an example Client in Add_group class will contain the following :

DatabaseManagment GroupManag = new GroupManagmentDB ();

BasicControl control = new BasicControl (GroupManag);

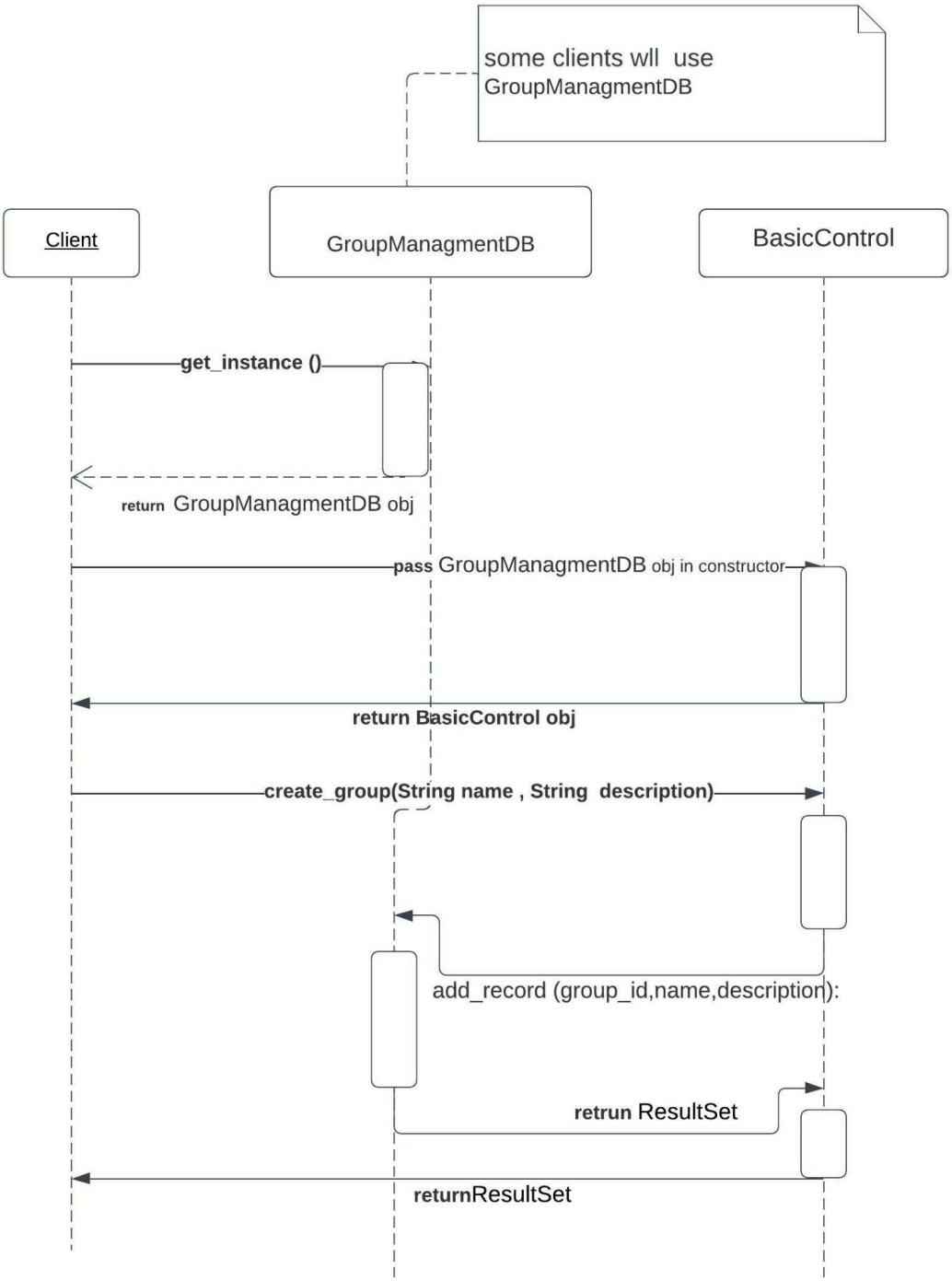
Control.create_group(name,description);èthis method would change based on the GUI class

In side create_group method has the method call add_record() which exists in the concrete implementation GroupManagmentDB è the called method also can change based on the abstraction method call .

