# Course assignment

VIA Management System

Daniela Koch 266502
Matej Michalek 266827
Michał Karol Pompa 26649
Michaela Golhova 266099

**Supervisors:**

Joseph Chukwudi Okika

Steffen Vissing Andersen

ICT Engineering

2nd Semester

May 2018

## Abstract

*The main aim of the course assignment was to make a client-server system that would contain as many design patterns as possible. The system was made for Vipassana (a centre for spiritual events) and was providing the user with a list of actions such as signing up new members, creating events, searching for lecturers or generating a newsletter. Each of the design patterns, as well as the client-server architecture, has been documented. Each documentation involves a description of analysis, design, implementation and (if possible) testing of a specific design or architecture pattern. The list of design patterns that can be found in the system is as follows: singleton, flyweight, observer, proxy, adapter and model-view-controller. Moreover, the assignment was developed using SCRUM. The use of it can be found in the documentation.*

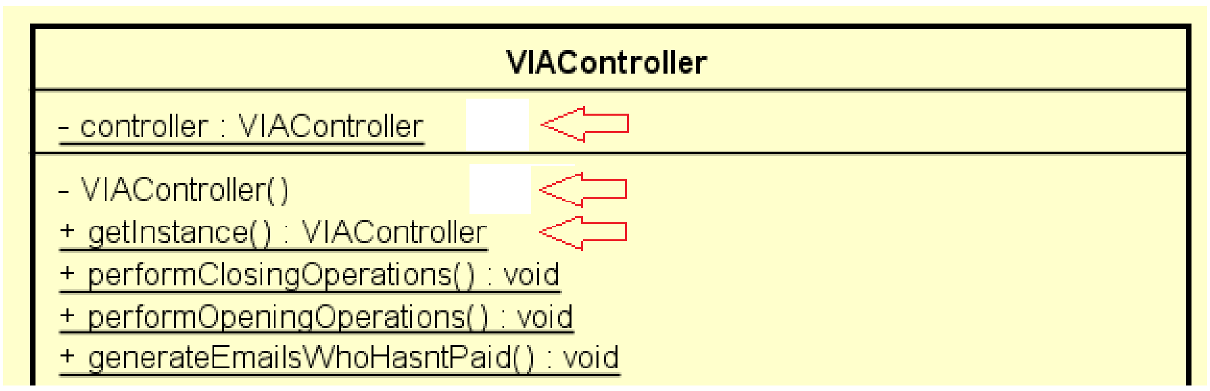
## Table of content

# 1. Design patterns

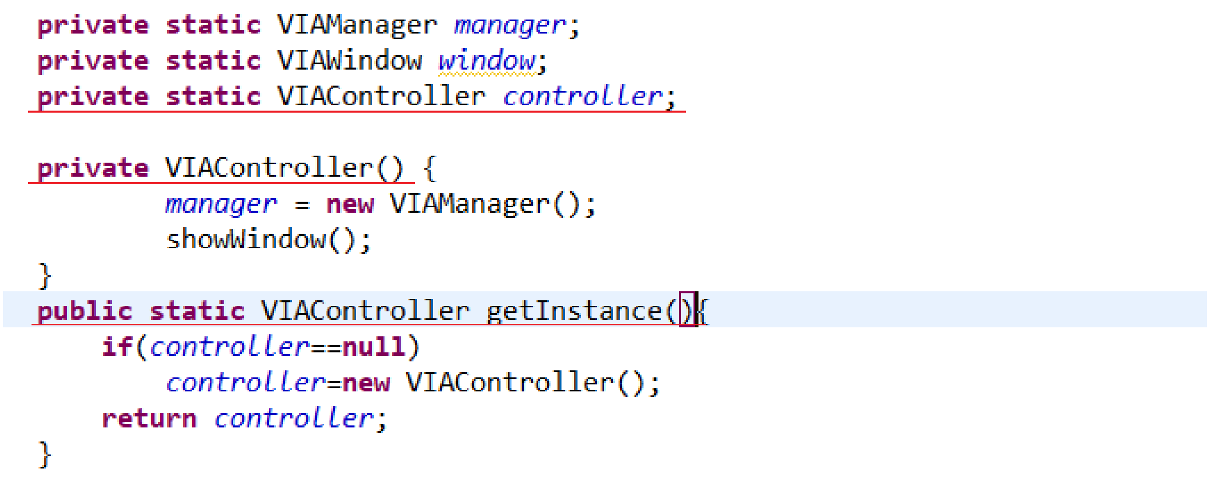
## 1.1. Singleton

The purpose of a singleton is to ensure that there is only one instance of a chosen class in the whole system as well as providing a global access to it. That is what a controller needs. The controller is a class that controls the whole system and provides a connection between the model and the view. These are the reasons why only one instance of it should exist and why a global access from other classes is needed.

Looking from the design point of view, there are three rules that must be fulfilled to call a class a singleton: a static instance of its own, a private constructor and a static get instance method. These are illustrated on a class diagram:


VIAController
- controller : VIAController
- VIAController()
+ getInstance() : VIAController
+ performClosingOperations() : void
+ performOpeningOperations() : void
+ generateEmailsWhoHasntPaid() : void

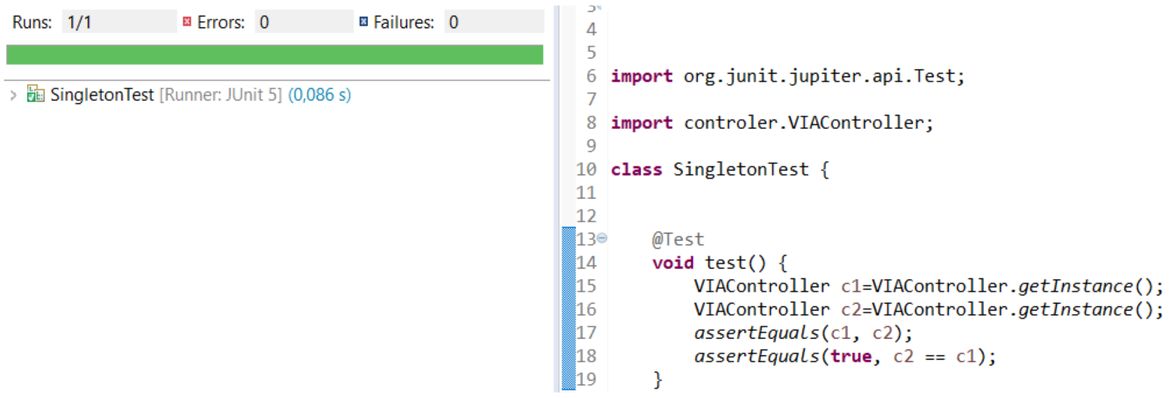
Translating it into implementation (the whole implementation can be found in appendix 1):



```
private static VIAManager manager;
private static VIAWindow window;
private static VIAController controller;

private VIAController() {
        manager = new VIAManager();
        showWindow();
}
public static VIAController getInstance(){
    if(controller==null)
        controller=new VIAController();
    return controller;
}
```

The getInstance method provides a global access to the instance thanks to being static. At the same time it ensures there is only one instance of the class, returning the instance if it already exists and creating it if it does not. Ensuring there is only one instance is also achieved with the private constructor, which does not allow creating an instance of the controller outside the VIAController class.

