COMP47480 Practical 6: Observer Pattern

Sukrat Kashyap (14200092)

24 April 2018

1 Work Done

In this practical, our task was to use observer pattern to solve a particular problem. The Person and the AlarmClock were acting as data classes and all the work was done in the AlarmApplication making it have god class tendencies. To solve this problem, we created an abstract class for the Subject which acts the source of broadcasting when something happens. The broadcasting is done to all the interface Observers which have subscribed to listen to the Subject. These 2 were extended and implemented by AlarmClock and Person class respectively. AlarmClock calls the Observer update method upon a change in its time. Person class on update checks whether the AlarmClock has reached the alarm time if it has it wakes itself up. The AlarmApplication was then made to only instantiate the objects, register the person and call the tick method. Since we had to notify the person whenever the alarm time is set as well. The Person class this time had to store the wake-up time as well. For this reason, we changed the update method to pass in the object for pulling the state and an args object for pushing the event name. This was done so that the Person knows which event is being called and act accordingly. Cron class was extended on Observer which also listened to the Clock and acted when the time was right.

2 Reflections

Design patterns are a general reusable solution to a commonly occurring problem within a given context. It is not the absolute truth in software development but a template. In this practical, we tried to employ and solve the problem using the Observer pattern. Observer pattern is a behavioural design pattern that deals with common object communication. We were also dealing with an object communication between the alarm clock and the person to get up when the alarm is reached. Observer pattern uses an object called subject which maintains the list of observers who have subscribed to the subject and notifies the observers upon a change in the state of the subject. This notification is done via calling one of the methods of the observer. I feel observer pattern is great to solve a number of simple communication problems but also has a number of downfalls.

In the practical, our subject was the alarm clock and the observers were the person and cron objects. The person and cron were subscribed to the subject by adding them

to the observer's list held by the subject. Our state change in the case of the alarm clock was the tick. The complexity increased when the person class also wanted to know whenever the alarm time is set. Push and pull are 2 different ways in which the observer can know about the state of the subject. To make things simpler and allow for both the methodologies. I made the update method to receive the subject and an args object to be passed. The pulling of the state is done using the subject-object received whereas the pushing the state could be done through the args object. This allowed the flexibility in the implementation of the subject in the alarm clock. Differentiating between 2 events coming from the same subject was another problem. For this, I made the person to pull the state from subject-object being passed and pushed the event name from the subject. This event name pushed was used as a distinguisher between the 2 objects.

When writing the code for the person class, I felt that the person could be made to subscribe any event due to this decoupling. Because of this, I added an if statement to check if the subject passed to the person is actually alarm clock or not. This was also done since we have a common interface for the Observer this forces us to cast to access alarm clock specific methods. The argument being pushed is an Object as well requiring another cast to string. This actually introduced coupling between the alarm clock and the person object but removed coupling from the notification mechanism. This seems quite troublesome but in fact, make sense as the person know that it should act only when a type of clock tells you the type rather than any other object.

Adding of Cron was trivial and was implemented without touch any of the other classes. The communication between the two objects could have introduced complexity and number of other issues in the program in the future. It was nice to see observer pattern solving the problem with allowing easy extension in the future. Personally, these design patterns are also useful in communicating the structure of the code to another developer. The problem which I think is hard to solve in observer pattern in automatic garbage collection is the holding of the reference in the subject class. This stops the collection of the observer when it's not needed anymore. The solutions to these are weak references but that is not a stable and correct method as the garbage collection is done by the VM at an arbitrary point in time. I believe a better approach is to unsubscribe them after use. In languages where garbage collection is not done, it's easier as a destructor can be implemented to remove its reference from the subject. The delete call on the object will first call the destructor removing references from the subject and then destroy the object itself.

In my experience, observer pattern has been used a lot where ever there is a requirement of acting upon a state change. This decoupling nature of the implementation of the observer from the subject state change allows for greater extensions. I also found that this notification mechanism actually saves quite a lot of computing as the observers do not have to keep polling for the change in the subject state as they are only notified when state changes in real-time.