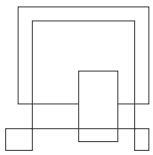


Course Report

Claudiu Rediu 266129

Dominika Kubicz 266148

Nikita Roskovs 266900

Tudor Ciobanu 267632

Ib Havn (IHA)

Joseph Chukwudi Okika (JOOK)

Knud Erik Rasmussen (KERA)

Mona Wendel Andersen (MWA)

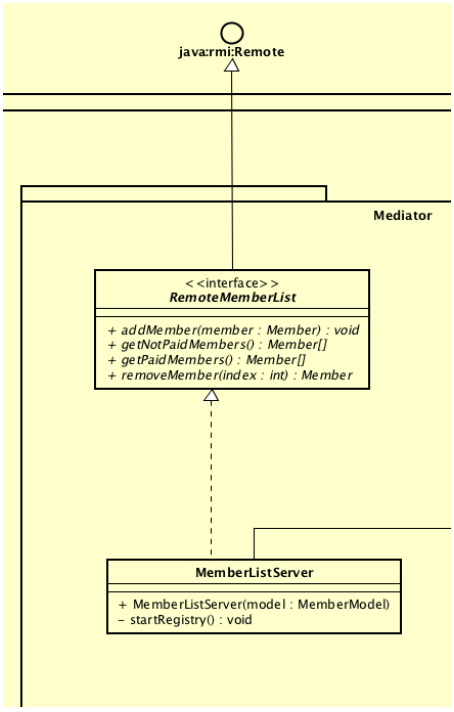
ICT Engineering

Semester 2

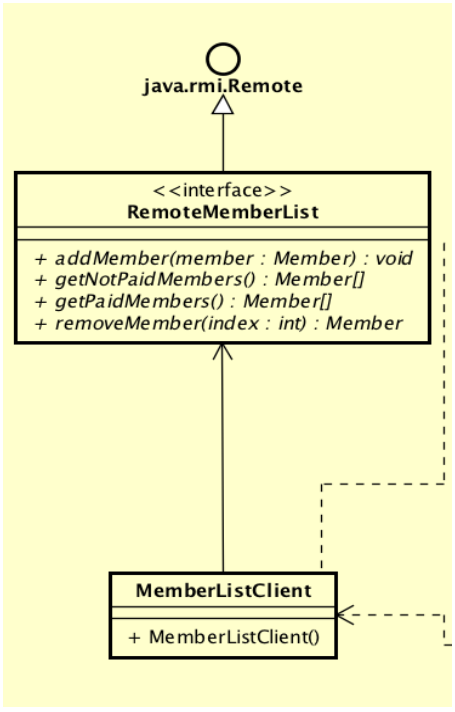
26.04.2018

Client-Server System (RMI)

The client-server part of the system is implemented using RMI. The reason we used RMI is that it already handles lots of network related code that allows the transmission of messages between clients and server. Thus, we used the salvaged time and work to design and implement the design patterns.



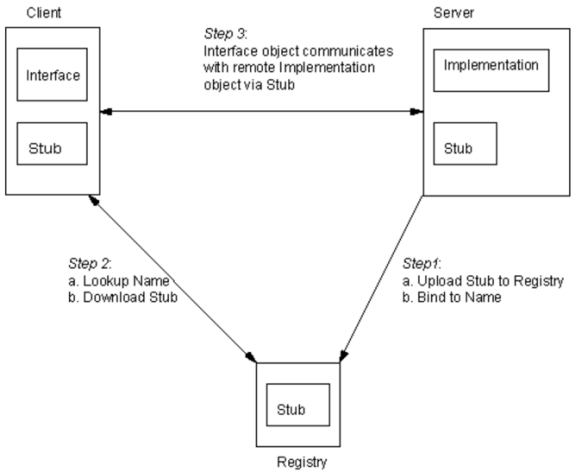
Server-side



Client side

The server class implements the RemoteMemberList, which extends Remote. On the client side, instead of implementing the interface, the client class has an instance of it. This way, the client knows which actions it can perform on the server.