The Singleton design pattern has been tested using JunitTest and proved to be working:



```
Runs: 1/1    Errors: 0    Failures: 0

SingletonTest [Runner: JUnit 5] (0,086 s)
```

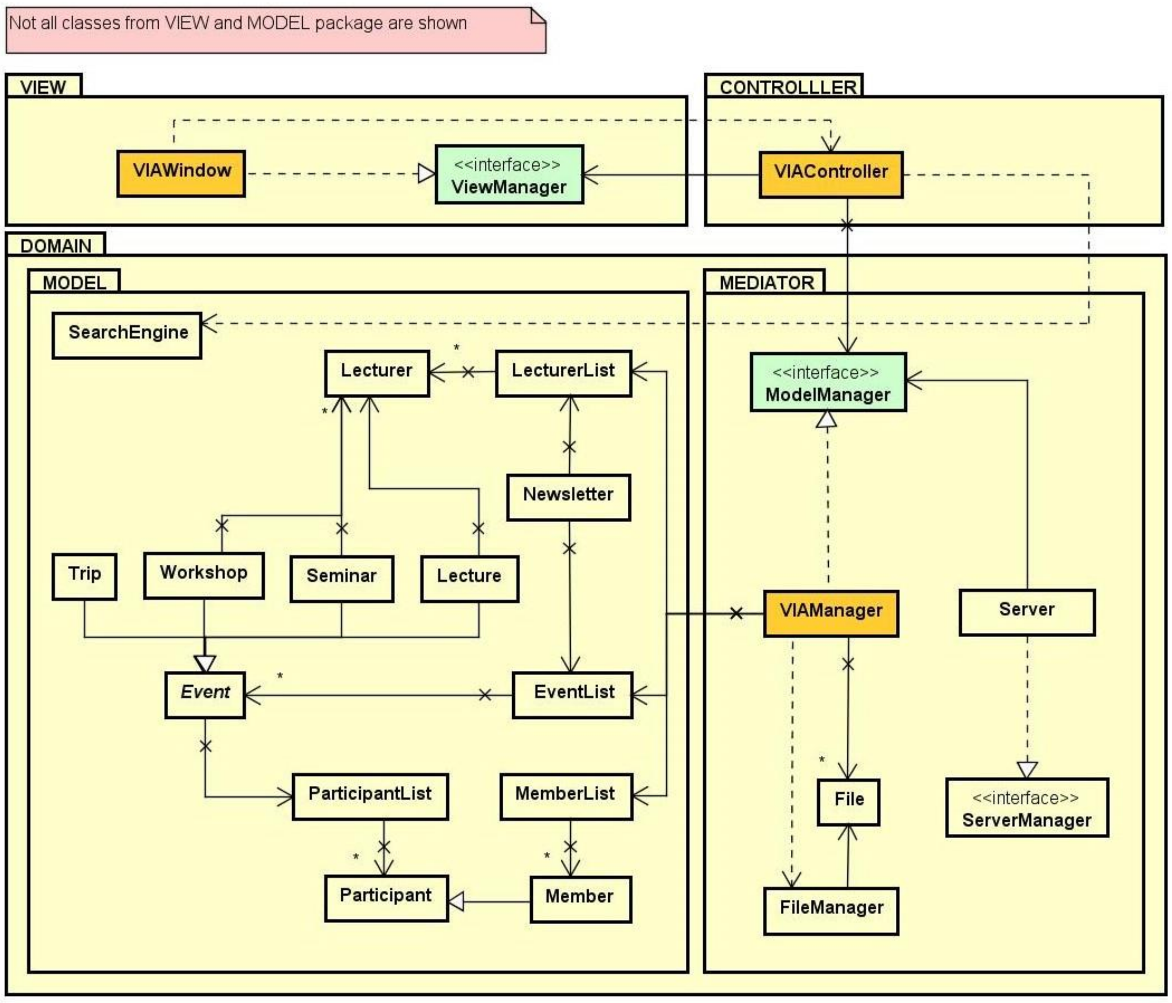
```java


import org.junit.jupiter.api.Test;

import controler.VIAController;

class SingletonTest {


    @Test
    void test() {
        VIAController c1=VIAController.getInstance();
        VIAController c2=VIAController.getInstance();
        assertEquals(c1, c2);
        assertEquals(true, c2 == c1);
    }
```

