**SDJ2 Assignment of Group 4**

**Taha Mohamed Alzain(2669055)**

**Muhammad Nadeem 266704**

**Fadi Atia Dasus(266265)**

**Oskars Arajs(266534)**

**Alexandru Vieru(267013)**

**Supervisor**

Martin Zmija

**ICT Engineering, VIA University College, Horsens**

**2nd Semester**

Table of Contents

[1 Introduction 4](#_Toc513101604)

[2 Requirements 4](#_Toc513101605)

[2.1 Functional requirements 4](#_Toc513101606)

[2.2 Nonfunctional requirements 4](#_Toc513101607)

[3 Time schedules 5](#_Toc513101608)

[3.1 Burndown Chart 6](#_Toc513101609)

[4 Class Diagram 7](#_Toc513101610)

[5 Implementation 7](#_Toc513101611)

[5.1 Design patterns 7](#_Toc513101612)

[5.2 RMI 9](#_Toc513101613)

[6 Testing 10](#_Toc513101614)

References

**Abstract**

This document’s aim is to inform the reader about the different methods and techniques used to fulfill the main objectives for this project. For this case the objectives solved where linked to the business case given by Vipassana which were struggling with their daily activities given that they had limited technology in use.

As a result, different technical choices were made to produce a satisfactory outcome for the given report. This document will proceed with a small description of the current client and the struggles they face in their organization.

Based on an interview taken at the costumers headquarters a set of requirements have been subtracted and are crucial to get a better understanding of the client’s needs.

# Introduction

Vipassana is a spiritual center that focuses on Buddhist spiritual practices and insight awareness.

The company does not practice any religious beliefs but rather focus their attention on dream interpretation and healing practices. They are responsible for organizing different types of events and adding new members as they sign up. Vipassana is also holding lectures about certain topics where members can also register for.

The current system of keeping track of all the mentioned events is slow and inefficient to say the least, relying on pen and paper to execute and it lacks reliability since sometimes employees misplace certain records and forget where they stored them originally.

As up lately some members have not been paying their membership fee which is causing a financial deficit for the company. For that reason, Vipassana would like a solution for this problem which can inform them which member has not played their fees.

The client mentioned in the interview case that they would like it if this project will focus on a client server approach since this would allow users to see directly and efficient the list of their debtors.

The project planning method used for this report is SCRUM and in the following section the project backlog or requirements shall be presented.

# Requirements

Based on the interview case with Vipassana, a set of requirements have been taken out and are split in two categories: functional and nonfunctional.

As mentioned earlier these requirements are crucial for this step since they provide a good understanding of the costumer’s needs and will help in the development of this assignment.

## Functional requirements

* The system must allow the employee to list all members.
* The system must allow employee to list members that have not paid their membership fee.

## Nonfunctional requirements

* The system must provide a client server connection.
* At least 4 java design patterns must be implemented.

# Time schedules

For this report the managing method chosen is called SCRUM and it provides an agile way of organizing the work load.

The work for this report has been conducted in a team of 5 people and as such tasks had to be split accordingly. For that reason and the fact that there was a 2-week deadline to complete, a time schedule has been created to organize the activities in an efficient manner.

