**Appendices O**

**Observer Pattern**

To implement the observer design pattern, there was a number of steps I had to take which are documented below.



Fig. 1 – Extract from ClientImpl, class header

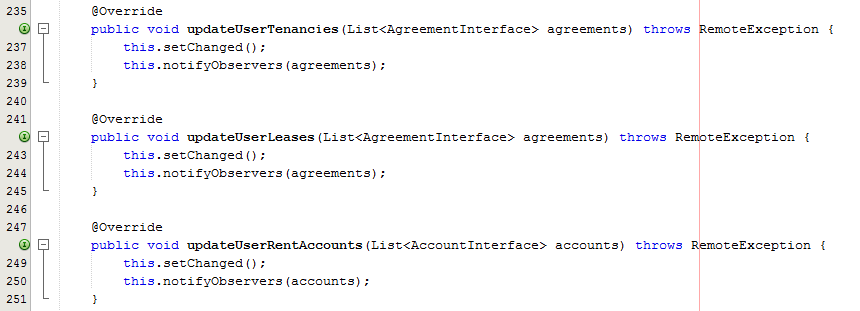


Fig. 2 – Extract from ClientImpl class

As you can see from Fig. 1 and Fig. 2, to do this I first had to make the ClientImpl class extend Observable, which meant that I had to create a method which notifies any observers when there is a change.

The methods that notify the Observers (GUI) when any changes to either agreements or rent accounts are updateUserTenancies(), updateUserLeases() and updateUserRentAccounts(), of which all of these methods invoke two methods inherited from the Observable class, setChanged() and notifyObservers() which will notify the list of observers, that there has been a change to the object being observed (ClientImpl), and then passes the updated object as a parameter of the notifyObservers() method to the observers (GUI).

Now I have to amend the home screen, which is the GUI that will be the observer, and needs to updated whenever the observable object (ClientImpl) notifies of any state change.



Fig. 3 – Extract from HomeForm class (GUI), class header

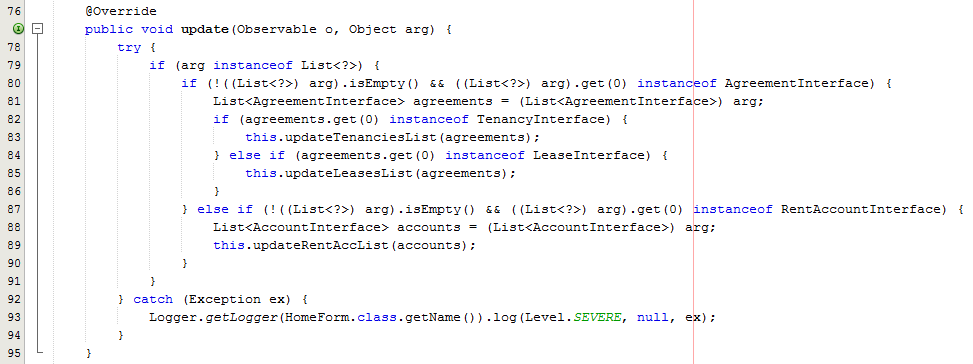


Fig. 4 – Extract from HomeForm class (GUI), update()

As you can see from Fig. 3 and Fig. 4, the HomeForm class implements Observer, which also means the class is able to extend JFrame as it is a GUI. I then have to provide an implementation for the update method, which as explained before, will be invoked by the Observable class, when the Observer invokes setChanged() and then notifyObservers(). As you can see from Fig. 4, the updated object is passed as a parameter. However, because the updated object (either a list of Agreements or Accounts) is passed as an Object I need to check if the object is an instance of the required object, and because the object passed should be a list, I first need to check to see if the Object is instance of List. But because of Type Erasure the compiler at run time does not know the type of object within a list, so I am unable to test if the list has the correct type of elements without actually obtaining an element from the list and checking the elements type.

So if the object is of type List, I then need to check if the list is empty and if not then actually get an element out of the list and test the type of the object is either instance of Agreement or instance of Account.

I then invoke an update method (updateTenanciesList, updateLeasesList or updateRentAccountList, depending on List passed as parameter for update method, which will amend the GUI display to reflect the change that has occurred to the Observable object.

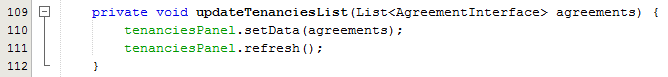


Fig. 5 – Extract from HomeForm class (GUI)

Above I have shown the implementation of the Observer pattern, and updates occurring in the HomeForm GUI (View) as a result of an update to the ClientImpl object (Model), however I have not shown how the ClientImpl object is updated, which I am now going to show, and as explained, I adopt the push data exchange model for this part of the system functionality, as the server needs to notify all users of a change to the office tenancies, leases or rent accounts.