## 1.2. MVC

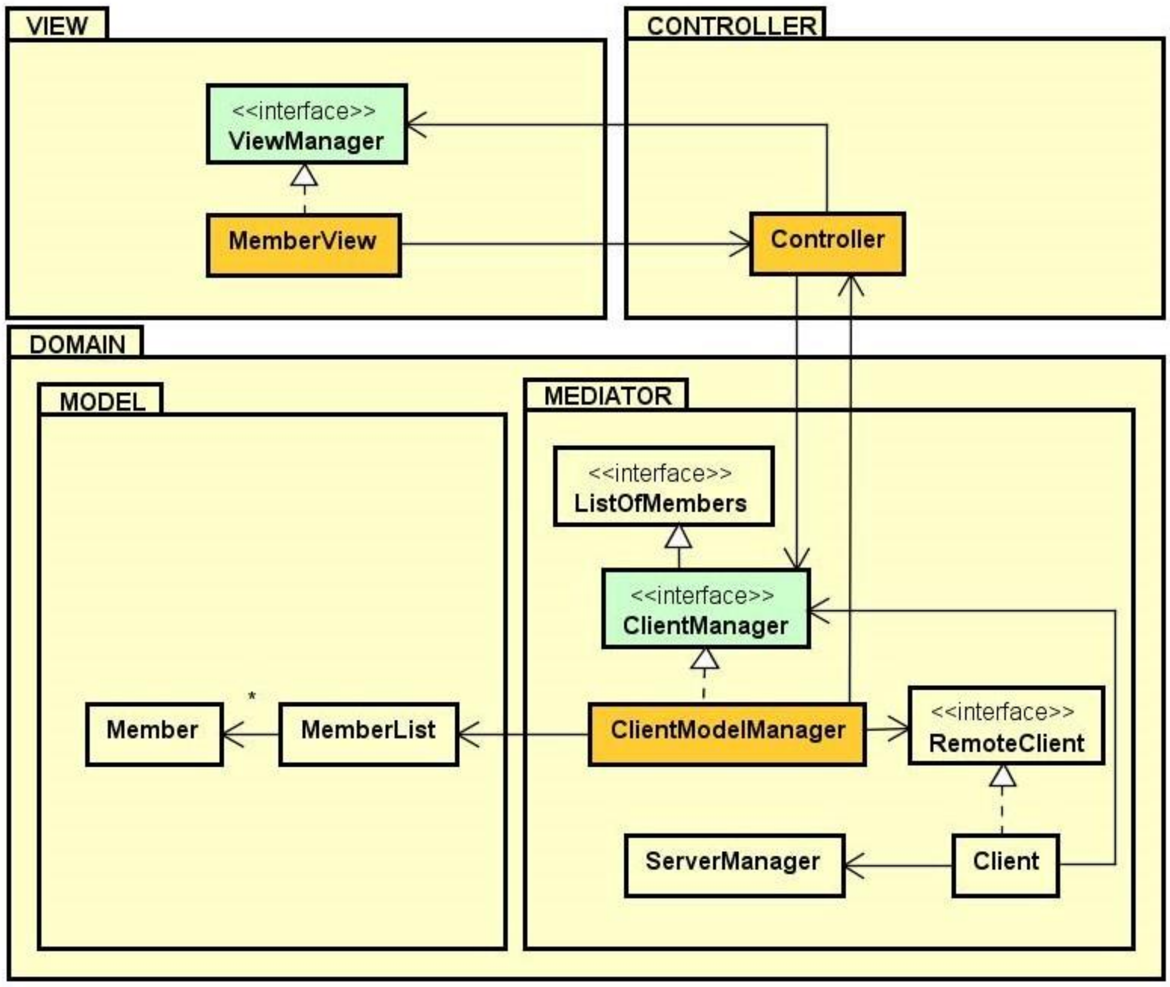
VIA management system is based on MVC architecture and design pattern. The main idea behind using this pattern is to divide responsibilities into three different parts of the system (on the server's side as well as on the client's side) :

1) Model – represents the business logic of VIA
2) View – responsible for showing data (GUI, console) requested from the Controller and for getting input from the user
3) Controller – controls the data flow from View to Model and vice versa

Class diagrams for server and client sides can be seen on the following diagrams.



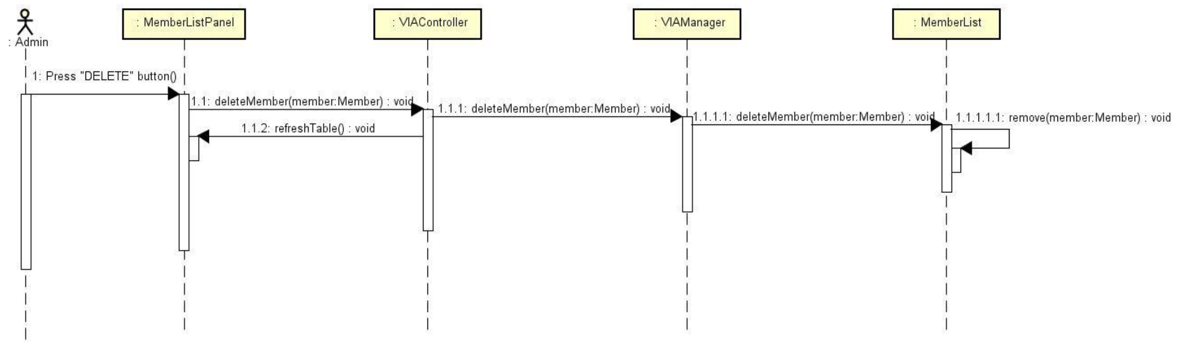
Server side of the system



Client side of the system

Moreover, the advantage of using the MVC design pattern is even clearer when a ViewManager implements an interface that holds all the methods needed for the view. That means that if the customer wants to change the view, then only the class implementing the view interface will be replaced and the system design will remain the same. Not only ViewManager implements the general interface, but also ModelManager implements it for the model and if there is some change in the business logic, then only the ModelManager class has to be replaced.

The sequence diagram for deleting members shows how MVC design pattern works.

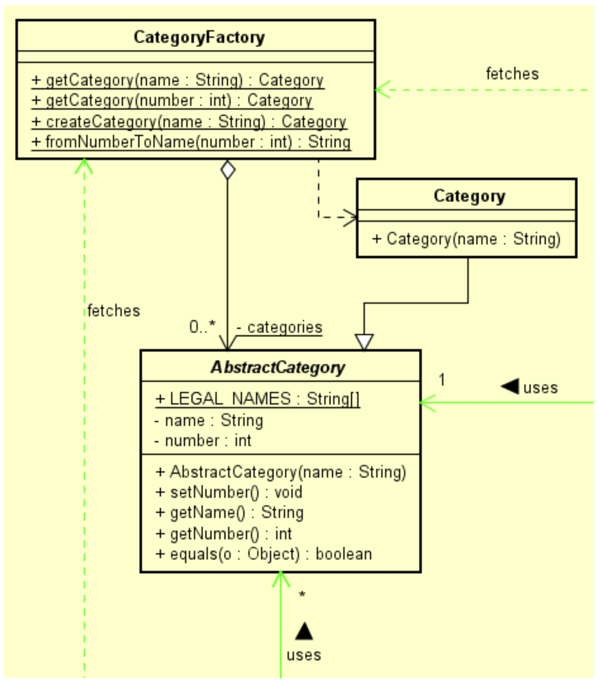

: Admin
: MemberListPanel
: VIAController
: VIAManager
: MemberList
1: Press "DELETE" button()
1.1: deleteMember(member:Member) : void
1.1.2: refreshTable() : void
1.1.1: deleteMember(member:Member) : void
1.1.1.1: deleteMember(member:Member) : void
1.1.1.1.1: remove(member:Member) : void

To sum up, MVC design pattern makes the system clear, easily maintainable and efficiently reusable.