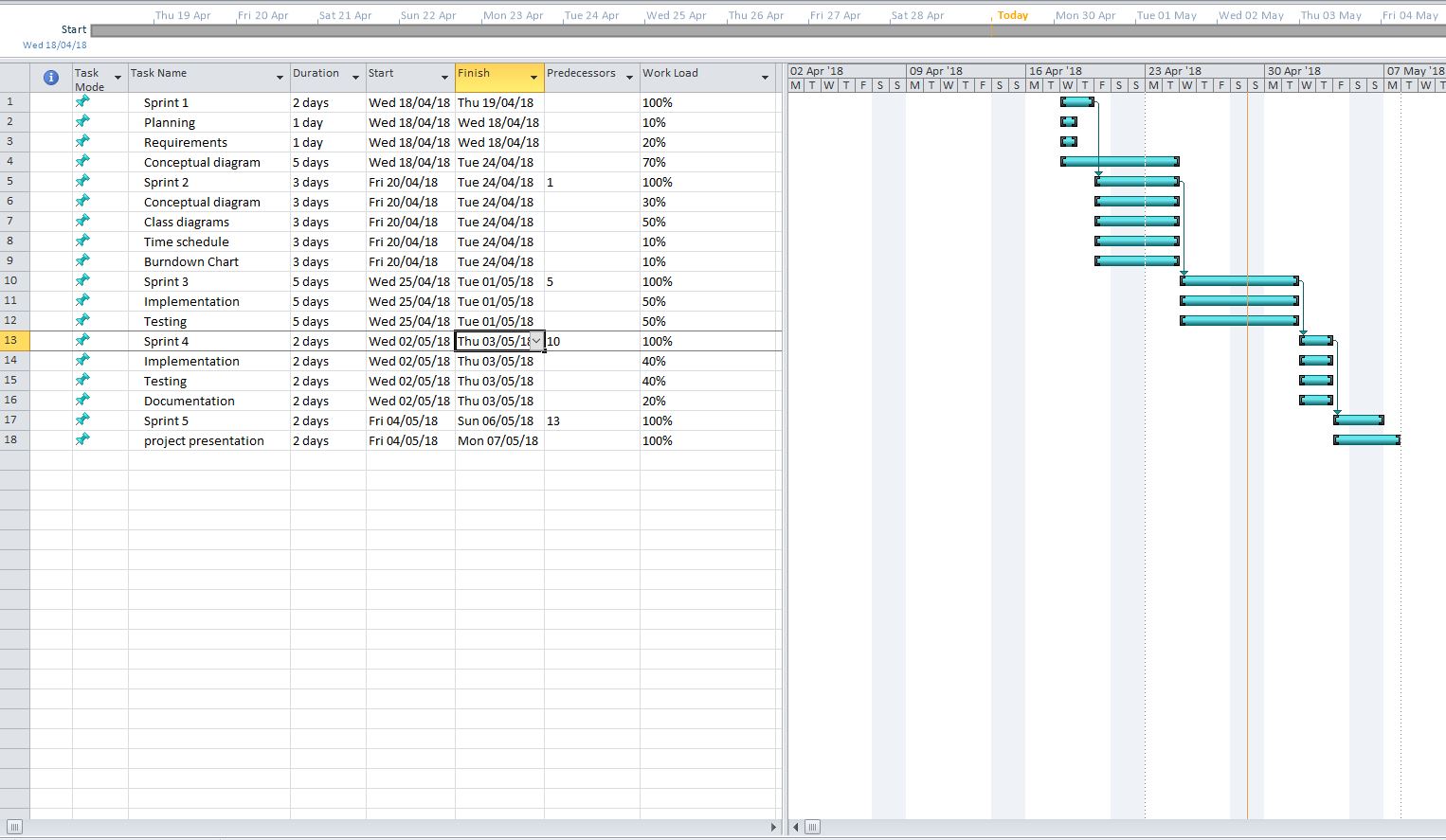
The assignment started on the 18th of April 2018 and ended on 4th of May 2018.

Due to such a short time period 4 sprints where created and instead of just having one for each week the group had 2 every Wednesday and Friday for the given time. A fifth sprint has also been created but it only serves as a meeting where the presentation for this report will be made and is not affected by the deadline.

Each of the 4 sprints have been given several requirements to complete, some being higher priority than others thus having a bigger percentage of time allocated for their completion.

A record of these sprints has been taken and is presented in form of a Gantt Chart. The Chart was an essential tool that helped organize and keep track of each task of for the given assignment.

