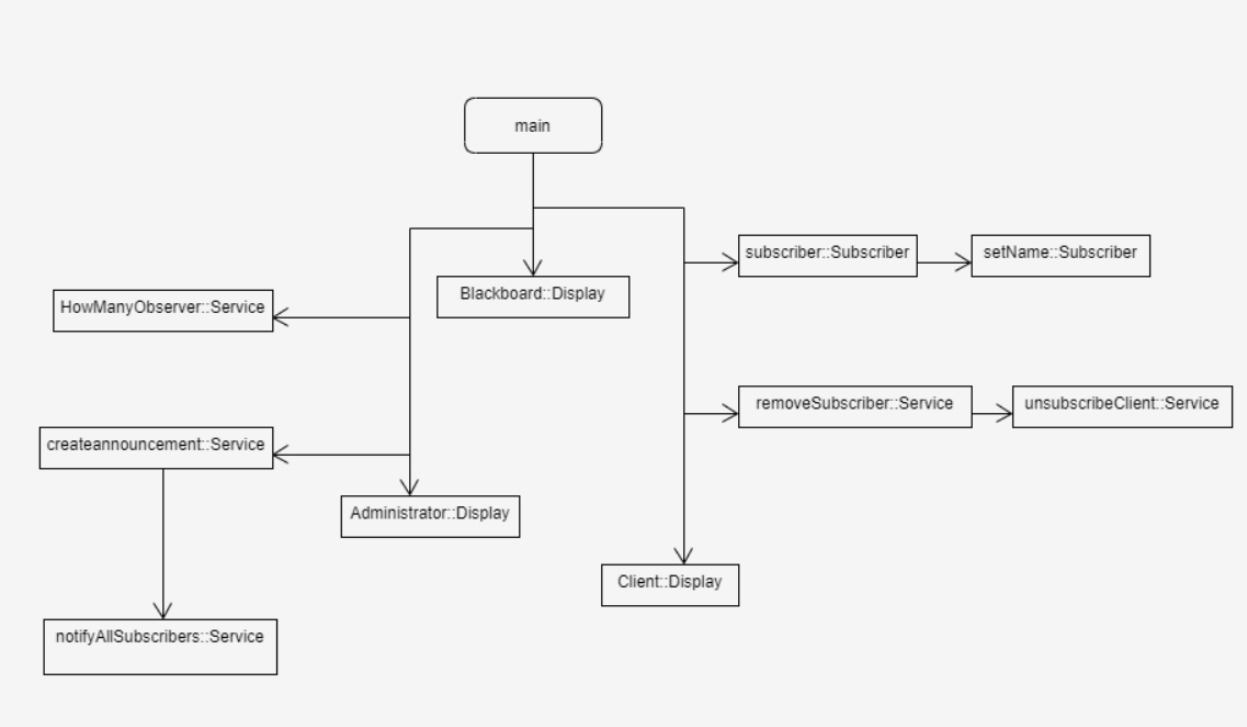
Report-2

Section 1 (Specifications):