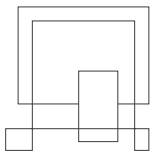
There are 3 steps to implement RMI.



Server side:

```
public MemberListServer(MemberModel model) {
   this.model = model;
   startRegistry();
   try {
      UnicastRemoteObject.exportObject(this, 0);
      Naming.rebind("Member", this);
      System.out.println("Server started...");
   }
   catch(RemoteException | MalformedURLException e)
   {
      System.out.println("Unable to started server!!! ");
      e.printStackTrace();
   }
}
private void startRegistry() {
   try {
      Registry reg = LocateRegistry.createRegistry(1099);
      System.out.println("Registry started... ");
   } catch (RemoteException e) {
      // TODO Auto-generated catch block
      e.printStackTrace();
   }
}
```

1. Create and start the registry.
2. Publish the object to start listening to clients.
3. Upload the stub to registry and bind it to “Member”.



Client side:

```
private RemoteMemberList list;

public MemberListClient(String host) throws IOException, NotBoundException
{
    list = (RemoteMemberList) Naming.lookup("rmi://localhost:1099/Member");
}
```

As stated above, the client class has an instance of the interface. In the constructor, it looks up to find the “Member” registry and assigns it to the instance of the interface.

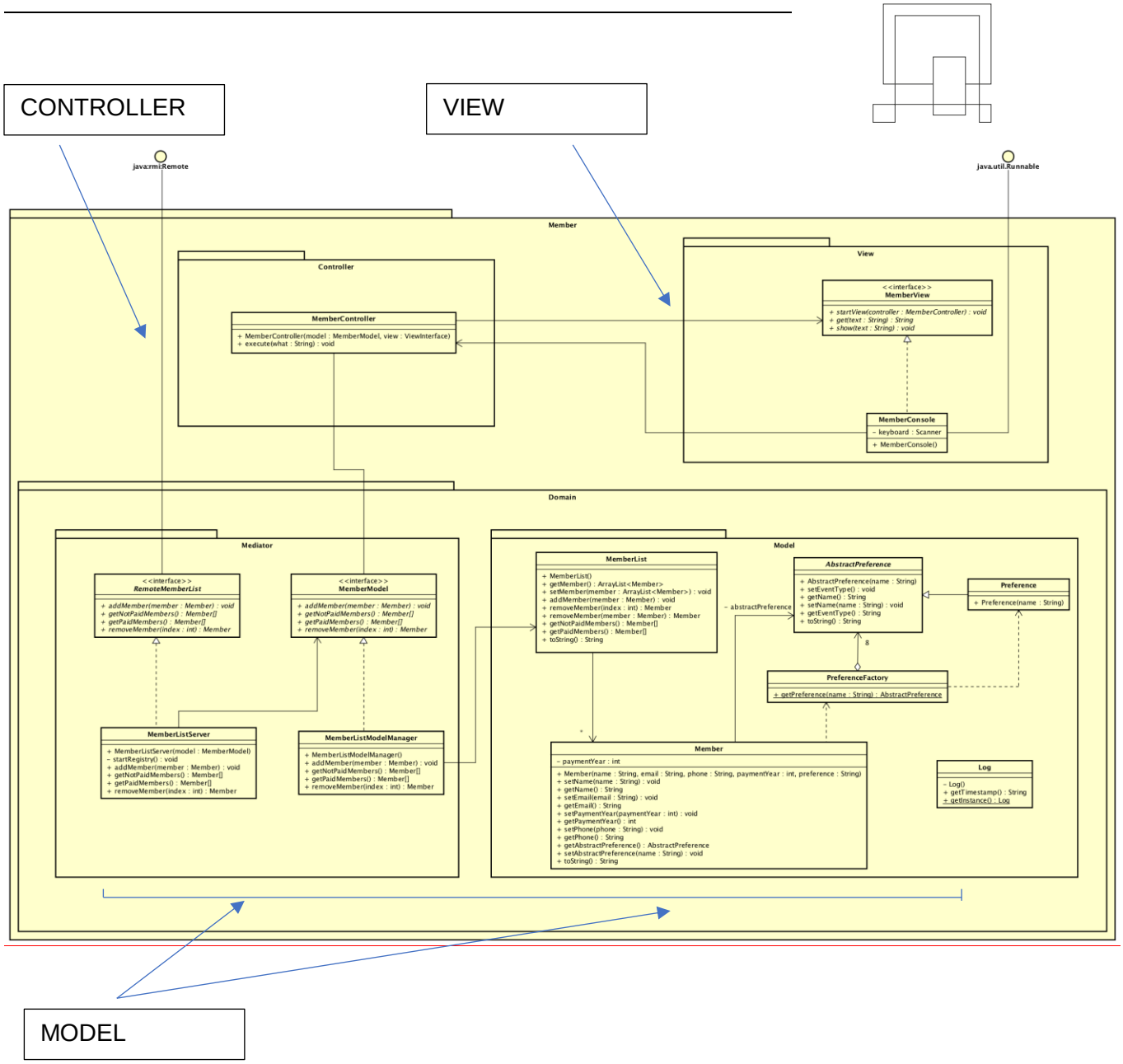
Design Pattern

Model-View-Controller (MVC)

Reason for use: It is used to separate the application’s view/user interface and its business logic.

Design:

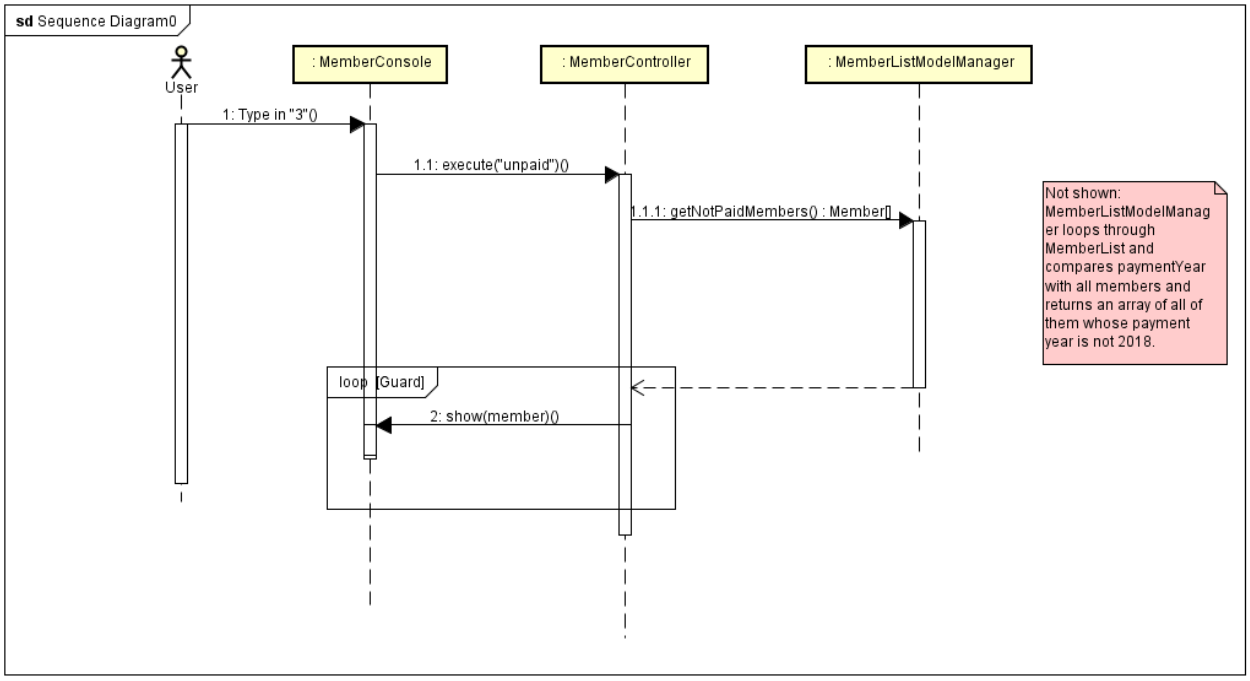
The system is split into 3 main parts: the domain, controller and view. The domain includes the mediator and the model. The model includes all the business logic needed for the system, while the mediator is just the façade for the model. The controller translates the user’s interaction with the view into actions that the model can perform and it may also select a new view (displaying some results to the user). The view presents the contents of the model and register the user’s interactions with the system.