***Figure 1 Gantt Chart for SDJ Assignment 1***



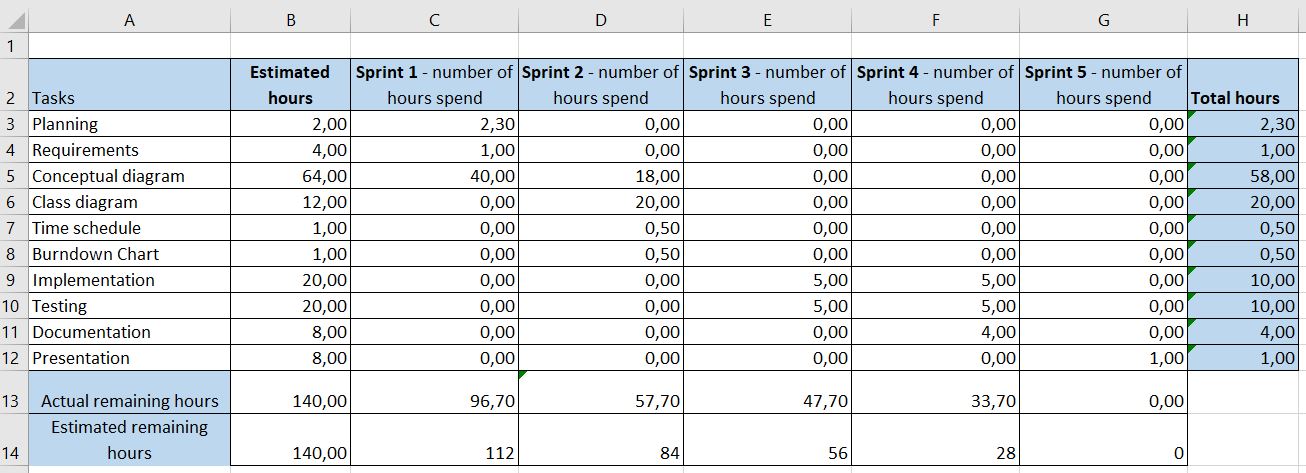
## Burndown Chart

In this section of the chapter the Burndown Chart shall be discussed. This tool is essential to keep track of the schedule established earlier and informs the team if it is behind or ahead schedule.

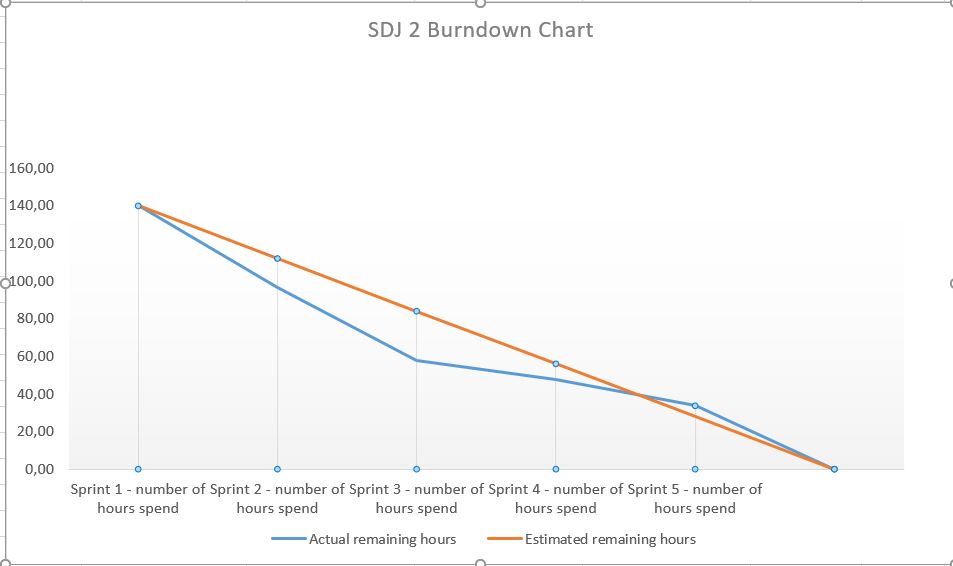
The chart is based on each task in each sprint for 5 sprints in total. It has two main components which consist of estimated time given for an individual task and an actual time spent for it.

