Entry Six*:* The Observer Pattern Exercise

Student Name: Gregory Sloggett

Student Number: 14522247

Work Done:

I firstly inspected the AlarmApplication class to see where the “God class tendencies” lay, as described by the assignment. It was clear this class was cluttered with excessive code and functionality. I also looked at the Person and AlarmClock classes and their basic functionality.

Following step one of the exercise, I made several changes to the class. As the alarm clock becomes the observable, and the person responding to it becomes the observer, an Observable abstract class and Observer interface were made. To achieve this step some refactoring was made across the three classes, particularly within the AlarmApplication class. Feature Envy was an evident characteristic of this class and removing these bad traits was critical. The Observer interface enforces the inclusion of an update method within the AlarmClock class. The Observerable class implements critical methods for the observer pattern; notify, add, and remove.

To produce the resulting output that the initial code created, the tick method in the AlarmClock class was updated with some of the God class tendencies of the original AlarmApplication class. The wake-up call invocation was removed and the notify method called from the Observable abstract class. These updates meant the code ran as expected now.

The AlarmApplication class needed to be extended to create three person objects. Each object wakes up in response to the same alarm time. Following the “push” model, the notification method along with the update method were refactored in order to achieve this change and ensure the alarm clock interacted accurately with the observers. A simple comparison check between strings of the alarm time and persons wake up time were added to other minor updates in the notify and update methods of the AlarmClock and Person classes respectively.

Following step four, and now following the “pull” model, I refactored the code by deleting the variables passing between the update and notify methods. This simplified the code but caused the update method to perform additional functionality. However, this corresponds to the objective of the update method as described in part one. With the functionality remaining the same between the two sections, and the system performing as desired, the pull model was completed.

Finishing with the Cron class was relatively straightforward, with the Cron class taking up much of the functions seen in other classes. As the question suggested, it was trivial to add this, with minor adaptations required within the AlarmClock class so that notify was invoked upon every iteration (time change) on the clock.

Reflection:

I must admit that I did find this practical quite tricky to grasp. The concept is clear to me, but the actual implementation I found quite challenging. Even though the changes are quite small that are required, knowing where to put them and how to implement them effectively was a confusing task and took some teasing out and understanding to implement. The immediate effect of extending the person class with an observer interface and extending the person class with the abstract observable class caused some initial errors. I was unsure also where the additional functionality required should

lay in the system and how inclusive the alarm application class should read. In the end the alarm application is significantly longer than it was originally, but the logic and clarity of it is much clearer.

The interaction between the AlarmClock class and Person class really takes the hassle of trying to produce everything in the AlarmApplication class. Without the observer pattern being implemented here, a lot of complications and code would have to be made to the AlarmApplication class, but by using this design pattern we can reduce that complication and almost leave the AlarmApplication class untouched.

It is safe to say that the direct implementation of the observer pattern is not something that I have used or ever implemented in my code before, but there are aspects of coding I have done in the past, and in real life where I have seen how this could be applied.

A few days ago, I was queueing in PI, the O'Brien Centre for Science café, and noticed a new system they had in place for coffee. The new system almost identically performs the observer and observable pattern that we used in this exercise. The queue was quite large, perhaps fifteen or so, and they quickly took the orders of all fifteen. We all gathered around by the milk and the sugar to await being presented with our different coffees. We (the observers), observed the coffee being made by the staff and the coffee machine (observables). By taking all orders together they improve efficiency as they can make more of the same coffee in the same batch, but I wasn't best pleased because it was a dog fight for your coffee when it arrived, and the coffee name called out (about seven people ordered americano's)! I never would have looked at this before and thought of observers and observables, and similarly I never would have coded like this either.

Granted, this is the effective way to code such a model, and I wish I had known of this observer pattern back in third-year as it would have been useful for our third-year software engineering project. This project involved the implementation of a poker twitter bot, which would automatically hold and play a game of poker with other twitter users. If you think of this with the observer pattern applied, the twitter account for the bot is the observable, as it runs the poker game. The observer (other twitter users) interacts with the observable to play the game, with the observable sending tweets/notifying the observer of the development of the game. With many players of the same game of poker, every player must be notified and updated of the development of the game. This would have been a perfect opportunity to implement and exercise the observer pattern had I been aware of it!

Overall, it was an insightful introduction into the observer design pattern. Given that I'd never seen or used this pattern before, it is quite obvious how and where it may be used and just how beneficial it may be. The two personal examples I've just given reinforce that I feel. It is definitely a pattern which I would appreciate using in the future.