FIT3077 Assignment 2 – Stage 2

David Lei (26029391) and Patrick Shaw (26898187)

Note: More information regarding the codebase can be found in the project's readme.md.

1. Design Patterns

1.1. Observer pattern

The observer pattern was used for on click events and in the `LocationMonitoringManager` and `SessionMonitoringManager`. The frontend observers allow decoupling between view components and the implementation of what happens when they are clicked [1]. The observer interfaces used in the assignment mimic the Android SDK's implementation of the observer pattern [2].

1.2. Factory pattern

We use the factory pattern to allow the `FullLambdaWeatherService` to create a `WeatherClient` from a `WeatherClientFactory` so that the service does not need to know the underlying steps to create the client. Retry-client-creation functionality will be written into the `FullLambdaWeatherService` as an example of using the pre-existing factory code to potentially build multiple `Promise<WeatherClient>`s.

1.3. Model-view-controller (MVC) architecture/pattern

The model, view and controller (as previously discussed) are separated in the codebase. The `WeatherPageContainer` determines how user interactions are interpreted as if it were a controller. All other `React.Component`s serve as view type classes. All business type logic classes are handled by the backend server. This allows, for example, view components to be used regardless of the implementation of the controller, without modification of code.

1.4. Adapter pattern

The object adapter pattern is used to wrap the new weather timelapse SOAP client and allow it to conform to the expected interface, `WeatherClient`, expected by `FullLambdaWeatherService`. This was done by making a new class that implements `WeatherClient` which wraps the soap client. This abstracted away the communications to the SOAP service, in turn, allowing the otherwise incompatible SOAP client to interact with the rest of the codebase as if it was a `WeatherClient` [3].

2. Design principles

2.1. Principle of composition over inheritance

Wherever possible, composition of classes was used in place of inheritance. This mentality is typically considered to enable more flexibility in code. This principle utilised in many modern libraries such as the Java Jigsaw library which is expected to be a native library in Java 9 [4]. An example of this principle was the choice to compose the `LocationItem` of a `GenericListItem` rather than inheriting from it.

2.2. Principle of dependency inversion

Dependency injection, specifically, was used to comply with the dependency inversion. As an example of the benefits of dependency injection in the code base; the `WeatherClientFactory` is passed into the `FullLambdaWeatherService`'s constructor. This is particularly useful for replacing fully featured objects with mock test objects in code [5]. For example, it enables the ability to swap out create backend server instances that use a `TestWeatherClient` instead of a `MelbourneWeatherClient` which allows for easier debugging. It also allows the ability to create two `FullLambdaWeatherService`s that use either the original SOAP service or the new time-lapse SOAP service without any modification to any classes.

2.3. Don't repeat yourself (DRY principle)

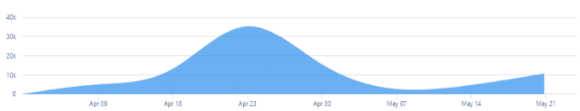
To abide by the concept of DRY, whenever code copying and pasting code in the codebase is an option, either functions or new classes were created to avoid code repetition. This allows better maintainability as modifications to code only needs to be made in a single place. For example, originally, the WebSocket frontend end points for temperature and rainfall monitors were not using the same code, but essentially executed the same functionality; thus, a class was created called `MonitoringConnection` was created to wrap this initialisation and handling of the WebSocket code, in turn, minimizing code repetition.

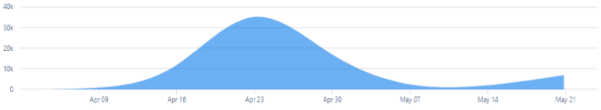
3. Proof of design

The following section provides statistic evidence over the stability, extensibility and maintainability of the codebase as evidence of the effectiveness of the previously mentioned design patterns and principles.

3.1. Git additions and deletions over time

The following graphs represent the number of lines inserted and deleted, respectively, over time. It is evident that in stage 2 (2 May) minimal code was added and removed as the most stage 1 code could be extended to comply with the stage 2 requirements.





3.2. Class design statistics

The following table depicts the fact that most classes added to the project were related to the frontend codebase. This was a necessary for the creation of the chart monitor and Google Map bonus mark UI. Thus, most classes created to comply with UI requirements rather than attempts to fix design issues. The table does not account for interfaces and anonymous classes.

	Stage 1	Stage 2
Frontend classes added	13	9
Frontend classes removed	0	0
Backend Classes added	17	3
Backend classes removed	0	0

References

[1] A. Shvets, G. Frey and M. Pavlova, "Observer Design Pattern," [Online]. Available: https://sourcemaking.com/design_patterns/observer. [Accessed 9 May 2017].

[2] "Android Developers," Google, [Online]. Available: https://developer.android.com/guide/topics/ui/ui-events.html#EventListeners. [Accessed 9 May 2017].

[3] A. Shvets, G. Frey and M. Pavlova, "Adapter Design Pattern," [Online]. Available: https://sourcemaking.com/design_patterns/adapter. [Accessed 11 May 2017].

[4] G. Lagorio, M. Servetto and E. Zucca, "Featherweight Jigsaw — Replacing inheritance by composition in Java-like," *Information and Computation,* vol. 214, p. 86, 2012.

[5] *DAGGER 2 - A New Type of dependency injection.* [Film]. America: Google, 2014.

[6] A. Shvets, G. Frey and M. Pavlova, "Abstract Factory Design Pattern," [Online]. Available: https://sourcemaking.com/design_patterns/abstract_factory. [Accessed 10 May 2017].