NOTE: MVC is also used on the client side of the system.

Description:

This is the sequence diagram for displaying all members who have not paid for the membership.

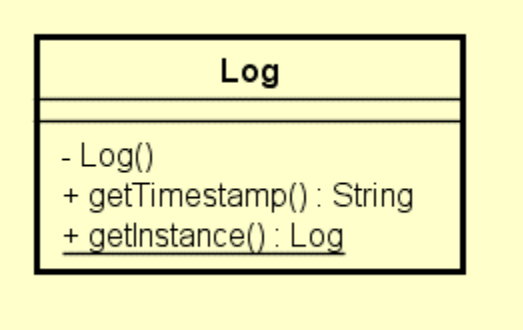


Singleton Pattern

Definition:
A singleton has a private constructor and a private static method to get an instance of itself to ensure that a class only has one instance and provides a global point to access it.

Reason for use: It is used for the intent of storing the exact time and date of when a change occurs in the system.

Design:



The main parts of the singleton are the private constructor Log() and the static method +getInstance() which retrieves a Log Object. This Class is not connected to any other class and because of the static method it can be used from anywhere in the system.

Implementation:

```
    private Date date;
    private static Log instance;

    private Log()
    {
        date = Calendar.getInstance().getTime();
    }

    public String getTimestamp() { //use this to get the present time when you add a member
        SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy HH:mm:ss");
        return sdf.format(date);
    }

    public static Log getInstance() // singleton
    {
        if(instance == null)
            instance = new Log();
        return instance;
    }
}
```

When a new member is added we want to keep the time when he/she was added. We achieve this by using Log as a singleton. By using the Log class, we make all the responsibility of saving information regarding the time of adding a member be handled by this class.

Test:

A Junit test was used for the test. During its use the following were concluded:

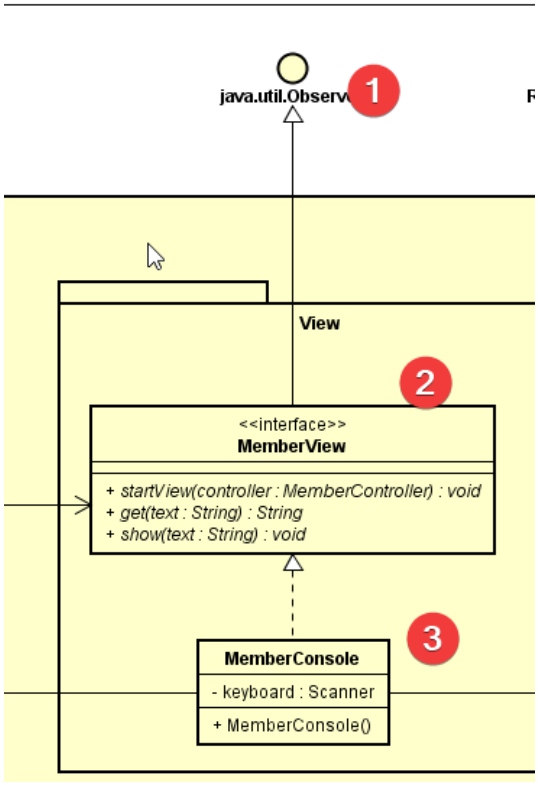
Number	Feature being tested	Passed
1	A log being created and printed when a member is added	YES
2	A log being created and printed when a member is removed	YES

