# 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

In our Assignment requirement, we were specifically asked to create monitors monitoring “Rainfall or Temperature or Both” at a location. Keeping that in mind, (Partha, 2004) states that abstract Factories are “useful for plugging in a different group of objects to alter the behavior of the system”. Furthermore, she also states that this pattern permits us to create objects in a consistent way. Object creation is an import aspect of our system thus keeping that consistent and abstract would be key to our system’s architecture Which is exactly what an abstract factory does. In addition, this pattern allows us to use dependency inversion to make code

## 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 ever that eventuality was to occur it could be overcome by using the Adapter pattern to adapt a new type of concrete class to the current abstract factory.

# Observer

## Description

(Vaskaran, 2016) “In this pattern, there are many observers (objects) which are observing a particular subject (object). Observers are basically interested and want to be notified when there is a change made inside that subject. So, they register themselves to that subject. When they lose interest in the subject they simply unregister from the 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’s interested in all the information about the subject. Thus, we may be sending out useless information at times.

# Adapter

## Description

(Bipin, 2016) Describes structural patterns as patterns that focuses on how classes and objects are composed so as to create a larger and more complex structure. The name of this pattern suggests its use. It is used to adapt one class to 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 allowed us to our service interface open for extension but closed for modification.

## 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

Very simple pattern that basically creates an abstract server class which is the concrete service classes must implement and this abstract class maintains the relationship between the client and the different services.

## Reason For Implementing

The main purpose 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 implement 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.

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 .