FIT3077 - Software Architecture and Design
Semester 1, Assignment 2 Stage 2

Melbourne Weather Application Design Choices

Yun Hao (Jack) Zhang
Yifei (Freya) Gao

The basis of our application is based on the Observer pattern. This project requires us to retrieve information regarding different locations from a server and display it in different monitors as they are continuously updated over time. Using the observer pattern, we are able to more efficiently manage the relations between them.

The *Subject* is responsible for interacting with the source and storing the state of the data at an instance, while the *Observer* stores an up-to-date copy of the data for operations. Once the information is updated, the *Subject* will notify specific *Observers* to retrieve the new data. Separating them encapsulates their functionality and minimises coupling in our application, and avoids the need for each *Observer* to constantly check if the data has been changed. In our application, we create threads that run asynchronously in the background to executes our *Grabber* classes to simulate the update of the weather service, and retrieves new data every specified amount of time (5mins for MelbourneWeather2, 20 seconds for MelbourneWeatherTimeLapse), and the *Subject* will call the *update()* method on all its observers to retrieve that information. This allows us to more efficiently manage multiple *Observers* observing the same *Subject* as needed. When an *Observer* is no longer needed, it can simply be detached from its *Subject*.

We have our concrete *LocationSubject* class inherit from the abstract *Subject* class, and *LocationObservers* class implement the *Observer* interface. We wanted the allow different types of Observers to be attached to one Subject, and therefore made the *Observer* class an interface. This separates the specific functions needed for our weather service, and allows expandability if other types of subjects and observers are to be implemented.

Another pattern we used is the Adapter pattern. Different to Stage 1 specifications, we were required to create different types of monitors for the same data. To achieve this, instead of having the Observer directly interacting with the GUI monitors, we created different Adapter classes (*LiveFeedAdapter* and *TimeLapseAdapter*) that acts as a bridge to parse the retrieved data and apply it to our GUI monitors. This way, we can separate and hide the logic needed to convert the data into appropriate format, instead of modifying the Observer for a specific purpose and reducing reusability, and the GUI will only have methods to update its elements. An example in this assignment would be when creating a monitor to display timelapse graph view, the *TimeLapseAdapter* receives temperature and timestamp as strings, converts them into Float and Date objects respectively, and adds the entry to the TimeSeries variable that the monitor to plots from.

Initially, we thought about parsing temperature and rainfall data into float inside the LocationObserver instead of inside the Adapter. However, there are cases where the server fails to provide the data, and we receive an empty string in return. Since the input parameter types of a method are restricted in java, if we specify that the Adapter must receive variables of “Float” type, there is no ideal way for the LocationObserver to pass on a Float value that specifies the absence of correct value, since any float is potentially valid. Therefore, we simply pass on the raw string data to the Adapter, and it will then handle cases where the string can be parsed into a Float, or to display an indication of no data in the monitor.

We also wanted to reduce the dependency between our weather display monitors and other classes. All Adapters for an Observer are stored inside an array in that Observer, and when the user closes a weather display monitor, the monitor calls the *Adapter* to detach the *LocationObserver* from *LocationSubject* if no other adapters exist, before disposing itself.

References:

- Squire, D. (2017). *Design Patterns*. Lecture, Monash University.
- *Design Patterns Observer Pattern*. (2017). *www.tutorialspoint.com*. Retrieved 20 May 2017, from https://www.tutorialspoint.com/design_pattern/observer_pattern.htm
- *Design Patterns Adapter Pattern*. (2017). *www.tutorialspoint.com*. Retrieved 21 May 2017, from https://www.tutorialspoint.com/design_pattern/adapter_pattern.htm
- Martin, R. (2000). *Design Principles and Design Patterns* (1st ed.). http://www.objectmentor.com/. Retrieved from https://drive.google.com/file/d/0BwhCYaYDn8EgODUxZTJhOWEtMTZlMi00OWRiLTg0ZmEtZWQ5ODRlY2RmNDlk/view