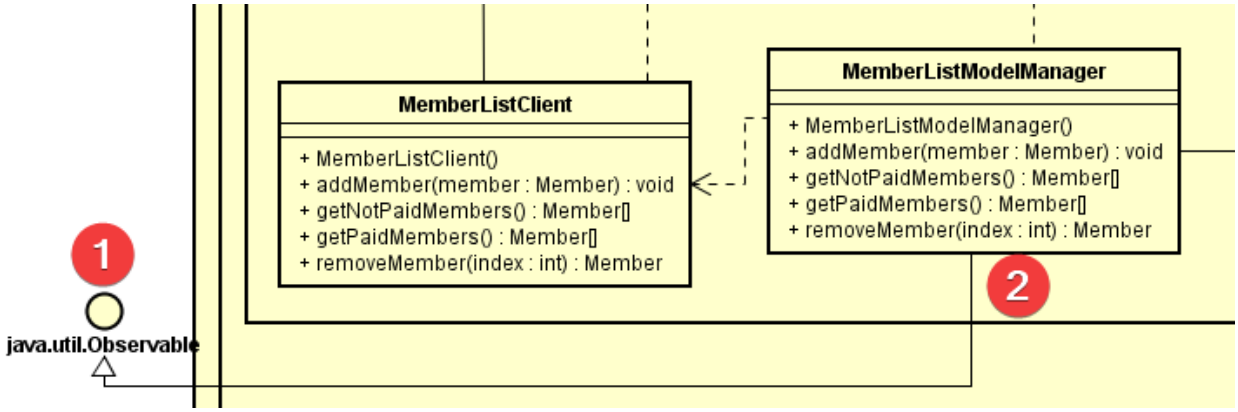
Observer Pattern

Definition: The observer pattern is a software design pattern which consists of a subject that contains multiple observers. Those observers are notified and updated when the state of the subject changes.

Reason for use: It is used with the intent of having the view updated each time something in the model is changed.

Design:





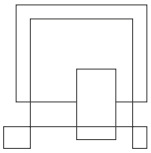
The subject to be observed is the model manager which extends Observable. It is used as a subject because it is the main link because the model and the controller.

The observer will be the MemberConsole but because it has the MemberView as an interface, the choice was made that MemberView extends the Observer. Inside the MemberConsole there is an update() method from extending.

Implementation:

```
public interface MemberView extends Observer{
```

```
public MemberController(MemberListModel model, MemberView view)
{
   this.model = model;
   this.view = view;
   Observable obs = (Observable) this.model;
   obs.addObserver(view);
}
```



```java
@Override
public void addMember(Member member) {
    model.addMember(member);
    setChanged();
    notifyObservers(Log.getInstance().getTimestamp() + " Member added");
}

@Override
public Member[] getNotPaidMembers() {
    return model.getNotPaidMembers();
}

@Override
public Member[] getPaidMembers() {
    return model.getPaidMembers();
}

@Override
public Member removeMember(int index) {
   setChanged();
   notifyObservers(Log.getInstance().getTimestamp() + " Member removed");
   return model.removeMember(index);
}
```

```java
 @Override
 public void update(Observable o, Object arg)
 {
    System.out.println(arg);
 }

}
```

The controller has both access to the view and model so there is where the view is added as an observer to the model. The view is notified when a member is added or removed by using the Log class for the time. The update is kept as a simple method that just prints the input from the methods in the model. The implementation is kept simple with the scope of showing the time when certain actions are performed on the system.

