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**

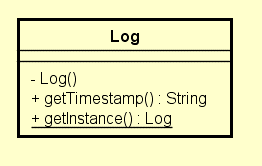
# Design Pattern

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.

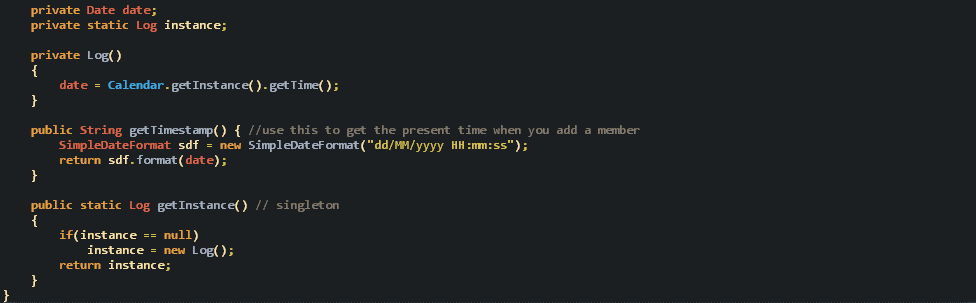
Reason for use: It is used for the intent of storing the exact time and date of when a change occurs in the system.

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:



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:

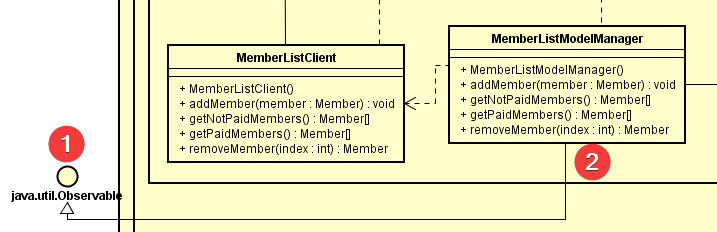
A Junit test was used for the test. During its use the following were concluded:

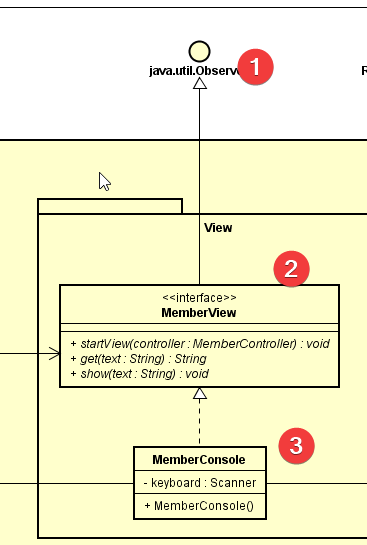
|  |  |  |
| --- | --- | --- |
| Number | Feature being tested | Passed |
| 1 | A log being created and printed when a member is added | YES |
| 2 | A log being created and printed when a member is removed | YES |

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:

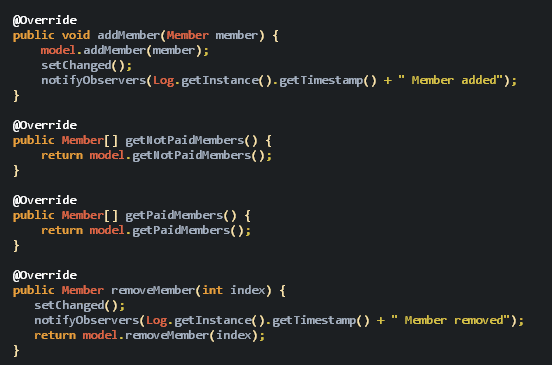
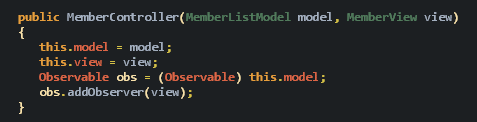


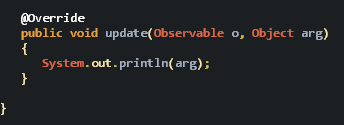


The subject to be observed is the model manager which extends Observable. It is used as a subject because it is the main link because the model and the controller.

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:



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.

Test:

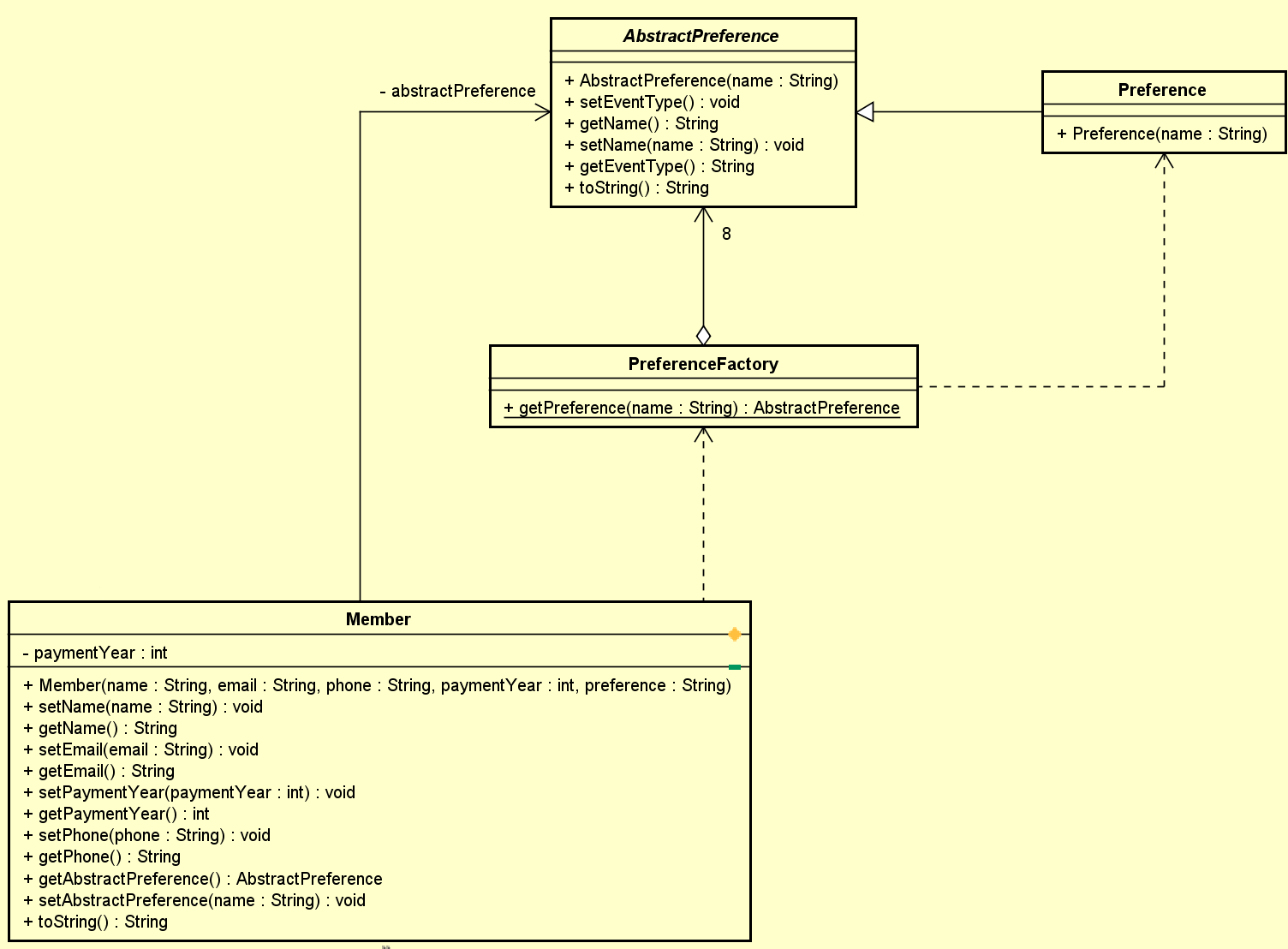
Flyweight pattern

Definition:

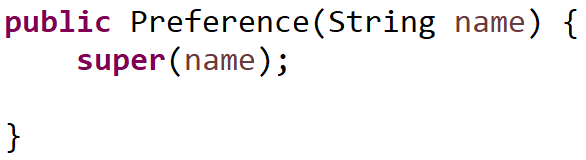
A flyweight has an object class and it’s “object factory” which has a hash map to store the objects. The factory implements an abstract class to which an object that uses flyweight’s instance variable has access to.

Reason for use:

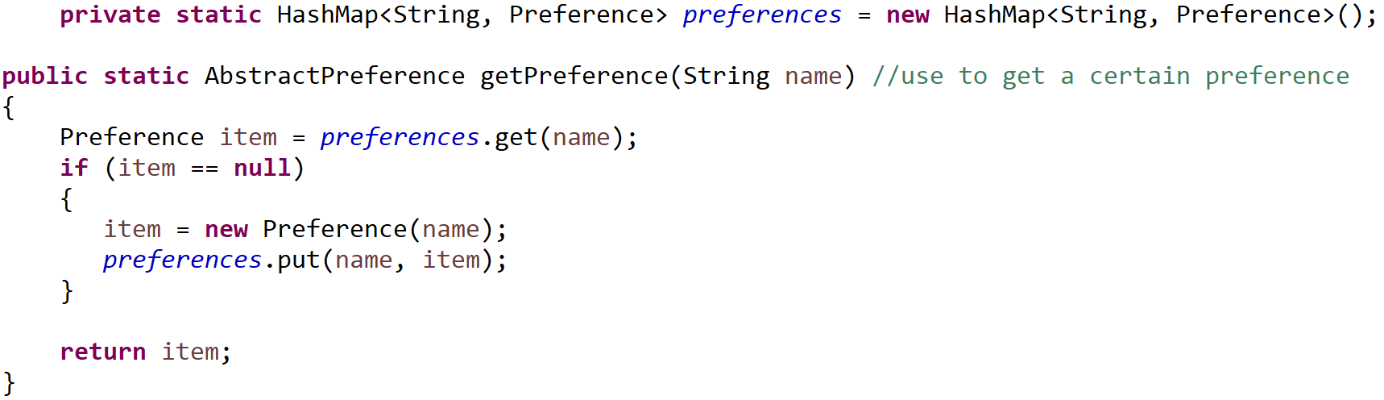
We use flyweight to have a limited number of preference assigned to numerous members, in order to lower the memory usage.

