Written explanation of design principles

**Introduction**

In this document, we discuss all the different principles we used for our program. The reason we used them is because we found that the best way to easily and seamlessly demonstrate and use the program was to use these principles. We decided to use the following principles as it made sense to do things such as using the observer patterns because it made it better to reduce traffic. The use of principles such as the open-closed principles also ensured that we did not have to have useless classes. We have implemented all these principles in the hope that we have managed to properly use the SOLID principles and managed to ensure good coding standards…

**Observer Pattern:**

The observer pattern is used when there is a one-to-many relationship between objects such as if one object is modified, its dependent objects are to be notified automatically with the use of one the following methods; *NotifyObserver*(), *RegisterObserver(),* and *RemoveObserver()*. We created an interface class *IObserver* and an interface class *ISubject*. Whenever anything changes, such as rainfall or temperature, or an object is added, the *ISubject* class will notify the *IObserver* class. Once it is notified it will make the necessary adjustments or changes based off the information gained when notified by the *ISubject* class. So, an example of why we used this pattern, is if we opened 4 monitors of the same location, we do not want each monitor to update separately as this will be too taxing, bandwidth wise, So, with an observer, if one monitor updates, the others will update according to the monitor they are watching.

**Open-Closed Principle:**

*“software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification”*. With the use of inheritance and polymorphism we could incorporate this principle. In doing so it made it easier for us to fetch data and made our code a lot less cluttered and easy to understand and read. For example, having the *Temperature* class inherit data from the *WeatherData* class.

**Liskov Substitution Principle:**

This principle is just an extension of the Open Close Principle and it means that we must make sure that new derived classes are extending the base classes without changing their behaviour. An example of this in our code is how we extended the *AllData, Monitor, Rainfall* and *Temperature* classes were extending the *WeatherData* class*,* yet not changing its behaviour in any way. These classes are using the getData() method from the *WeatherData* class. The *TimeLapse* class is extending the *Service* class and using its getLocation() method yet not changing its behaviour.

**Dependency Inversion:**

By using abstract classes and interfaces to implement the observers and subjects we are ensuring that the high-level modules are not dependent on the low-level modules, they are both dependent on the abstractions. The abstractions define the details of a class such as the *IObserver* and *ISubject* classes define what the classes that inherit from them will do.

**Common Closure Principle:**

This principle we used with the use of the observer, so if one class, for example, the *Temperature* class needed to be removed, then the observer would have to change, the *updateTemperature*() method would have to be removed. Similarly, for the *Rainfall* class, if it were removed, its corresponding method in the observer would also be removed. Within the System we have the observers and subjects are all within the same package so the changes would be limited.

**Interface Segregation**

This principle is used in the way that instead of having one fat interface with everything in it we made a few interfaces. This way things don’t depend on one interface if they don’t need to use it. So now for example, when monitor wants to display it only uses the *Display* interface and no other class uses it, because no other class needs to use it. So this is how we have implemented interface segregation.

**Acyclic Dependency Principle**

This principle has been implemented in the way the information received from the services is handled. Even if the code from the services is changed there would not be any fallout within the code since everything is abstracted and is references the data as an object rather than a specific variable.

**Stable Abstraction Principle**

Within the code there are stable classes that should not change such as *WeatherData*, *iSubject, Service.* There are many classes that depend on them making them stable. At the same time there are also classes that are unstable, these are the classes that will need to be changed over time, such as *Monitor, Rainfall* and *TimeLapseData.* The stables classes are the ones at the bottom allowing for OCP.

**References:**

Open closed: <http://joelabrahamsson.com/a-simple-example-of-the-openclosed-principle/>

Liskov - <http://www.oodesign.com/liskov-s-substitution-principle.html>

Interface Segregation - <http://www.oodesign.com/interface-segregation-principle.html>

Common closure - <http://wiki.c2.com/?CommonClosurePrinciple>

Dependency inversion - <http://www.oodesign.com/dependency-inversion-principle.html>