Flyweight pattern

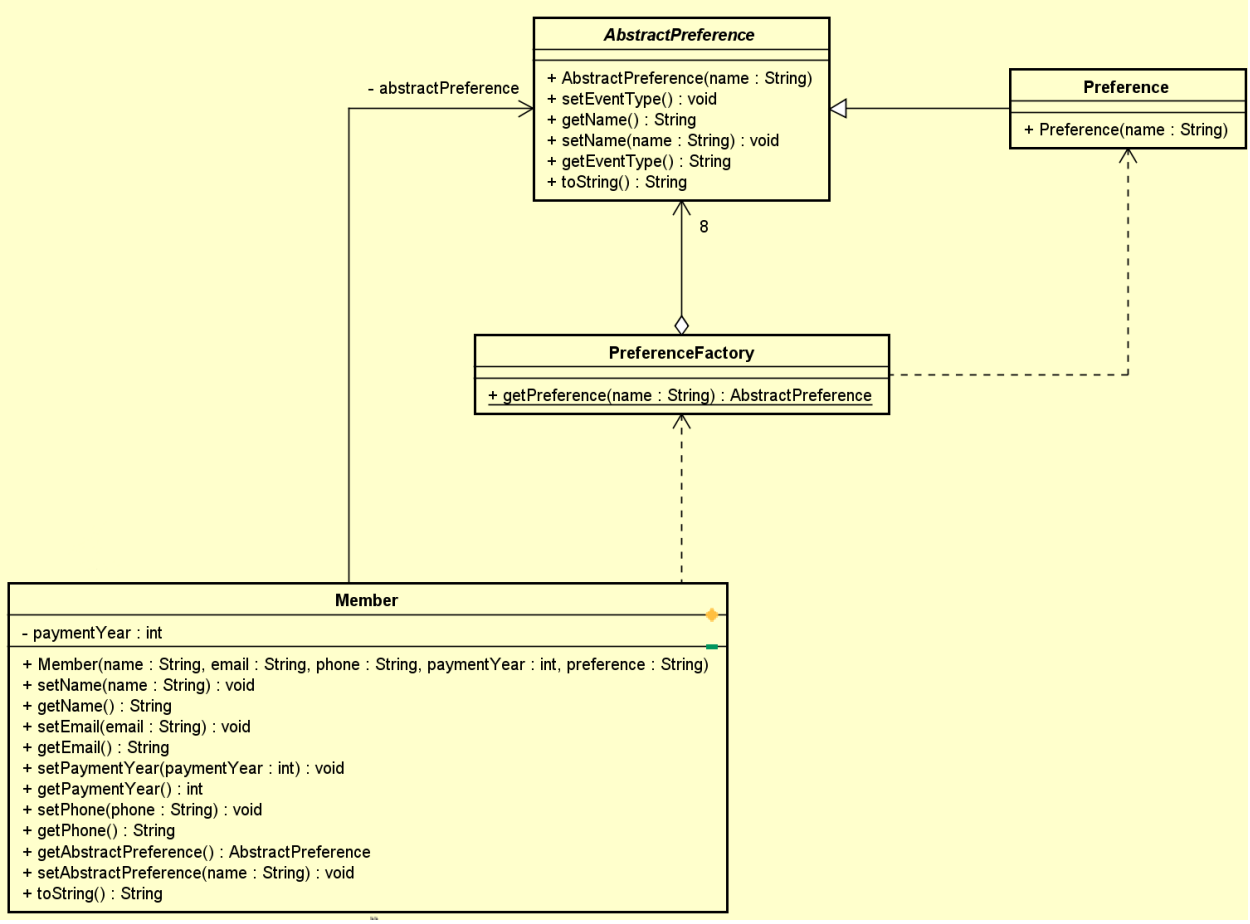
Definition:

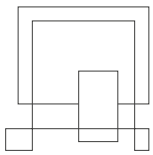
A flyweight has an object class and it's "object factory" which has a hash map to store the objects. The factory implements an abstract class to which an object that uses flyweight's instance variable has access to.

Reason for use:

We use flyweight to have a limited number of preference assigned to numerous members, to lower the memory usage.