## 1.3. Flyweight

The purpose of a flyweight is not to create a few identical objects, but share them instead. That is how the objects of the type category are supposed to be used. There is a limited list of categories to choose from and the categories should be the same no matter which object uses them (lecturer, lecture, workshop or seminar).

Looking from the design point of view a flyweight consists out of 3 classes: the CategoryFactory, the AbstractCategory and the actual Category. What the first one does is ensuring that the objects are shared properly. That is made in getters, where it searches for the requested flyweight and either returns it or creates a new one. However, the actual object is created in the Category class, which extends the AbstractCategory. The connections are shown on the diagram:


CategoryFactory
+ getCategory(name : String) : Category
+ getCategory(number : int) : Category
+ createCategory(name : String) : Category
+ fromNumberToName(number : int) : String
fetches
Category
+ Category(name : String)
fetches
0..*
- categories
AbstractCategory
+ LEGAL_NAMES : String[]
- name : String
- number : int
+ AbstractCategory(name : String)
+ setNumber() : void
+ getName() : String
+ getNumber() : int
+ equals(o : Object) : boolean
1
uses
*
uses

And the body of the methods shown in code:

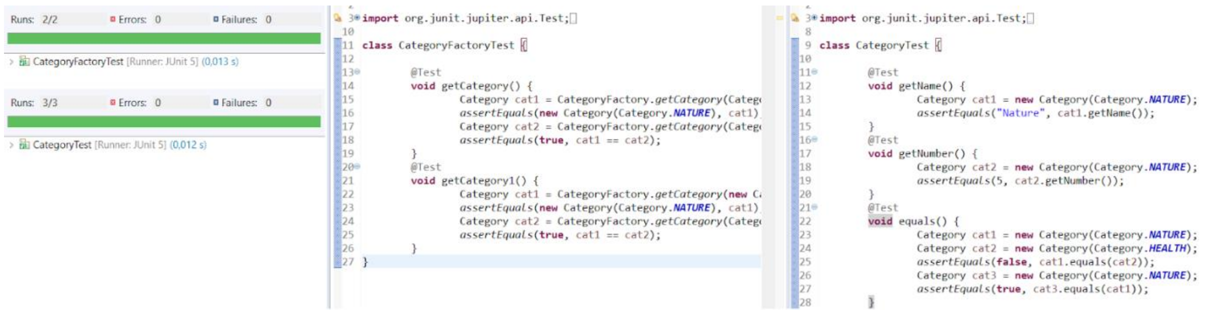
```
public static Category getCategory(String name) {
        name = Character.toUpperCase(name.charAt(0)) + name.substring(1).toLowerCase();
        return createCategory(name);
}

public static Category getCategory(int number){
        String name = fromNumberToName(number);
        return createCategory(name);
}

private static Category createCategory(String name){
        Category item = categories.get(name);
        if (item == null) {
                item = new Category(name);
                categories.put(name, item);
        }
        return item;
}
```

The underlined lines in the CategoryFactory ensure that the objects are shared properly, i.e. as mentioned before.

```
public abstract class AbstractCategory {
    public static final String ASTROLOGY = "Astrology";
    public static final String MEDITATION = "Meditation";
    public static final String REINCARNATION = "Reincarnation";
    public static final String HEALTH = "Health";
    public static final String BUDDHISM = "Buddhism";
    public static final String NATURE = "Nature";
    public static final String OTHER = "Other";

    private String name;
    private int number;

    public AbstractCategory(String name) {
        this.name=Character.toUpperCase(name.charAt(0))+ name.substring(1).toLowerCase();
        setNumber();
    }
```

The above code illustrates how the AbstractCategory creates the objects and limits the categories to choose from.

The Flyweight design pattern has been tested using JunitTest and proved to be working:



```
Runs: 2/2        Errors: 0        Failures: 0

> CategoryFactoryTest [Runner: JUnit 5] (0,013 s)

Runs: 3/3        Errors: 0        Failures: 0

> CategoryTest [Runner: JUnit 5] (0,012 s)
```

```
import org.junit.jupiter.api.Test;

class CategoryFactoryTest {

        @Test
        void getCategory() {
                Category cat1 = CategoryFactory.getCategory(Categ
                assertEquals(new Category(Category.NATURE), cat1)
                Category cat2 = CategoryFactory.getCategory(Categ
                assertEquals(true, cat1 == cat2);
        }
        @Test
        void getCategory1() {
                Category cat1 = CategoryFactory.getCategory(new C
                assertEquals(new Category(Category.NATURE), cat1)
                Category cat2 = CategoryFactory.getCategory(Categ
                assertEquals(true, cat1 == cat2);
        }
}
```

