Appendices N

Observer Pattern

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

```
54    public class ClientImpl extends Observable implements Client {
```

Fig x – Extract from ClientImpl, class header

```
145     @Override
        public void updateUserAgreements(List<AgreementInterface> agreements) throws RemoteException {
147         this.setChanged();
148         this.notifyObservers(agreements);
149     }
150
151     @Override
        public void updateUserRentAccounts(List<RentAccountInterface> accounts) throws RemoteException {
153         this.setChanged();
154         this.notifyObservers(accounts);
155     }
```

Fig x – Extract from ClientImpl class, updateUserAgreements() - updateUserRentAccounts()

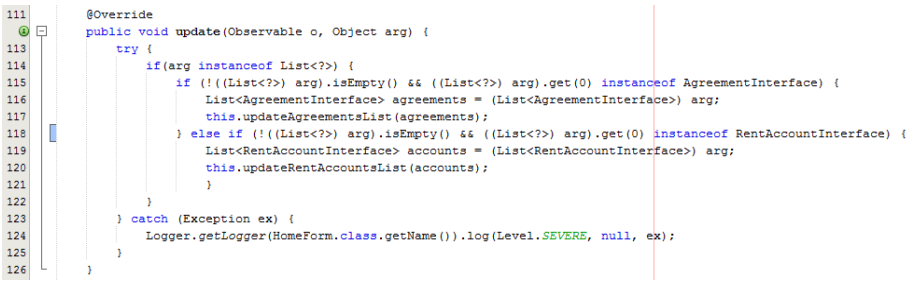
As you can see from fig x and fig x, to do this I first had to make the ClientImpl class extend Observable, which meant that I had to create a method which notifies any observers when there is a change.

The methods that notify the Observers (GUI) when any changes to either agreements or rent accounts are updateUserAgreements() and updateUserRentAccounts() and both 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 need to update whenever the observable notifies of any state change.

```
30    public class HomeForm extends JFrame implements Observer {
```

Fig x – Extract from HomeForm class (GUI), class header



```
111     @Override
        public void update(Observable o, Object arg) {
113         try {
114             if(arg instanceof List<?>) {
115                 if (!((List<?>) arg).isEmpty() && ((List<?>) arg).get(0) instanceof AgreementInterface) {
116                     List<AgreementInterface> agreements = (List<AgreementInterface>) arg;
117                     this.updateAgreementsList(agreements);
118                 } else if (!((List<?>) arg).isEmpty() && ((List<?>) arg).get(0) instanceof RentAccountInterface) {
119                     List<RentAccountInterface> accounts = (List<RentAccountInterface>) arg;
120                     this.updateRentAccountsList(accounts);
121                     }
122             }
123         } catch (Exception ex) {
124             Logger.getLogger(HomeForm.class.getName()).log(Level.SEVERE, null, ex);
125         }
126     }
```

Fig x – Extract from HomeForm class (GUI), update()

As you can see from fig x and fig x, the HomeForm class implements Observer, which also means as it is a GUI, it is able to still extend JFrame, but also means that it then needs 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 x, the updated object is passed as a parameter. However, because it 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 Rent Account.

I then invoke an update method which will amend the GUI display to reflect the change that has occurred to the Observable object.

The last part of the implementation of the Observable pattern (although it is actually now fully implemented), is that the controller must invoke ClientImpl.updateAgreements() or ClientImpl.updateRentAccounts() and again this is one of the only implementations of the push data exchange model, because the server is actually pushing a change to the client, instead of the client pulling “requesting” data from the server.

```
3803    private void updateUserAgreements(String officeCode) throws RemoteException {
3804        if (this.database.officeExists(officeCode)) {
3805            List<AgreementInterface> agreements = new ArrayList();
3806            List<AgreementInterface> tempAgreements = this.getOffice(officeCode).getAgreements();
3807            int i = 0;
3808            AgreementInterface tempAgreement = null;
3809            while (i < 15 && !tempAgreements.isEmpty()) {
3810                for (AgreementInterface temp : tempAgreements) {
3811                    if (tempAgreement != null) {
3812                        if (tempAgreement.getExpectedEndDate().compareTo(temp.getExpectedEndDate()) < 0) {
3813                            tempAgreement = temp;
3814                        }
3815                    } else { // tempAgreement == null
3816                        tempAgreement = temp;
3817                    }
3818                }
3819                agreements.add(tempAgreement);
3820                tempAgreements.remove(tempAgreement);
3821            }
3822
3823            for (Client client : users.values()) {
3824                if (client.isAlive() && officeCode.equals(client.getOfficeCode())) {
3825                    client.updateUserAgreements(agreements);
3826                } else if (!client.isAlive()) {
3827                    users.remove(client.getUsername());
3828                }
3829            }
3830        }
3831    }
```

Fig x – Extract from ServerImpl class updateUserAgreements()

```
3833    private void updateUserRentAccounts(String officeCode) throws RemoteException {
3834        if (this.database.officeExists(officeCode)) {
3835            List<RentAccountInterface> accounts = new ArrayList();
3836            List<AccountInterface> tempAccounts = this.getOffice(officeCode).getAccounts();
3837            int i = 0;
3838            AccountInterface tempAccount = null;
3839            while (i < 15 && !tempAccounts.isEmpty()) {
3840                for (AccountInterface temp : tempAccounts) {
3841                    if (temp instanceof RentAccountInterface && tempAccount != null) {
3842                        if (tempAccount.getBalance() > temp.getBalance()) {
3843                            tempAccount = temp;
3844                        }
3845                    } else if (temp instanceof RentAccountInterface && tempAccount == null) {
3846                        tempAccount = temp;
3847                    }
3848                }
3849                accounts.add((RentAccountInterface) tempAccount);
3850                tempAccounts.remove(tempAccount);
3851                i++;
3852            }
3853            for (Client client : users.values()) {
3854                if (client.isAlive() && officeCode.equals(client.getOfficeCode())) {
3855                    client.updateUserRentAccounts(accounts);
3856                } else if (!client.isAlive()) {
3857                    users.remove(client.getUsername());
3858                }
3859            }
3860        }
3861    }
```

Fig x – Extract from ServerImpl class updateUserRentAccounts()

As you can see from fig x and fig x, each of these methods prepare the wist which will be the updated list (object to be passed via the notifyObservers method from ClientImpl class) after

any changes have occurred. Once the list is created and all required elements have been added to the list, I then go through a list of clients checking to see if client is still alive, and if so if the client needs to receive the update (only send out update to clients of that office), and if the client is then I invoke Client.updateUserRentAccounts() or Client.updateUserAgreements(), and pass the updated list as a parameter to the method.

This then makes the Observable object the ClientImpl invoke the setChanged() and notifyObservers methods discussed earlier, 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.