Design:





In our system flyweight has a Preference, a PreferenceFactory, an AbstractPreference and a Member class. The Preference extends the AbstractPreference and in the Preference class we only have a constructor where we call the AbstractPreference constructor. In the PreferenceFactory we have a hash map to store the preferences and a static method getPreference() which gets an existing preference from the hash map or creates a new one and add it to the hash map. In the AbstractPreference we have a constructor that when we create a new preference sets it name and depending on the name assigns an event type to it. The member class in its constructor calls the PreferenceFactory to get a preference from the hash map and assigns it to a certain member.

Implementation:

```
public Preference(String name) {
    super(name);

}
```

Preference constructor calling AbstractPreference's constructor.

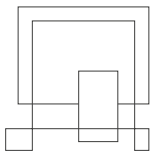
```
    private static HashMap<String, Preference> preferences = new HashMap<String, Preference>();

  public static AbstractPreference getPreference(String name) //use to get a certain preference
  {
      Preference item = preferences.get(name);
      if (item == null)
      {
         item = new Preference(name);
         preferences.put(name, item);
      }

      return item;
  }
```

PreferenceFactory has a hash map and a static method getPreference().

```
public AbstractPreference(String name) {
    this.name = name;
    setEventType(); // we create a preference using the name of it and the event type attached to it
    // which is checked by the method
}
 public void setEventType() {
     if (name.equals("Astrology"))
         eventType = "Astrology Event";
     else if (name.equals("Yoga"))
         eventType = "Yoga Event";
     else if (name.equals("Trips"))
         eventType = "Trips";
     else if (name.equals("Meditation"))
         eventType = "Meditation Event";
     else if (name.equals("Workshop"))
         eventType = "Workshop Event";
     else if (name.equals("Reincarnation"))
         eventType = "Reincarnation Event";
     else if (name.equals("Karma"))
         eventType = "Karma Event";
     else if (name.equals("Alternative Health Care"))
```



AbstractPreference has a constructor setting a name and calls a setEventType() method and it assigns a correct event type depending on the preference.

```
public Member(String name, String email, String phone,int paymentYear ,String preference) {
    this.name = name;
    this.email = email;
    this.paymentYear = paymentYear;
    this.phone = phone;
    this.abstractPreference = PreferenceFactory.getPreference(preference);
}
```

Member class is calling the getPrefrence method from the PreferenceFactory.

Testing:

A Junit test was used for the test. During its use the following were concluded:

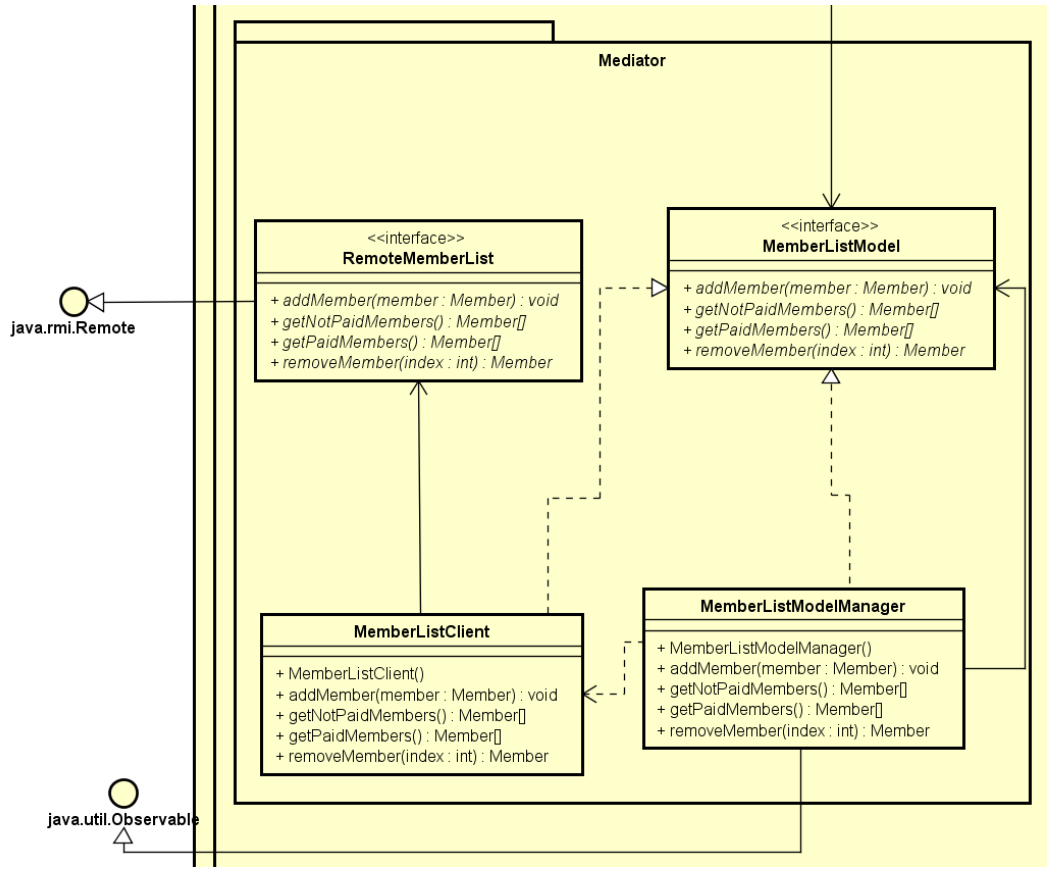
Number	Feature being tested	Passed
1	Creating a new preference and putting it in the hash map	YES
2	Assigning a preference already existing in the hash map	YES
3	Assigning an event type to a preference	YES

Proxy pattern:

Definition: The observer pattern is a software design pattern that allows to control an access to an object by a placeholder.

Reason for use: The intent of using this pattern is to provide a placeholder for an object to control references to it to manage the list of members obtained through RMI by remote class.

Design:



MemberListClient obtains the MemberList through RMI .

MemberListClient and MemberListModelManager classes implement MemberListModel, therewith MemberListModelManager has an instance variable of MemberListModel class, therefore MemberListModelManager class has access to MemberListClient class which obtains MemberList that must be managed.

The fact that both MemberListClient and MemberListModelManager have the same interface allows to delegate methods from managings class to the client's class that has an actual list of members.

Implementation:

```
10 public class MemberListModelManager extends Observable implements MemberListModel{ #1
11    private MemberListModel model; #2
12
13    public MemberListModelManager() throws IOException {
14       try
15       {
16          model = new MemberListClient("localhost"); #3
17       }
18       catch (NotBoundException e)
19       {
20          // TODO Auto-generated catch block
21          e.printStackTrace();
22       }
23    }
24
25    @Override
26    public void addMember(Member member) {
27       model.addMember(member); #4
28       setChanged();
29       notifyObservers(Log.getInstance().getTimestamp() + " Member added");
30    }
31
```

MemberListModelManager implements MemberListModel(#1). Instance variable "model" (#2) actually references to MemberListClient class (#3) that actually has a MemberList (#5). Methods are delegated to MemberListClient class (#4). MemberListClient class works right with a MemberList (#6) that was obtained through RMI (and parsed into RemoteMemberList)

```
12 public class MemberListClient implements MemberListModel
13 {
14    private RemoteMemberList list; #5
15
16    public MemberListClient(String host) throws IOException, NotBoundException
17    {
18       list = (RemoteMemberList) Naming.lookup("rmi://localhost:1099/Member");
19    }
20
21    @Override
22    public void addMember(Member member) {
23       try
24       {
25          list.addMember(member); #6
26       }
27       catch (RemoteException e)
28       {
29          // TODO Auto-generated catch block
30          e.printStackTrace();
31       }
32    }
```