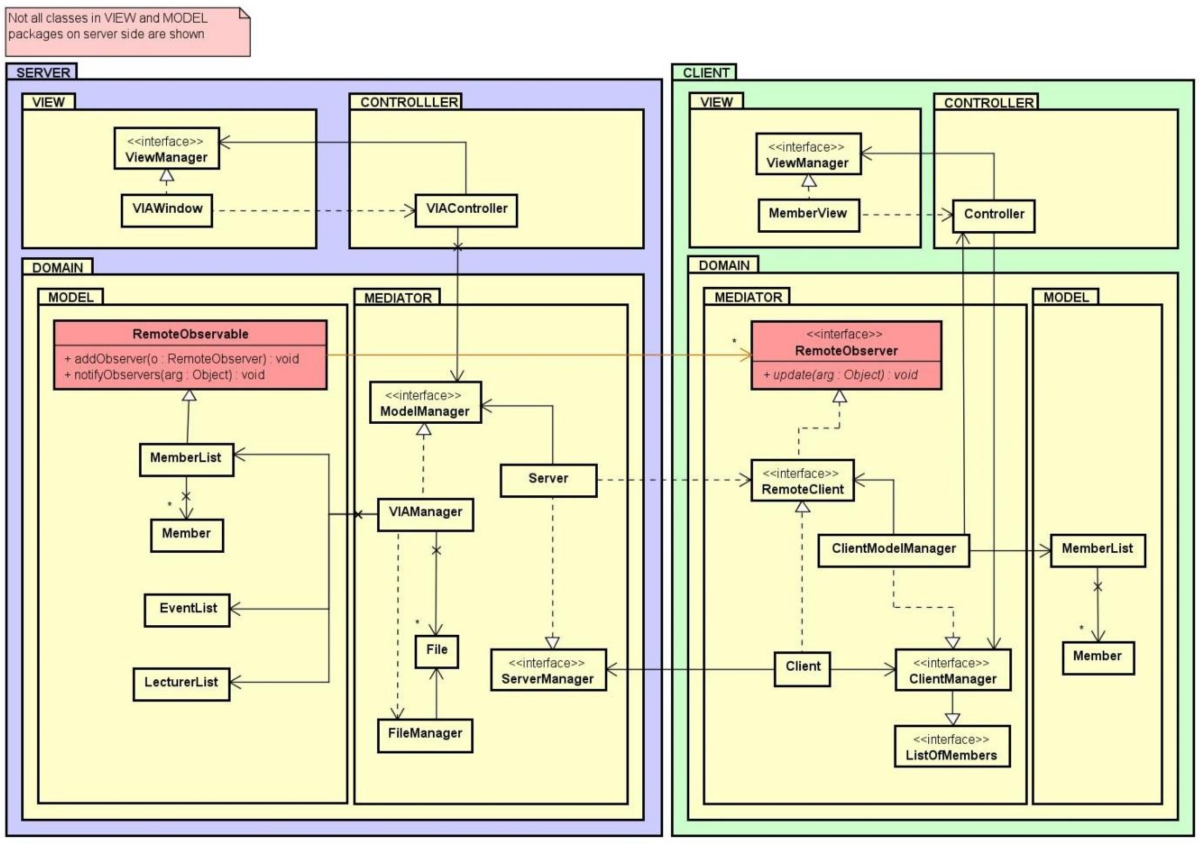
```
import org.junit.jupiter.api.Test;

class CategoryTest {

        @Test
        void getName() {
                Category cat1 = new Category(Category.NATURE);
                assertEquals("Nature", cat1.getName());
        }
        @Test
        void getNumber() {
                Category cat2 = new Category(Category.NATURE);
                assertEquals(5, cat2.getNumber());
        }
        @Test
        void equals() {
                Category cat1 = new Category(Category.NATURE);
                Category cat2 = new Category(Category.HEALTH);
                assertEquals(false, cat1.equals(cat2));
                Category cat3 = new Category(Category.NATURE);
                assertEquals(true, cat3.equals(cat1));
        }
```

## 1.4. Remote observer

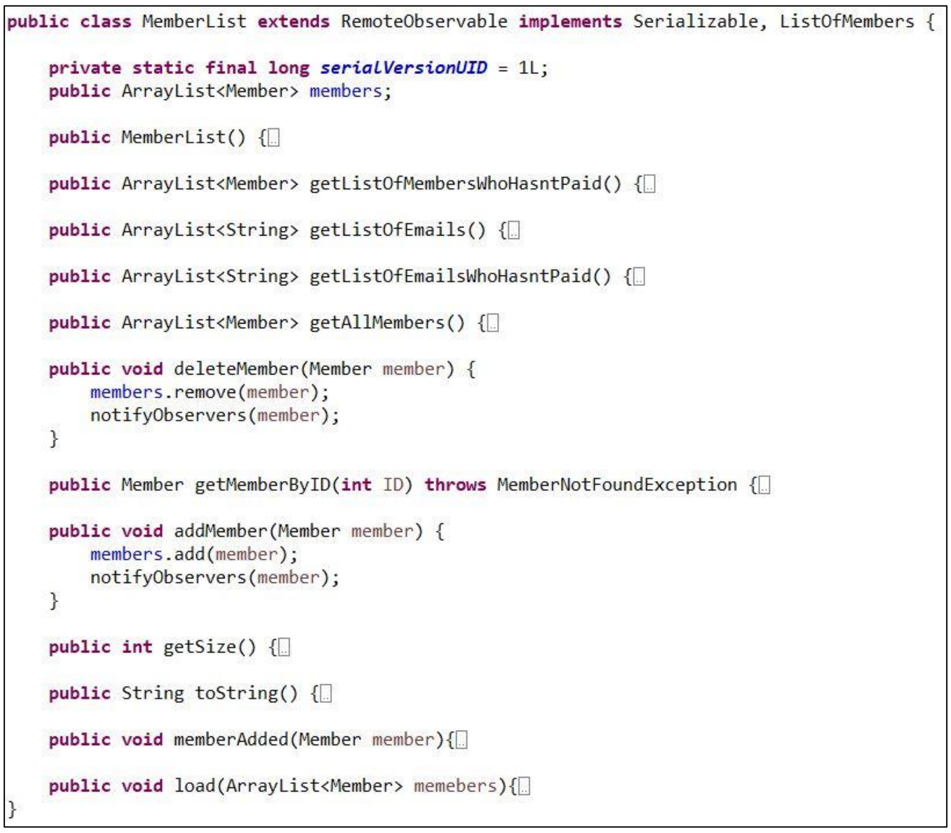
Another design pattern implemented in VIA Management system is the Remote observer design pattern. The main reason for using this pattern is the need of an automatic update of some classes (observers) every time a specific class (observable) is changed.

In this system, MemberList class (observable) on the server side is observed by the Client class (observer) on the client side in the following way:

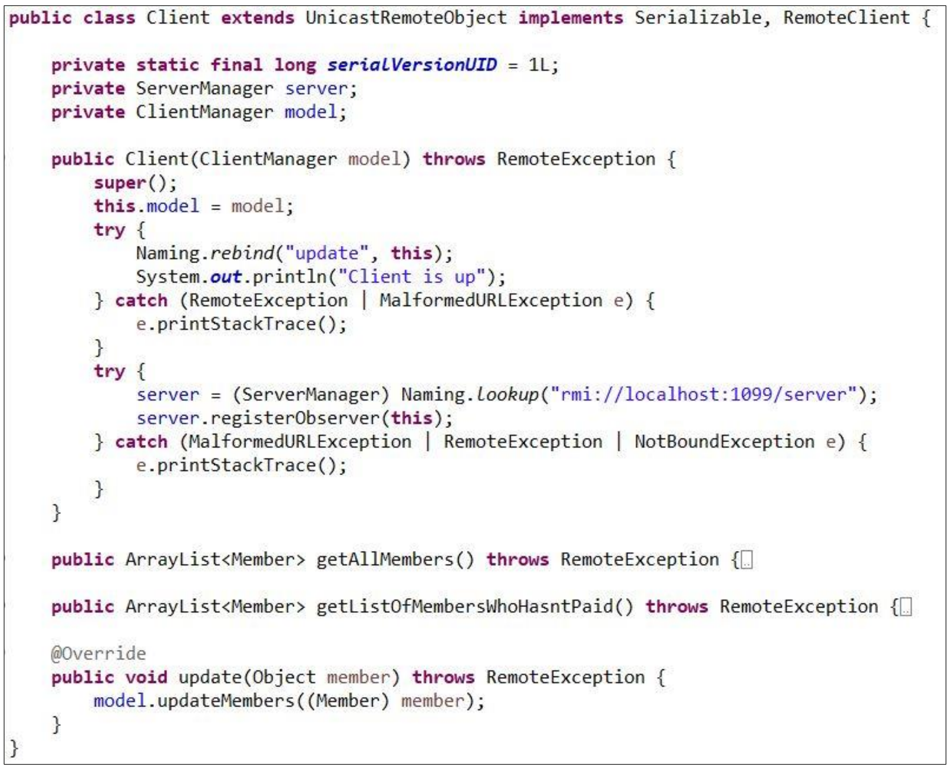
1) Not only the server, but also the client makes itself as a stub object (due to issue with managing MemberList, calling update method on client using RMI).
2) The client establishes a connection with the server and adds itself as an observer
3) If there is a change in the server model, specifically in the MemberList, the observer pattern calls on each Client class an "update" method that updates the MemberList on the client side.


Not all classes in VIEW and MODEL packages on server side are shown
SERVER
VIEW
<<interface>> ViewManager
VIAWindow
CONTROLLLER
VIAController
DOMAIN
MODEL
RemoteObservable
+ addObserver(o : RemoteObserver) : void
+ notifyObservers(arg : Object) : void
MemberList
Member
EventList
LecturerList
MEDIATOR
<<interface>> ModelManager
Server
VIAManager
File
FileManager
<<interface>> ServerManager
CLIENT
VIEW
<<interface>> ViewManager
MemberView
CONTROLLER
Controller
DOMAIN
MEDIATOR
<<interface>> RemoteObserver
+ update(arg : Object) : void
<<interface>> RemoteClient
ClientModelManager
Client
<<interface>> ClientManager
<<interface>> ListOfMembers
MODEL
MemberList
Member

Shown in Java:

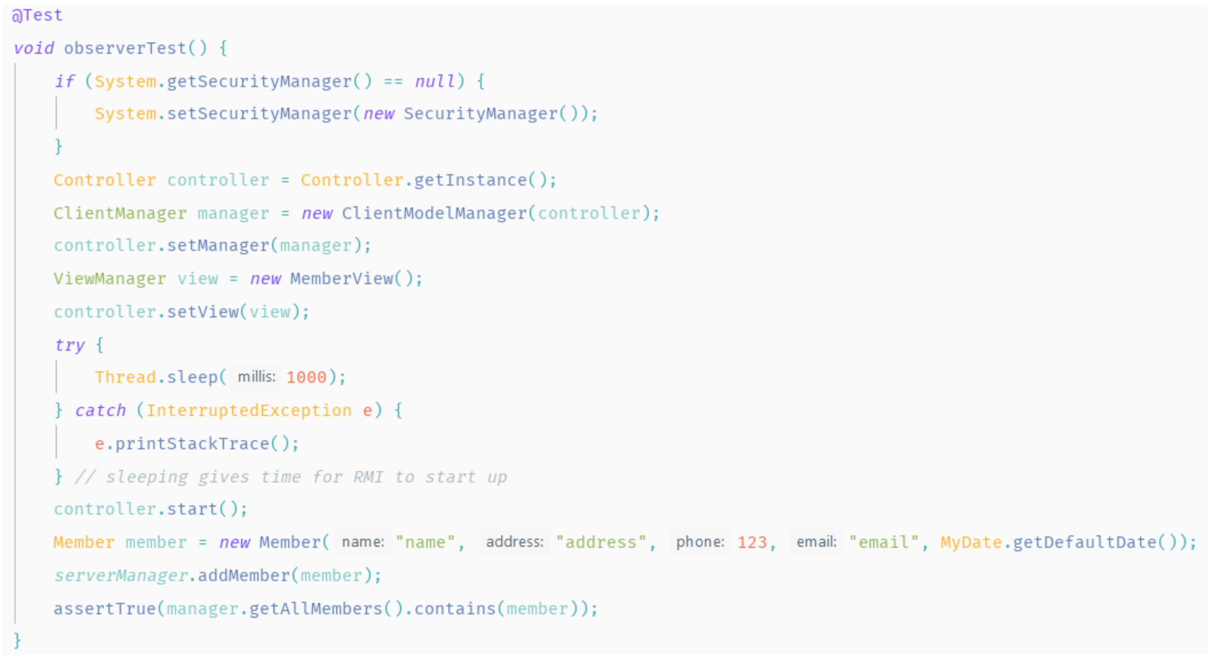


```java
public class MemberList extends RemoteObservable implements Serializable, ListOfMembers {

    private static final long serialVersionUID = 1L;
    public ArrayList<Member> members;

    public MemberList() {□

    public ArrayList<Member> getListOfMembersWhoHasntPaid() {□

    public ArrayList<String> getListOfEmails() {□

    public ArrayList<String> getListOfEmailsWhoHasntPaid() {□

    public ArrayList<Member> getAllMembers() {□

    public void deleteMember(Member member) {
        members.remove(member);
        notifyObservers(member);
    }

    public Member getMemberByID(int ID) throws MemberNotFoundException {□

    public void addMember(Member member) {
        members.add(member);
        notifyObservers(member);
    }

    public int getSize() {□

    public String toString() {□

    public void memberAdded(Member member){□

    public void load(ArrayList<Member> memebers){□
}
```