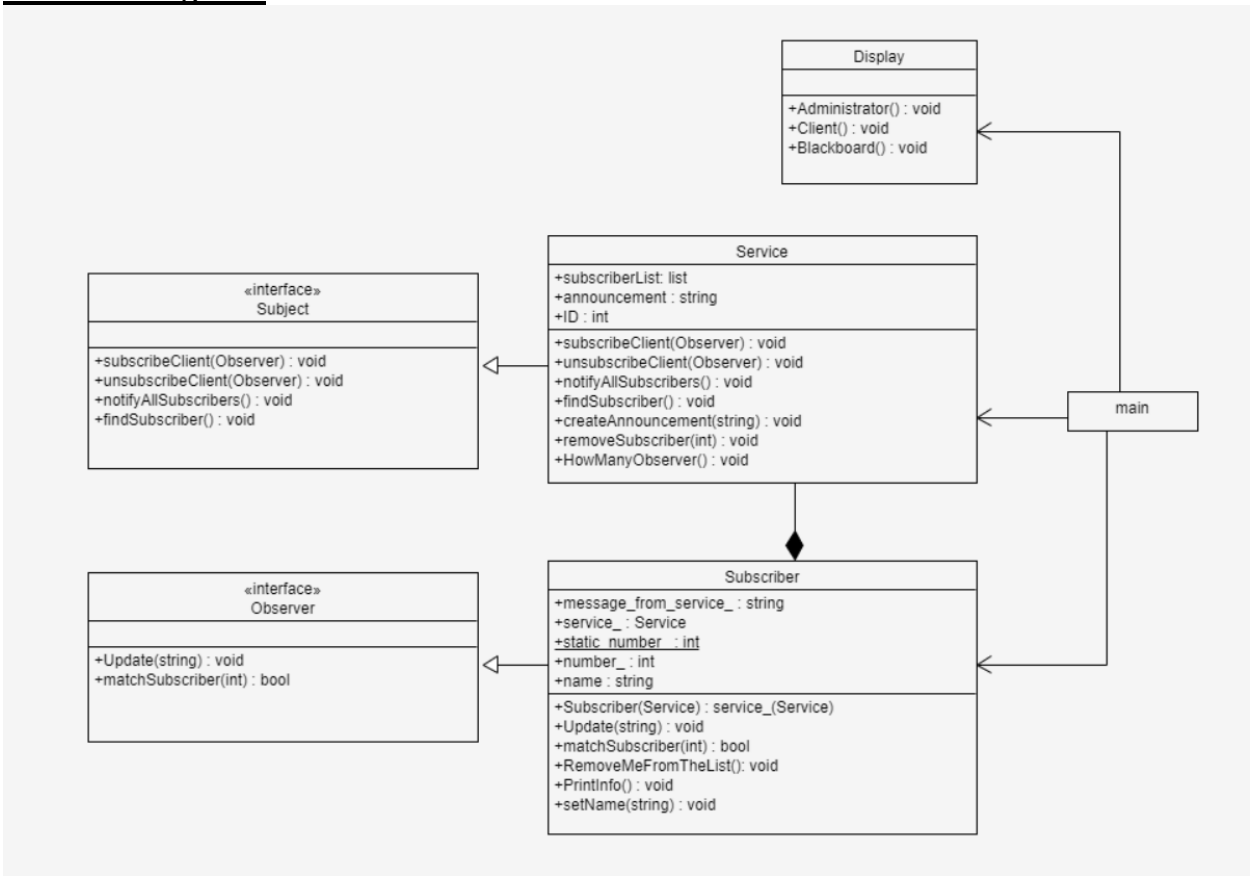
In the 2nd Scenario, I select the 2nd case. In this case I have to design and implement a subscription service called NewsOnScreen. Notifying announcement to the subscriber of this service on the VLE Blackboard from Department of Computer Science at USW. If subscribers want, they could unsubscribe from this service anytime when they want to stop receiving updates. This design pattern will let me define a subscription mechanism to notify multiple objects about any announcement that happens to the object they're observing. For this case I selected the Observer design pattern which provides one to many relationships between objects. The way I have used this design pattern is, Since this pattern uses r actor classes. Subject, Observer, Subscriber and Service. Subject and Observer are interface and concrete class Service and Subscriber that extends Subject and Observer respectively. Display is the additional class to display some info through its object. Subscribers add through the object of service class. And the subscriber who wants to unsubscribe the service is removed from the list through the function of the object of the Subscriber class. This calls a service object to find the object from the list. After finding objects was removed from the list. And for notifying, the announcement is shown to all the objects of the list.

Section 2 (Design):

2.1 Call Graph:



2.2 UML Diagram:



2.3 Implementation:

```
// interfaces
class Observer
{
public:
    virtual void Update(const string &message_from_subject) = 0;
    virtual bool matchSubscriber(const int id) = 0;
};

class Subject
{
public:
    virtual void subscribeClient(Observer *observer) = 0;
    virtual void unsubscribeClient(Observer *observer) = 0;
    virtual void notifyAllSubscribers() = 0;
    virtual void findSubscriber() = 0;
};
```

```
// Service class extends Subject public interface
class Service : public Subject
{
// fucntion of this Service class which is public
public:
    // Subscription management method
    void subscribeClient(Observer *observer) override
    {
        subscriberList.push_back(observer);                                     // adding object to list
    }

    void unsubscribeClient(Observer *observer) override
    {
        subscriberList.remove(observer);                                       // removing object form list
    }

    void notifyAllSubscribers() override
    {

        std::list<Observer *>::iterator itr;
        cout << "\n-------------------------NewsOnScreen-------------------------";
        cout << "\n-------------------------VLE Blackboard-------------------------\n\n";
        for(itr = subscriberList.begin(); itr != subscriberList.end(); ++itr)
        {
            (*itr)->Update(announcement);                                                   // calling Update() function of Subscriber class with string parameter

        }

    }

    void findSubscriber() override
    {
        bool isMatched = false;
        std::list<Observer *>::iterator itr;                                            // list iterator
        for(itr = subscriberList.begin(); itr != subscriberList.end(); ++itr)
        {
            isMatched = (*itr)->matchSubscriber(ID);                                    // calling matchSubscriber() function of Subscriber class which will re