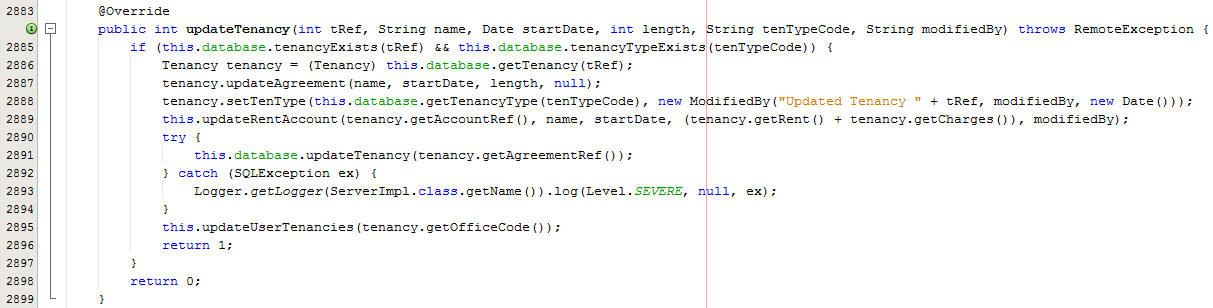


Fig. 6 – Extract from ServerImpl class – updateTenancy()

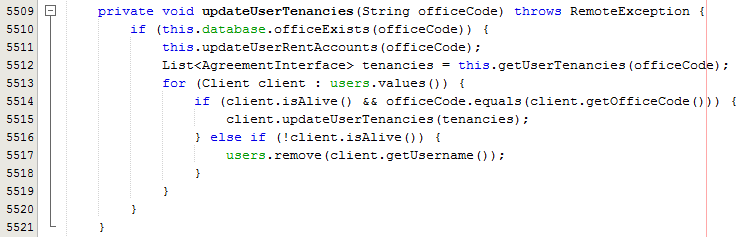


Fig. 7 – Extract from ServerImpl class - updateUserTenancies()

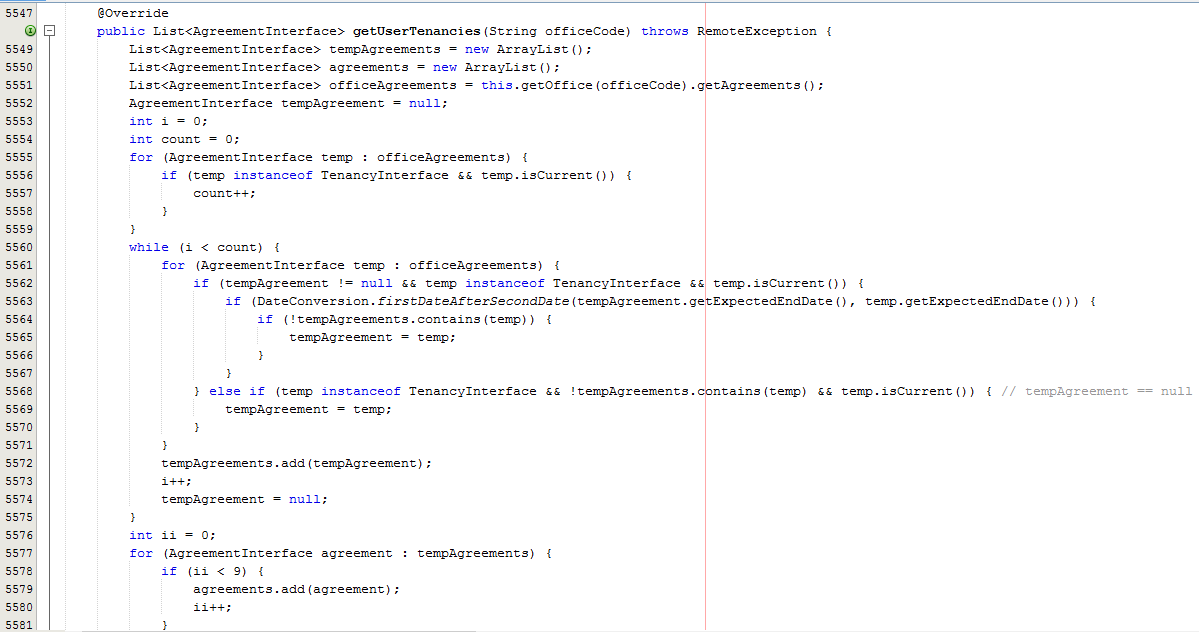


Fig. 8 – Extract from ServerImpl class getUserTenancies()

As you can see from Fig. 6, whenever a tenancy object is updated (as well as created, ended and deleted a tenancy), the updateUserTenancies method is invoked, this is the same as Leases and Rent Accounts. As shown in Fig. 7, the updateUserTenancies method invokes a getUserTenancies() method, which interacts with the database class to get all tenancies and then sorts the list of tenancies, and produces a list of 10 tenancies for the required office and returns the list to the updateUserTenancies.

Once the list is created and all required elements have been added to the list and passed back to the updateUserTenancies method, I then go through a list of clients checking to see if the client is still alive, and if so and the client needs to receive the update (only send out update to clients of that office I invoke updateUserTenancies() on the client, and pass the updated list as a parameter to the ClientImpl method, which as shown earlier in the Appendices, invokes the setChanged() and notifyObservers methods, which invokes the Observer objects update method.

The observer pattern then allows the system for ‘MSc Properties’ to ensure that the clients home form is always updated, but also does not send unnecessary updates to all clients that don’t need the update. A similar implementation is provided for each of the three lists, Tenancies, Leases and Rent Accounts.