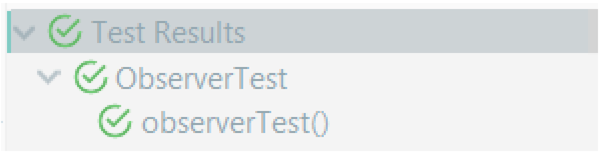


```java
public class Client extends UnicastRemoteObject implements Serializable, RemoteClient {

    private static final long serialVersionUID = 1L;
    private ServerManager server;
    private ClientManager model;

    public Client(ClientManager model) throws RemoteException {
        super();
        this.model = model;
        try {
            Naming.rebind("update", this);
            System.out.println("Client is up");
        } catch (RemoteException | MalformedURLException e) {
            e.printStackTrace();
        }
        try {
            server = (ServerManager) Naming.lookup("rmi://localhost:1099/server");
            server.registerObserver(this);
        } catch (MalformedURLException | RemoteException | NotBoundException e) {
            e.printStackTrace();
        }
    }

    public ArrayList<Member> getAllMembers() throws RemoteException {□

    public ArrayList<Member> getListOfMembersWhoHasntPaid() throws RemoteException {□

    @Override
    public void update(Object member) throws RemoteException {
        model.updateMembers((Member) member);
    }
}
```

Observer pattern has been tested with unit tests and proven working.



```
@Test
void observerTest() {
    if (System.getSecurityManager() == null) {
        System.setSecurityManager(new SecurityManager());
    }
    Controller controller = Controller.getInstance();
    ClientManager manager = new ClientModelManager(controller);
    controller.setManager(manager);
    ViewManager view = new MemberView();
    controller.setView(view);
    try {
        Thread.sleep( millis: 1000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    } // sleeping gives time for RMI to start up
    controller.start();
    Member member = new Member( name: "name", address: "address", phone: 123, email: "email", MyDate.getDefaultDate());
    serverManager.addMember(member);
    assertTrue(manager.getAllMembers().contains(member));
}
```



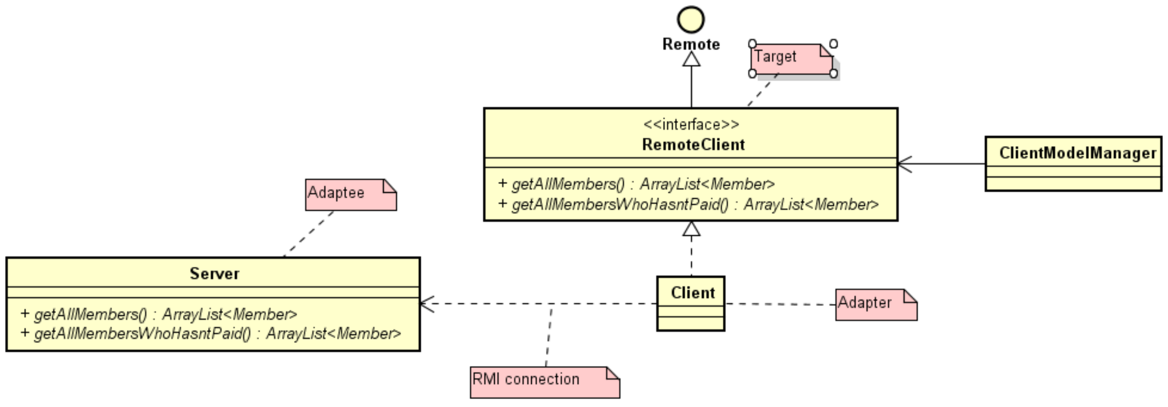
```
Test Results
  ObserverTest
    observerTest()
```

## 1.5. Adapter

One of the many advantages of using the RMI connection is a very easy implementation of the Adapter pattern. The purpose of this design pattern is to adapt one part of the system to another part that is not compatible with it. This case can be seen on the Client side of the system, where the source of data to the application is provided by RMI. RemoteClient interface is the target for the adapter class - Client.

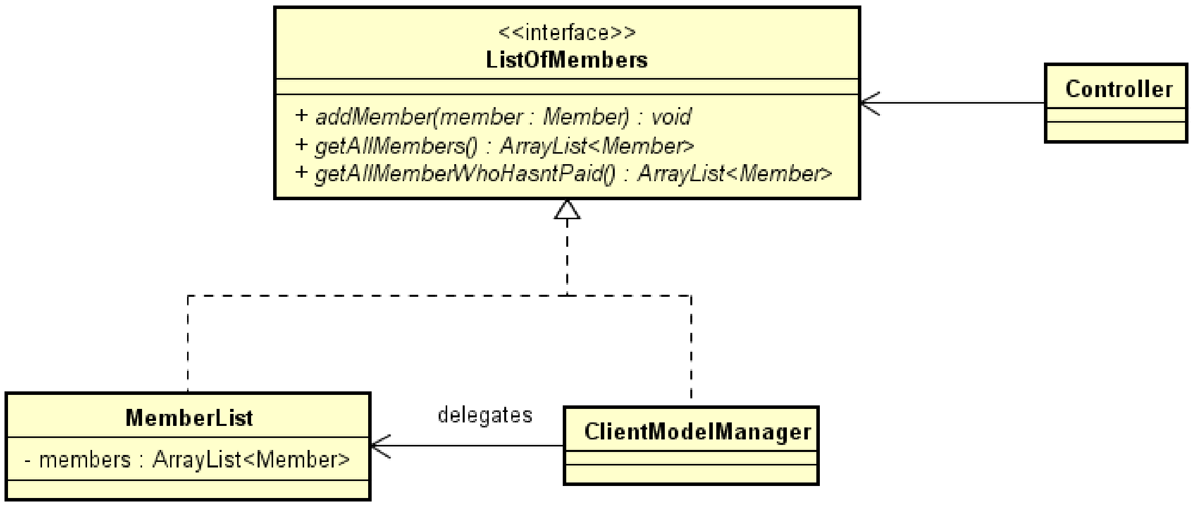
Moreover the RemoteClient is also extending the Remote interface due to the use of RMI. The Adaptee class is the Server on the server side of the system. It is responsible for registering methods to use by the client. Thanks to this, the ClientModelManager does not have to handle the communication with the server and also marshaling and unmarshaling objects.

In addition, thus using Adapter pattern, Dependency Inversion SOLID rule is preserved. ClientModelManager is not dependent on the Client class, but it depends on an abstraction in the form of the RemoteClient interface.