```

```
        std::list<Observer *>::iterator itr;                                            // list iterator
        for(itr = subscriberList.begin(); itr != subscriberList.end(); ++itr)
        {
            isMatched = (*itr)->matchSubscriber(ID);                                   // calling matchSubscriber() function of Subscriber class which will re

            if(isMatched == true)
                break;
        }

    }

    void createAnnouncement(std::string message = "Empty")
    {
        this->announcement = message;                                   // initializing variables value
        notifyAllSubscribers();                                         // calling notifyAllSubscriber() function of this class
    }

    void removeSubscriber(int sID)
    {
        this->ID = sID;                                                 // initializing value
        findSubscriber();                                               // calling findSubscriber() function
    }

    void HowManyObserver()
    {
        cout << "\nThere are " << subscriberList.size() << " subscribers in the service.\n";                                  // printing value of list of subscriber
    }


// private variables
private:
    std::list<Observer *> subscriberList;
    string announcement;
    int ID;
};
```

```
// Subscriber class extends Observer public interface
class Subscriber : public Observer
{
// fucntion of this Service class which is public
public:
    // constructor
    Subscriber(Service &service) : service_(service)
    {
        this->service_.subscribeClient(this);                           // calling subscribeClient function of Service class
        cout << "ID:" << ++Subscriber::static_number_;
        this->number_ = Subscriber::static_number_;                     // initializing number variable
    }

    void Update(const std::string &message_from_service) override
    {
        message_from_service_ = message_from_service;               // initializing variables value
        PrintInfo();                                                // calling printInfo() function of this Subscriber class
    }

    bool matchSubscriber(const int id) override
    {
        if(this->number_ == id)
            {
                RemoveMeFromTheList();                      // calling RemoveMeFromTheList() function of Subscriber class
                return true;                                // returning value
            }
        else
            return false;                                   // returning value
    }

    void RemoveMeFromTheList()
    {
        service_.unsubscribeClient(this);                                       // callling ununsubscribeClient of Service class through object
        cout << this->name << " ID: " << this->number_ << ", unsubscribed this service.\n";          // printing value
    }

    void PrintInfo()
```

```
    void PrintInfo()
    {
        // print some value
        cout << "Subscriber name: " << this->name << ", ID:" << this->number_ << " => a new message is available --> " << this->message_from_service_ << "\n";
    }

    void setName(string uName)
    {
        this->name = uName;                     // intializing value of a variable
    }


// some private variable
private:
    string message_from_service_;
    Service &service_;
    static int static_number_;
    int number_;
    string name = " ";
};


// Initialize pointer to zero so that it can be initialized in first call to getInstance
int Subscriber::static_number_ = 0;
```

```
class Display
{
public:
// Administrator function
void Administrator()
{
    cout << "\n\n";
                cout << "-------------------------NewsOnScreen-------------------------\n\n";
                cout << "1. How many subscribers are there in this service?\n";
                cout << "2. Make an Announcement.\n";
                cout << "3. Go back.\n";
                cout << "\nEnter your number: ";
}

// Client function
void Client()
{
cout << "\n\n";
                cout << "-------------------------NewsOnScreen-------------------------n\n";
                cout << "1. Subscribe NewsOnScreen service.\n";
                cout << "2. Unsubscribe NewsOnScreen service.\n";
                cout << "3. Go back.\n";
                cout << "\nEnter your number: ";
}

// Blackboard function
void Blackboard()
{
    system("CLS");
        cout << "-------------------------NewsOnScreen-------------------------\n\n";
        cout << "1. Client.\n";
        cout << "2. Administrator.\n";
        cout << "3. Exit.\n";
        cout << "\nEnter your number: ";
}
};
```

```
// main function
int main()
{


    //by default we are making some subscriber
    Service *service = new Service;
    Subscriber *subscriber1 = new Subscriber(*service);
    subscriber1->setName("Srijon");
    Subscriber *subscriber2 = new Subscriber(*service);
    subscriber2->setName("Sijan");
    Subscriber *subscriber3 = new Subscriber(*service);
    subscriber3->setName("Suhel");