Design:

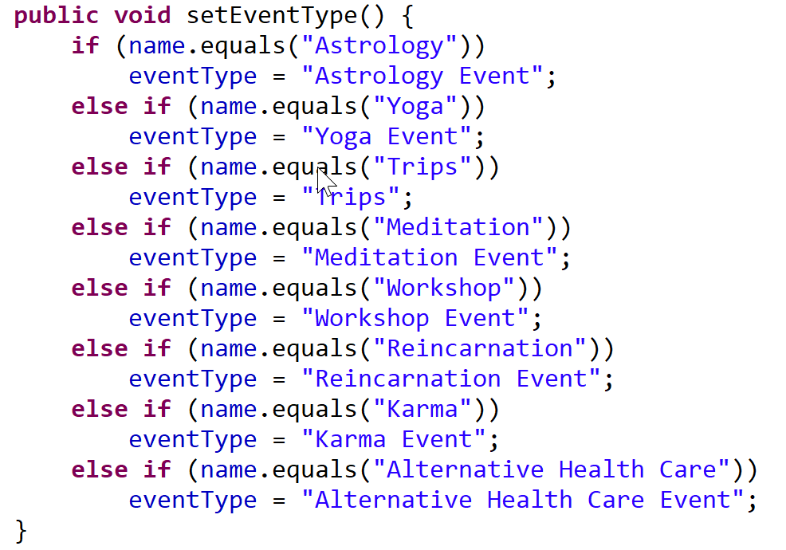
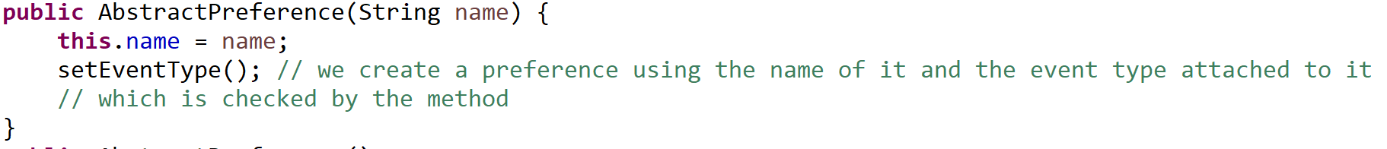
In our system flyweight has a Preference, a PreferenceFactory, an AbstractPrefrence and a Member class. The Preference extends the AbstractPrefrence and in the Preference class we only have a constructor where we call the AbstractPreference constructor. In the PreferenceFactory we have a hash map to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

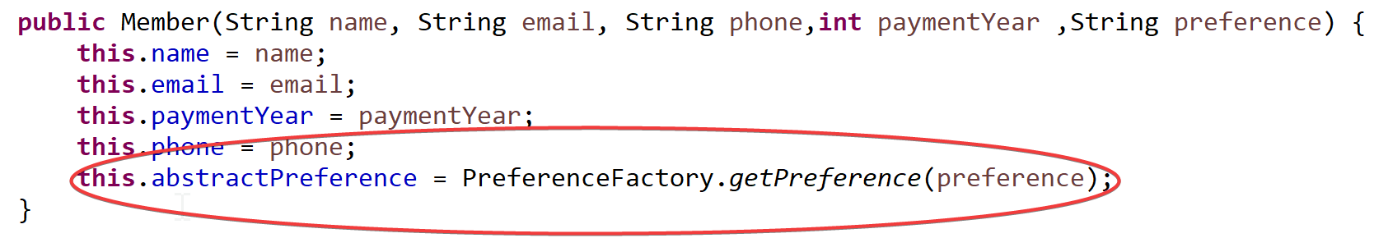
Implementation:

Preference constructor calling AbstractPreference’s constructor.



PrefrenceFactory has a hash map and a static method getPrefrence().



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

Member class is calling the getPrefrence method from the prefrenceFactory.

Testing:

|  |  |  |
| --- | --- | --- |
| 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 |