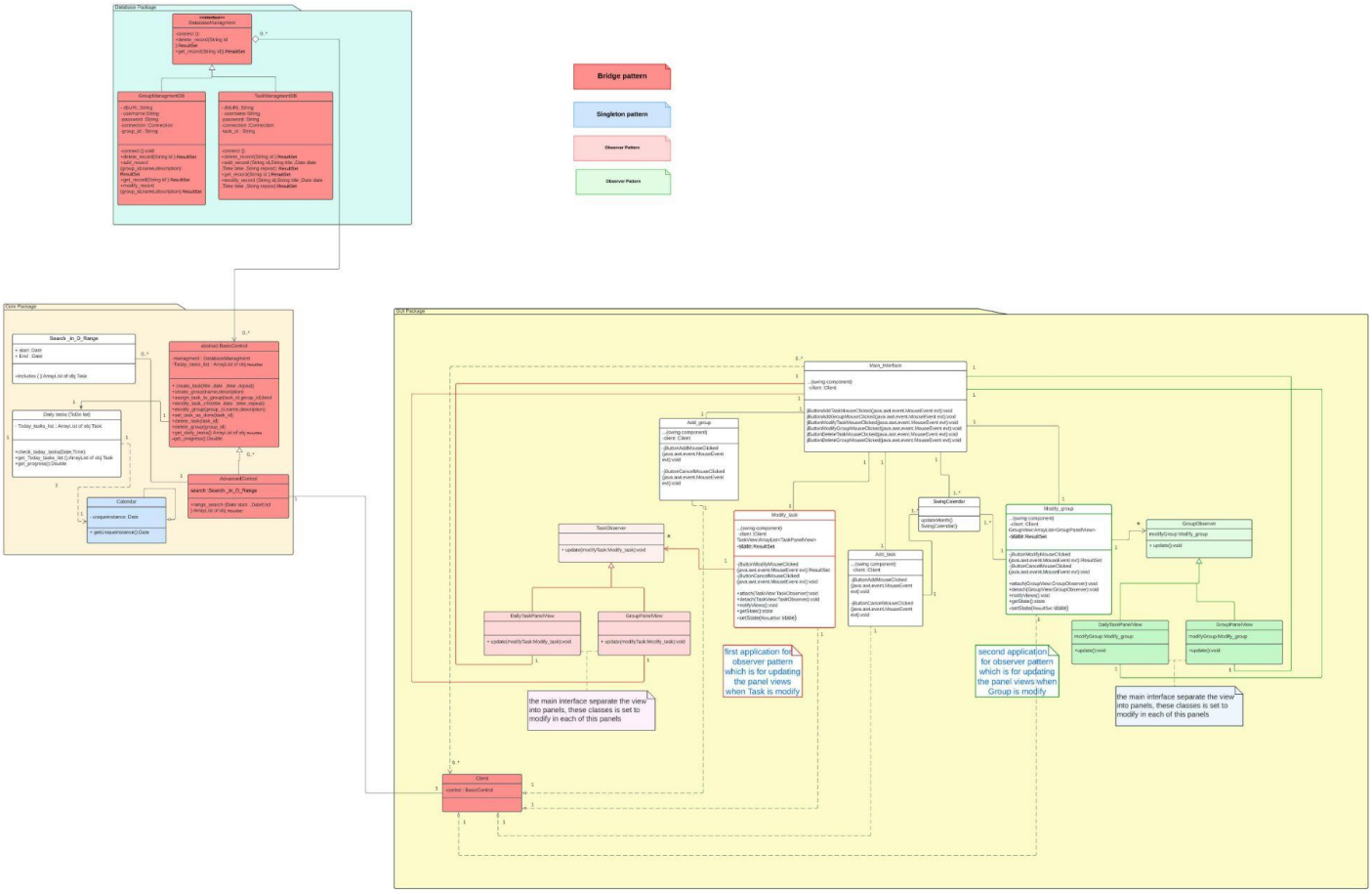

some clients wll use
GroupManagmentDB
Client
GroupManagmentDB
BasicControl
get_instance ()
return
GroupManagmentDB obj
pass GroupManagmentDB obj in constructor
return BasicControl obj
doOperation1 ()
operations associated with doOperation1 ()
retrun operations result
return thd result

Do operation1() can be any method on the abstraction Basic control according to it the method will be used from the concrete implementation will be called and returned to BasicControl then returned to Client .

This is a more specific example for add_group Client object behavior :



Class diagram with Patterns



Contribution

Team member	Contribution
Zayana Al Lamki	Class diagram , UI diagram , Observer Pattern , class diagram with pattern.
Fatma Al Ghafri	Class diagram , UI diagram , Singleton Pattern , class diagram with pattern.
Amna Al Nadabi	Class diagram , UI diagram , bridge Pattern , class diagram with pattern.