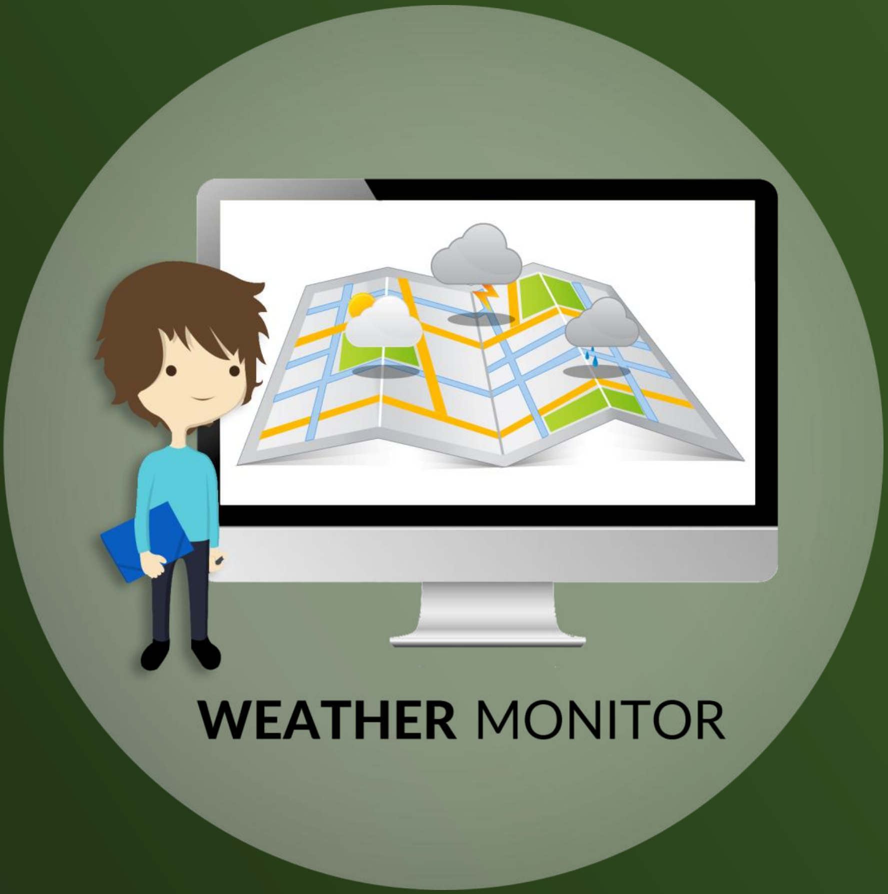
MONASH University

FIT3077 Software Engineering: Architecture and Design

Team Software Architecture Assignment

Stage 2 - Design Principles and Pattern



Semester 1, 2015

Team "CodeIneNoobs":

Arvin Wiyono (awiy1 - 24282588)

Darren Wong (dwwon6 - 25147064)

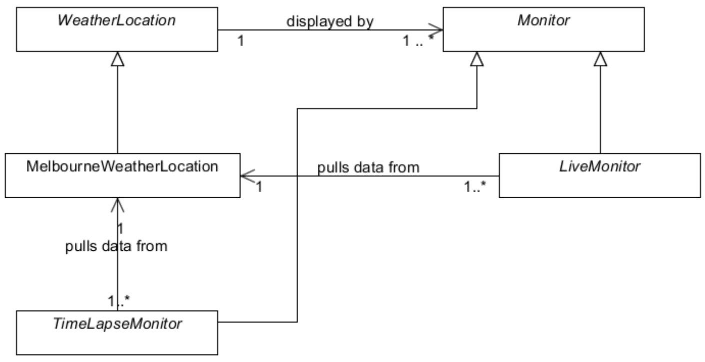
Design Decision Explanation

There are several design patterns and principles that have been applied to construct the Weather Monitor application. The design decisions were carefully considered so that it can be extended in the later stage of the assignment. It is expected that the system has both flexibility and stability in a balance manner. Extending upon our old system in Stage 1, there have been additional classes that are implemented and code refactoring to existing classes to provide a robust system.

Applied software design and architectural patterns:

A. Observer pattern

The core design for this application is derived from the *Observer* design pattern to implement the given requirements. The rationale behind the use of this pattern is that observer pattern enables a subscriber to receive updates from an observed provider (Observer Design Pattern, 2015). From this point of view, this is suitable with the application, that is whenever a WeatherLocation (provider/subject) changes its data after an update, it immediately sends request to the Monitors (subscriber/observer) to update its status. Below is a diagram which represent portions of the system, in which the pattern is applied:



Applied observer pattern in the Weather application

From the above diagram, it is evident that pull model is chosen instead of push model, i.e. Monitor subclasses pull data from the observed WeatherLocation. Pull interaction was considered since it is more efficient in the case as we have a lot of Monitors (worst case scenario) for a particular WeatherLocation at the same time. This frees up the WeatherLocation from the burdens and responsibility of pushing the data to an excessive number of Observers. It is also more flexible since Monitor subclasses can request the necessary data which it only needs. However, there is a tradeoff behind these benefits: WeatherLocation and Monitor become tightly coupled with 2-way navigability.