Remote
Target
<<interface>>
RemoteClient
+ getAllMembers() : ArrayList<Member>
+ getAllMembersWhoHasntPaid() : ArrayList<Member>
ClientModelManager
Adaptee
Server
+ getAllMembers() : ArrayList<Member>
+ getAllMembersWhoHasntPaid() : ArrayList<Member>
Client
Adapter
RMI connection

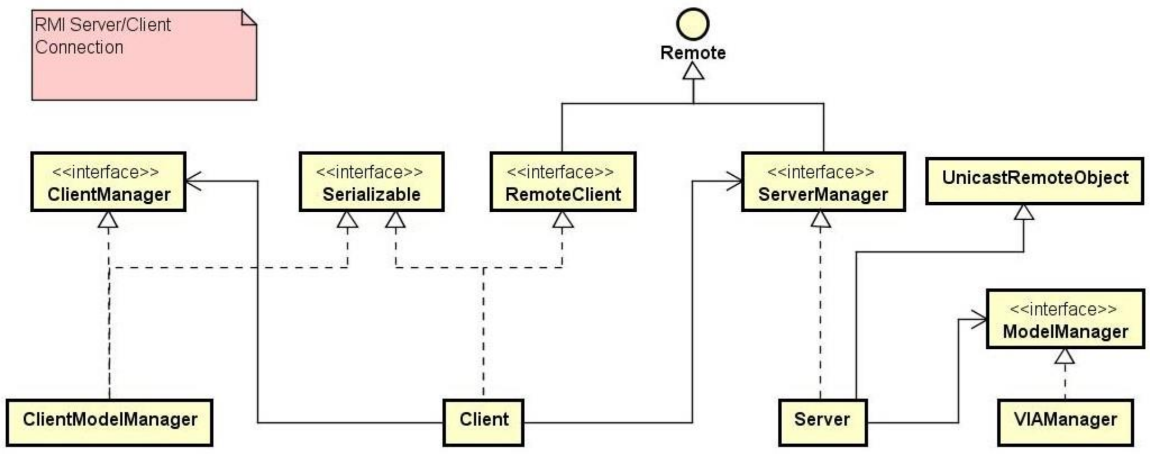
## 1.6. Proxy

The proxy design pattern is very useful in terms of organizing code and the project structure. The main goal is to create an easy access point to a part of the system. In this case, ClientModelManager is the proxy class between the Controller and the MemberList (both on the client side). Both ClientModelManager and MemberList are implementing the ListOfMembers interface. The controller invokes a method in the ClientModelManager, which is delegating it to the MemberList. Using the ClientModelManager is implied by the MVC pattern and allows it to expand with new functionalities, making it a Facade to the whole model.


<<interface>>
ListOfMembers
+ addMember(member : Member) : void
+ getAllMembers() : ArrayList<Member>
+ getAllMemberWhoHasntPaid() : ArrayList<Member>
Controller
MemberList
- members : ArrayList<Member>
delegates
ClientModelManager

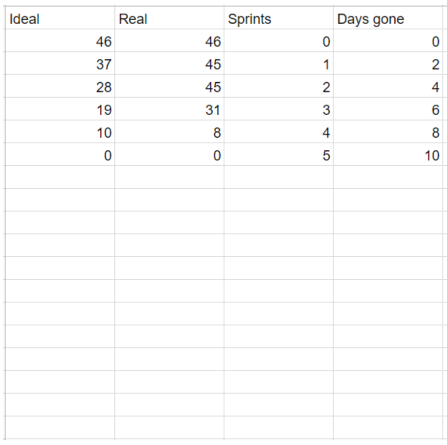
## 2. RMI client-server connection

For client/server connection the Remote Method Invocation (RMI) has been chosen in order to make the code clear and readable. The server extends the UnicastRemoteObject and it needs an interface which extends Remote interface to identify which methods can be invoked remotely. Then Stub has been uploaded to Registry. A stub for remote object acts as client's local representative or proxy for the remote object.  In the client side Serializable is implemented and "an object from interface" is needed. Interface object communicates with remote Implementation object via Stub.

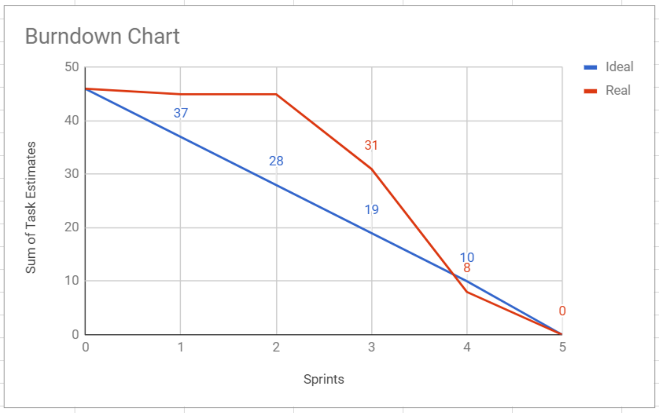

RMI Server/Client Connection
Remote
<<interface>> ClientManager
<<interface>> Serializable
<<interface>> RemoteClient
<<interface>> ServerManager
UnicastRemoteObject
<<interface>> ModelManager
ClientModelManager
Client
Server
VIAManager

# 3. SCRUM

The course assignment was developed using SCRUM. Due to having the duration of accomplishing the assignment established for two weeks, sprints have been decided to be 2 days long. That means that there were 5 sprints in total. Prior to each sprint there was a sprint planning meeting where the tasks for the next two days were being selected and a sprint backlog was created. Moreover, each sprint was followed by a sprint review meeting, during which the work done by each member was being discussed. For organising and keeping track of the tasks, Trello has been used. Afterwards, the tasks from the "to do" and "in progress" lists have been being copied to a sprint backlog, which was a text document and the points assigned to the tasks moved to the "done" list have been added to the burndown chart.



| Ideal | Real | Sprints | Days gone |
|---|---|---|---|
| 46 | 46 | 0 | 0 |
| 37 | 45 | 1 | 2 |
| 28 | 45 | 2 | 4 |
| 19 | 31 | 3 | 6 |
| 10 | 8 | 4 | 8 |
| 0 | 0 | 5 | 10 |



During the assignment, the members have learnt that not only each backlog story should have points assigned, but also each task composing the story. As a result of having points only assigned to a whole story, the burndown chart couldn't be updated until the whole story was accomplished and the progress couldn't be seen properly.

## Appendices

Appendix 1 – Source code – https://github.com/11michi11/SEP1/tree/master/SDJ/SDJ1/src