This helped the team to reorganize the time allocated for this project in case it fell behind on certain activities or plan forward if it was ahead.

***Figure 2 Burndown Chart Table***

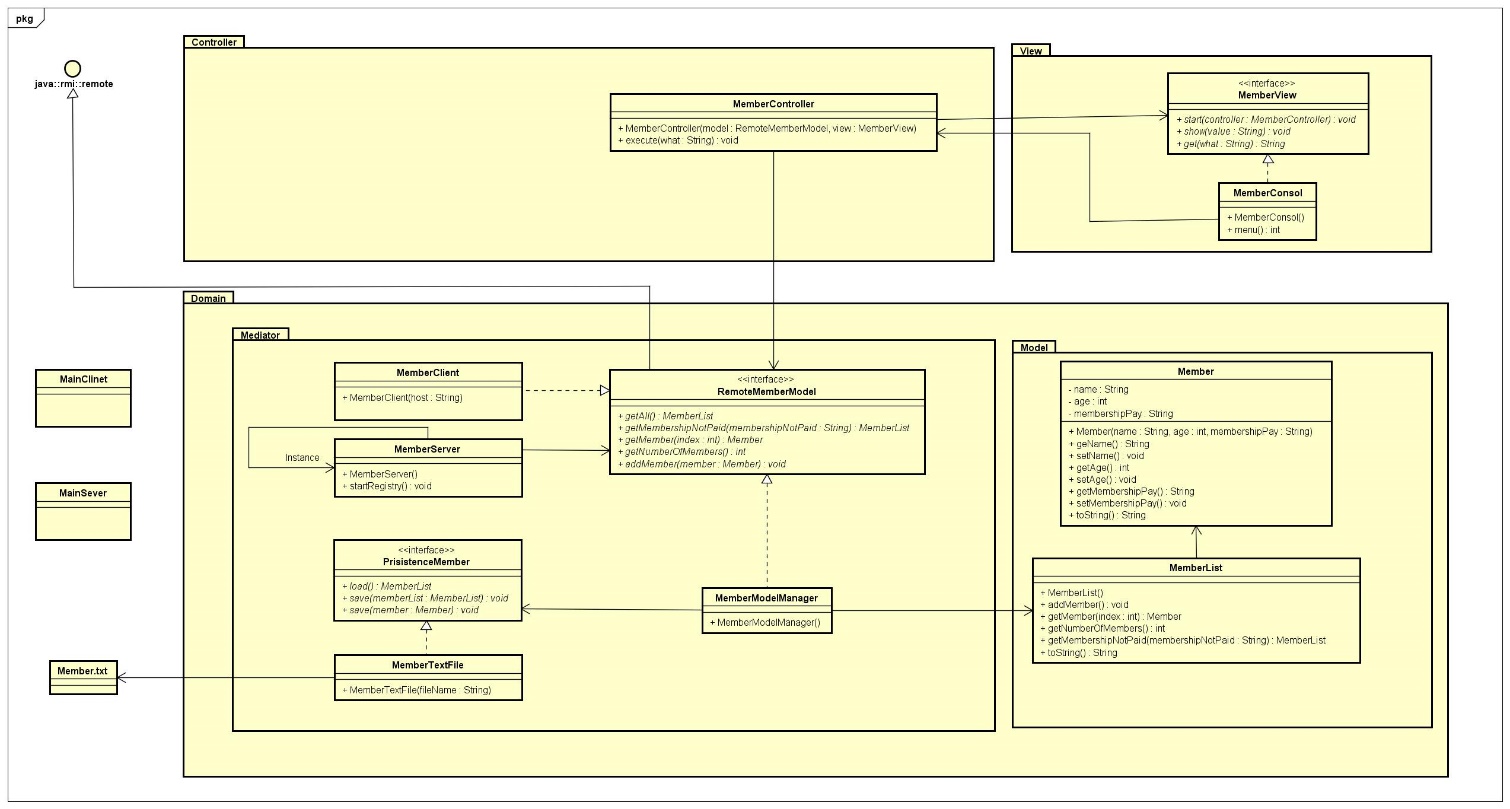


***Figure 3 Burndown Chart***



# Class diagram

***Figure 4 Class Diagram***



Class diagram in figure 4 is consist of 3 main packages Domain, View and Controller, for simplicity of coding domain package is divided into two more packages Model and Mediator. The division of these packages is called the design pattern i.e. MVC. Explanations of all the classes with methods would be in next paragraphs.

**MemberClass**

This class provides a constructor which creates member objects. This class provides the following methods: getName, getAge, getMembershipPay and etc.

MemberListClass**:**

This class provides methods an ArrayList to which multiple members can be added. In addition this class provides methods for adding member, getMembershipNotPaid, getNumberOfMembers and etc.

**MemberModelManeger**

This class is responsible for initializing the load() method to retrieve information regarding members.

**RemoteMemberModel**

This interface contains the following methods: getAll(), getNumberOfMembers(),getMembershipNotPaid(), getMember(), addMember()

**PersistanceMember**

This interface gathers information from the user and stores that information in a ArrayList and stores that information on a .txt file.

**MemberTextFile**

This class is responsible for creating a .txt file which holds information regarding created member objects. It has a method load(), with which the user can retrieve member information from the .txt file.

**MemberServer**

This class is responsible for creating a server on the PC on which the source code was started on. It has a non-argument constructer that has a try-catch clause to avoid exceptions when starting the server. It has override methods from the MemberList class: getAll(), getMamebershipNotPaid().