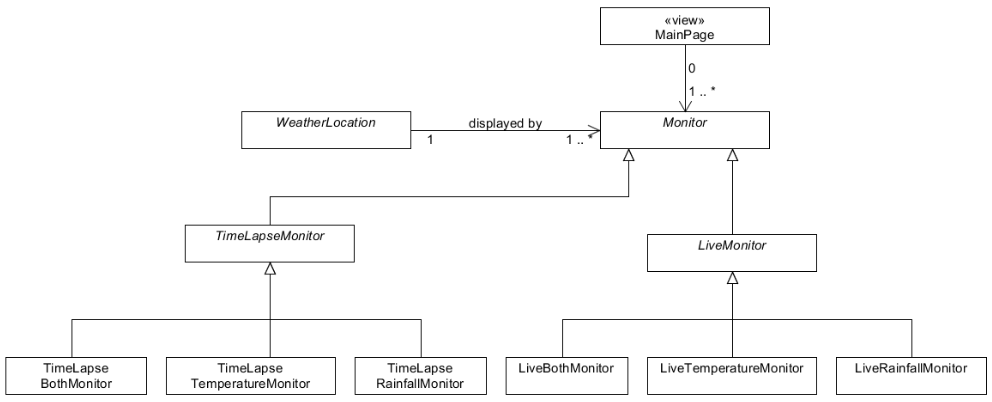
B. Adapter pattern

The adapter pattern enables flexible data transfer carried between a SOAP server and a WeatherLocation . By flexible means that the behaviour and services provided by the real server are not directly visible to the WeatherLocation and consequently, any behaviour changes can be solved in a more isolated manner by refactoring the WeatherAdapter and no recompilation of WeatherLocation class (and its subclass) needed since it depends on the methods provided by WeatherAdapter superclass.

The real benefit of this pattern can be gained during the addition of a new server. The WeatherAdapter handles all the information from both the service providers and becomes the medium of data delivery between a specific server and WeatherLocation (Adapter Design Pattern, 2015). Update timer interval can also be obtained by requesting the WeatherAdapter via the Controller class.

C. Abstract factory pattern and LSP principle

Abstract factory pattern in the system design infrastructure is applied to the implementation of different types of Monitor. As the requirement specifies that the user can select to view the temperature and / or rainfall weather info, this pattern is able to tackle this variety of Monitors since the clients (classes which rely on its services) depends on the abstraction instead of concrete classes (Alexander, 2013). Below is a diagram which depicts the pattern implementation:



Abstract factory pattern in the Weather application

The above diagram clearly shows that WeatherLocation and the view layer MainPage class depends on the Monitor abstract class, which serves as a blueprint and encapsulates the complex logic of its subclasses. Again, flexibility is the main element that must be achieved in this case and the implementation of TimeLapseMonitor to fulfill the Stage 2 requirement proves that system design is indeed flexible and can be extended.

Furthermore, one potential problem that needs to be avoided with abstract factory pattern is the violation of Liskov Substitution design principle, which states that subclass should not limit the 'power' of its superclass. Based on this consideration, it is required to separate the responsibilities and distribute them to subclasses which focus only in one functionality without ignoring LSP. This is why LiveMonitor and TimeLaspeMonitor abstract classes each has BothMonitor (displays both temperature and rainfall), TemperatureMonitor (temperature only) and RainfallMonitor (rainfall only) subclasses.

D. Model-view-controller architectural pattern (Passive)

We introduce the use of **Controller** class which performs and takes care of any logical thinking behind the GUI. This will result in separation of concern between the GUI and the classes. By doing this way, we can replace the GUI with any kinds of user interface, yet will still yield the same result since both UI and Controller are independent. The organization of the system is based on the *Model-View-Controller* pattern.

Next, it can be seen that WeatherAdater, WeatherLocation and Monitor must be visible to the WeatherController abstract class. This is because it has the responsibility to handle the functionalities of the system. To enhance its cohesion, subclasses namely LiveWeatherController and TimeLapseWeatherController are created, which each depends on a particular subclass of WeatherAdapter. The difference between the two is which monitor object is created. As the name suggest, LiveWeatherController creates LiveMonitor subclasses objects and similarly, TimeLapseWeatherController creates TimeLapseMonitor subclasses objects. The re-use of WeatherController abstract class also tells that the Controller's design is very flexible.

Moreover, the passive MVC is chosen instead of active. This is because of the VB API which supports how each Control (Label/Chart) in the Main Page to be associated with a data structure, which in this case is a Monitor object by using its *Tag* property. This eases the update functionality of the system since Control can get the latest status of its Monitor by invoking the feedData() method. Therefore, 'Model is inactive' and it does not need to notify the View when it is changed by Controller (Basher, 2013).

References

- Adapter Design Pattern. 2015. *Adapter Design Pattern*. [ONLINE] Available at: http://www.sourcemaking.com/design_patterns/adapter. [Accessed 24 May 2015].

- Observer Design Pattern. 2015. Observer Design Pattern. [ONLINE] Available at: https://msdn.microsoft.com/en-us/library/ee850490(v=vs.110).aspx. [Accessed 24 May 2015].

- 2013, Alexander, C., Ishikawa, S. and Silverstein, M., A Pattern Language: Towns, Buildings, Construction, Oxford University Press, 1977, lecture notes, FIT3077 Software Engineering: Architecture and Design Monash University [Delivered 16th March 2015].

- Basher, K. 2013. MVC Patterns (Active and Passive Model) and its implementation using ASP.NET Web forms - CodeProject. [ONLINE] Available at: http://www.codeproject.com/Articles/674959/MVC-Patterns-Active-and-Passive-Model-and-its [Accessed 24 May 2015].