# Design Principles Followed

Some of the core design principles that were adhered to are listed below:

## Open Closed Principle

## Description

The open closed principle states that “software entities should be closed for modification but open for extension”, such that the behavior of entities can be extended without having to modify the entity itself, (Martin, 2000).

## Reason for implementing

The open closed principle is perhaps the most important principle in software architecture, as new functionality is added to software entities more often than not. For this reason, this principle fits very well in our project as it allows us to add more monitors without having to change the existing ones, as required. Also, the new functionality uses the existing code as a hinge point for its propagation.

## Principles of package Architecture

As classes are an inefficient way of organizing a design, we used a number of package design principles to organize the entire project into subsets of cohesive and reusable packages. Some of those are the common closure principle, the common reuse principle, and the acyclic dependency principle, which follow the idea of grouping related modules and managing the relationships between them.

# Design Patterns Used

# Abstract Factory

## Description

According to (Partha, 2004) an Abstract factory can be used when a client class needs to create an instance of concrete class without having to know exactly which concrete class it wants to instantiate. In other terms, an abstract factory provides an interface for creating objects.

## Reason for Implementing

The requirement for this assignment was to create monitors monitoring “Rainfall or Temperature or Both” at different locations. We used this pattern because it allows us to create objects of different types in a consistent way, while hiding the implementation from the entities that call for their creation. This is key to our system because we create objects of varying types from one controller, in their appropriate factories. This also allows us to adhere to the principles of grouping entities based on their functions. For example, a controller will call for the creation of an object without having to worry about how it is done or in fact, what type of object it is.

## Drawback of the Pattern

When using the abstract factory, we can only use the same types of concrete classes as we defined in the abstract factory. However, if a need for a different type of object arises, the factories can just be extended by an adapter without breaking the open closed principle.

# Observer Pattern

## Description

In this pattern, there are many observers (objects) which are observing a particular subject”, (Vaskaran, 2016). As an anecdote, observers are basically interested and want to be notified when there is a change made inside that subject (what they are observing). When they lose interest in the subject they simply unregister themselves. This creates a one-to-many relationship between observers and their subject.

## Reason for Implementing

We implemented the observer pattern because it provides us with a consistent communication model between our locations and their different types of monitors. Despite the code-wise complexity of implementing this pattern, it is still worth implementing, since we were specifically asked to reduce network traffic to the services. Thus, using the observer pattern allows us to notify all monitors once any change happens to the subject leading us to only accessing the service once if there is more than one type of monitors for that service.

## Drawback of the Pattern

The observer patterns broadcast the changes experienced by the subject. However not every observer is interested in all the information about the subject. Thus, we may be sending out useless information at times. To get around this, we did some validation in the entity that notifies the observers to check their type and determining what they need before updating them.

# Adapter

## Description

Martin, 2000 describes an adapter as an object that implements an abstract interface to delegate to the server (usually a third party server, as in this assignment). This pattern is used in cases where inserting an interface to deal directly with a server is infeasible. In essence, it is use to adapt an entity to be used with another.

## Reason for Implementing

(Bipin, 2016) states that the adapters can be used to accomplish the following:

1. Control and grant access to an object

2. Map calls intended for a class to another class with a different design

3. Add new functionality or features to existing objects

4. Utilize a small number of objects to serve a large number of requests

5. Simplify access to a complex subsystem.

Our System had to allow for an extra service. Since we used an abstract server, the client only communicated with the server interface which had certain method signatures that had to be implemented by all services. Thus, we opted for option to create an adapter for the Melbourne Time Lapse Service. That would then be connected to the service interface and all conversions (Kelvin to Celsius cm to mm) would be handled by the adapter. This ensures that we do not break the open closed principle.

## Drawback of the Pattern

The adapter pattern unnecessarily increases the size of the code as class inheritance is less used and lot of code is needlessly duplicated between classes as stated by (Roshan, 2013). Furthermore, adapting a class to fit an unsupported interface may be a very challenging process depending on the complexity of the system.

# Abstract Server

## Description

When clients directly depend on a server, the dependency inversion principle is violated, as changes made to the server will propagate to the client, which will in tandem make it hard for the client to use other servers. This pattern allows the creation of an abstract server class which concrete service classes must extend. This helps to maintain the relationship between the client and the different services it needs.

## Reason for Implementing

The main reason for implementing this pattern was to create a hinge point in our system allowing us to extend our system in the future by adding additional services without having to change code in other classes. Furthermore, it also permitted us to adhere to the Dependency Inversion Principle (DIP) to reduce coupling in our code and make system extension simpler.

## Drawback of the Pattern

On its own, abstract server makes extension possible but limited to services that have a similar behavior as the one it was initially intended to abstract.

# References

Bipin, J. (2016). *Beginning SOLID principles and design patterns for ASP.NET developers.* New York: Springer .

MSA. (2013, February 25th). Principles of Object-Oriented. *Lecture Slides*. Faculty of Information Technology.

Martin, R. C (2000). Design Principles and Design Patterns. Retrieved from: www.objectmonitor.com

Partha, K. (2004). *Software architecture design patterns in Java.* Boca Raton: Auerbach Publications.

Roshan, A. (2013, 10 12). *Adapter Design Pattern*. Retrieved from javarevealed: https://javarevealed.wordpress.com/tag/adapter-pattern/

Vaskaran, S. (2016). *Java design patterns : a tour of 23 gang of four design patterns in Java.* New York: Springer .

**s**