**MemberClient**

This class is used to connect to the server. It uses a non-argument constructer and a try- catch clause to avoid exception when running the source code. It has some inherited methods like: getAll(), getNumberOfMembers(),getMembershipNotPaid(), getMember(), addMember();

**MemberConsol**

This class provides methods that start when the server has started.

**MemberView**

This interface contains the following methods: start() and show();

**MemberController**

This class provides instruction on what happens when a certain input is received.

It has a method execute() which uses cases to help guide the user through the system’s interface.

# Implementation

In this chapter the discussion will be on the design patters to make the client server application. RMI is used to connect the clients with the server. First design patterns are explained then RMI,

## Design patterns

For this report the following design patters have been chosen:

* MVC
* Singleton
* Façade
* Adapter

***MVC* or model view controller**

MVC design pattern, which divides the application into three interconnected layers these are Model, view, and controller.

*Model*represents the actual data that exists, and the different rules used for its interaction.

*View* shows the content of the model and tells how the data should be shown.

*Controller* creates a link between the user and the view in which the model executes the required operations.

MVC is somehow explained in class diagram (figure 4) but more details should be presented here,

***Model*** is consisted of with Member and Member list class. To make the connection between the MemberList and MemberModelManager the object is created in MemberModelManager. To access the data from the text file object is created from MemberAdopter in the MemberModelManger. We also have MemberServer and MemberClient in Model but will be explained later in RMI.

***View*** is consisted of MemberView and MemberConsole. MemberConsole is Implemented on the base of MemberView so all methods are implemented and overridden. We have the Menu method in console that is the representation of how our console will look and work. i.e.