    Subscriber *subscriber4 = new Subscriber(*service);
    subscriber4->setName("Tahmid");
    Subscriber *subscriber5 = new Subscriber(*service);
    subscriber5->setName("Sakib");
    Subscriber *subscriber6 = new Subscriber(*service);
    subscriber6->setName("Oly");

    //creating Display class object to print some info
    Display *display = new Display();


    while (true)
    {

        display->Blackboard();                                          // calling Blackboard() function to print some info
        int opn;
        cin >> opn;


        if (opn == 1)
        {
            while (true)
            {
```

```
    if (opn == 1)
    {
        while (true)
        {

            display->Client();                                          // calling Client() function to print some info
            int opn;
            cin >> opn;

            if (opn == 1)
            {
                cout << "Enter your name? (small letter)\n";
                cout << "Name: ";
                string name;

                // taking input of string
                getline(cin >> ws, name);                                   // Usage of std::ws will extract all the whitespace character

                cout << endl << name << " subscribed NewsOnScreen service. Your ";
                Subscriber *subscriber = new Subscriber(*service);          // adding Subscriber through creating object of Subscriber class with object
                subscriber->setName(name);                                  // calling setName() method of Subscriber to set name through passing string
                //Subscriber *name = new Subscriber(*service);
            }
            else if (opn == 2)
            {
                cout << "Enter your ID: ";
                int ID;
                cin >> ID;
                service->removeSubscriber(ID);                                // calling removeSubscriber() function of Service class with int parameter
            }
            else if (opn == 3)
            {
                break;
            }
            else
                cout << "\nPlease enter correct number.\n";
        }
```

```
    else if (opn == 2)
    {
        while (true)
        {

            display->Administrator();                                   // calling Administrator() function to print some info
            int opn;
            cin >> opn;

            if (opn == 1)
            {
                service->HowManyObserver();                             // calling HowManyObserver() function of Service class to know the number of subsc
            }
            else if (opn == 2)
            {

                cout << "Enter your announcement: \n";
                string notice;

                // Usage of std::ws will extract all the whitespace character
                getline(cin >> ws, notice);

                service->createAnnouncement(notice);                    // calling createAnnouncement() function of Service class to create an announcemen
            }
            else if (opn == 3)
            {
                break;
            }
            else
                cout << "\nPlease enter correct number.\n";
        }
    }
```

```
                }
                else
                    cout << "\nPlease enter correct number.\n";
            }
        }


        else if (opn == 3)
            return 0;
        else
            cout << "\nPlease enter correct number.\n";
    }


}
```

Section 3 (Discussion):

The chosen design pattern is the Observer design pattern. Which is a Behavioral design pattern. Among all other design patterns this observer design pattern is best for this case. In this case it will need a design pattern which uses a one-to-many relationship between objects. This observer design pattern provides these features. In the 2nd scenario case-4 there needs to be some features like subscribing and unsubscribing. For subscribing and unsubscribing there is a list where objects are added when subscribed and removed when unsubscribed. Observer design pattern provides this loose coupling as: a. Subject only knows that the observer implements the Observer interface. Nothing more. b. There is no need to modify the subject to add or remove observers. c. we can reuse subject and observer classes independently of each other. In this design pattern. In this observer pattern if a user subscribes to this service no longer need to ask for the announcement. Instead the administrator sends the announcement directly to VLE Blackboard right after publish or even advance. The administrator has a list of subscribers. Since we used an observer design pattern in this application, the administrator could be notified about the announcement with the subscribers at once.

The solid principles of OOP are Single-responsibility-Principle, Open-closed-principle, Liskov-Substitution-principle, Interface-segregation-principle and Dependency-Inversion-Principle. And the observer pattern used in this case is full the SOLID principle of OOP.