**private** **int** menu()

{

System.***out***.println("Welcome to Member Application");

System.***out***.println("--------------");

System.***out***.println("1) List all Member's");

System.***out***.println("2) Search Members by Membership not Paid");

System.***out***.println("3) Quit");

System.***out***.println();

System.***out***.print("Select an item by pressing number 1-3: ");

**int** selection = in.nextInt();

in.nextLine();

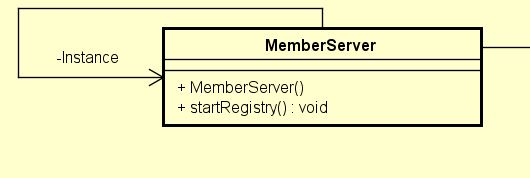
**return** selection;

}

***Controller*** is used to link between model and view. The connection is made by creating the objects from the MemberView and RemoteMemberModel.

**Singleton Design Pattern**  is a design pattern which ensure that a java class has only one instance and provides a global access for it. In figure 5 MemberServer is a singleton. So, it has a global access and all clients can access the server but server will have only one instance.

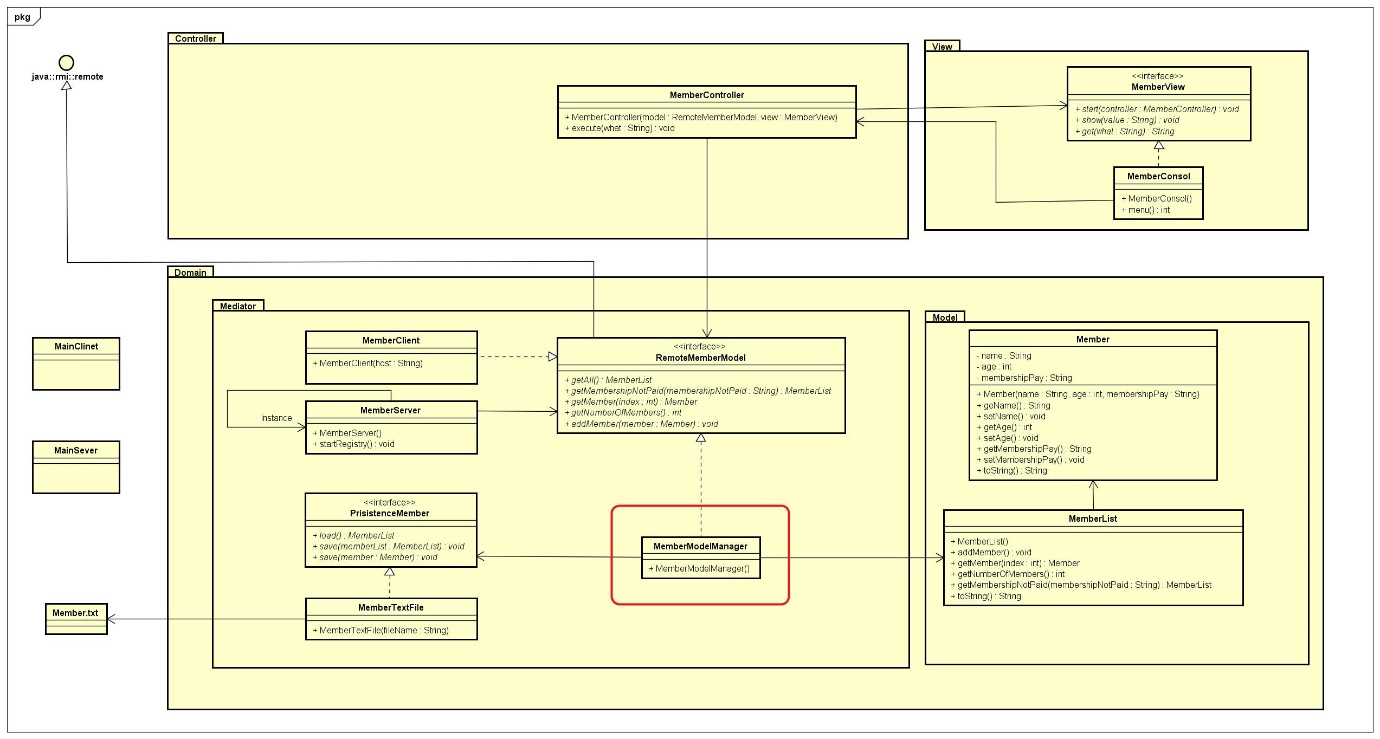
***Figure 5 Singleton***



**Façade** is another design pattern that was chosen for this project. The façade design pattern is a wrapper interface that connects multiple interface together to help the client’s application interact with the system.

In figure 6 with red tringle, MemberModelManager is the façade that connects to MemberList and PersistenceMember. It also has the connection with MemberContoller through RemoteMemberModel interface.

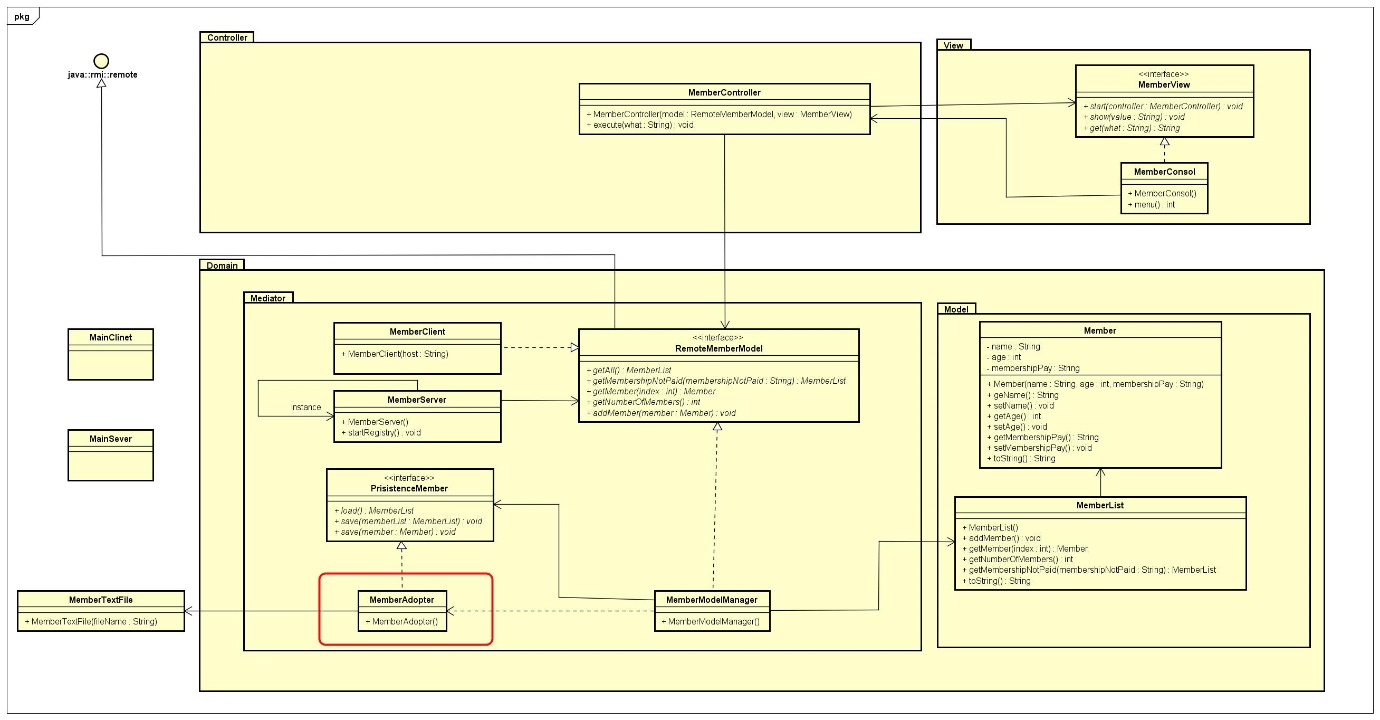
***Figure 6 Facade***



**Adapter** is the final design pattern used in this report. The adapter helps in the connection of two separate interfaces which without it would not be able to connect otherwise due to incompatibility issues.

In figure 7, class within red line triangle is an adopter design pattern i.e. MemberAdopter. AS there are two separate interface PersistenceMember and RemoteMemberModel. So MemberAdopter is created to connect the MemberModelManager to the MemberTextFile.

***Figure 7 Adopter***



## 

## RMI

RMI (Liang 2009) is the Java Distributed Object Model for facilitating communications among distributed objects. RMI not only enables to pass data among objects on many systems, but also to invoke methods from a remote object. Following points needed to work for RMI as : (1) A server is registered with the RMI registry; (2) A client works through the RMI registry for the remote object; (3) when the remote object is located, stub is returned the client; (4) The remote object could be used in the same way as a local object. The communication between the server and the client is through the stub and skeleton.

MemberServer and MemberClient are works as RMI for facilitate communication. In MemberServer port (1099) is created to register the object and rebind it with the name of “MemberSever”.

**public** **class** MemberServer **extends** UnicastRemoteObject **implements** RemoteMemberModel {

**private** RemoteMemberModel memberList;

**private** **static** MemberServer *instance*;

**public** MemberServer() **throws** RemoteException {

**try** {

Registry reg = LocateRegistry.*createRegistry*(1099);

Naming.*rebind*("MemberServer", **this**);

} **catch** (MalformedURLException e) {

e.printStackTrace();

}

memberList = **new** MemberModelManager();

System.***out***.println("Starting server...");}

MemberClient looks up the server through local host as

"rmi://localhost:1099/MemberServer".

**public** **class** MemberClient **implements** RemoteMemberModel {

**private** RemoteMemberModel server;

**public** MemberClient() {

**try** {

server = (RemoteMemberModel) Naming.*lookup*("rmi://localhost:1099/MemberServer");

} **catch** (MalformedURLException e) {

e.printStackTrace();

} **catch** (RemoteException e) {

e.printStackTrace();

} **catch** (NotBoundException e) {

e.printStackTrace();

}

}

So the connection is established between server and client.

Main Methods are written in MainServer and MainClient. Where we create the object from the Model, View and controller then view start the controller.

# Testing

In this chapter there will be a discussion about the way the code has been tested and what method was used to perform such a task. The method used to test the code is called j unit testing and has been implemented for the current code.

Testing is the documentation of the result to make sure it fulfils all the requirements. The system functionality has been tested on the base of the requirements. In Model package, Member and MemberList has been tested with unit root test and most of the classes have same methods as in Model. To check the functionality all methods have been tested by Junit test, but discussion is here on Get Members who did not paid membership fees. Test code should be as

**public** **class** MemberListTest {

Member member1 = **new** Member("Taha", 26, "paid");

Member member2 = **new** Member("Alex", 24, "not paid");

Member member3 = **new** Member("Oskar", 22, "not paid");

MemberList list = **new** MemberList();@Test

@Test

**public** **void** testGetsMembershipNotPaid() {

list.addMember(member1);

list.addMember(member2);

list.addMember(member3);

*assertEquals*(member2, list.getMembershipNotPaid().getMember(0));

*assertEquals*(member3, list.getMembershipNotPaid().getMember(1));

}

Three members object are created and added in the list. AssertEquals is used to test the equality of expected and actual list. So member2 and member3 are the expected member who did not paid, and the actual is list.getMembershipNotPaid(). So, unit root passes the test and it means that method is functioning.

# References

C. Larman (2005) “*Applying UML and Patterns*” Third Edition Pearson Education

R. Joshi (2015) “**Java Design Patterns**” Java Code Geeks

Y. Daniel Liang (2009) “Introduction to Java Programming” Seventh Edition Pearson Education