OOD design principle

Open/Close Principle, classes should be open for extension, but closed for modification (Martin, 2003). We have applied this principle by ensuring we have no global variables and ensured that member variables have been made private through the use of encapsulation. We, therefore, have prevented our classes from depending on such variables and have closed our classes against such dependencies. We have also applied this principle by providing abstraction where we expect the system is likely to experience change (weather monitors(data), weather services and monitors) As a result we have created an abstract weather service class and general monitor as well as weather monitor interface. By creating these we have fixed the modules and closed them against modification. We have then, used inheritance and polymorphism to override the abstraction to extend the classes and so support the open-closed principle.

Interface Segretragation Principle

This principle states that clients should not be forced to implement interfaces they don' use. Therefore, instead of having interfaces that list many methods rather have a groupings of interfaces specifying one or a few methods. We have applied this principle with the interface, Subject and weather monitor, and abstract factory interface as these interface specify only methods used by their respective client. They are also not considered a fat interface as they implement only one or two methods, In the case of the weather service interface we initially violated this principle for the purpose of implementing the Abstract factory. We forced for example the Rainfall class to depend on the weather service interface where it only used one of the methods specified by the interface. However we were able to rectify this by changing the weatherServive class to an abstract class and creating three new interfaces iRainfall , iTemperature and iTime that the abstract weatherService class implemented . This then resulted in the rainfall monitor class only depending on the iRainfall interface to request data from the weather service and so we had applied the ISP. We made use of casting to support this.

The Liskov Substitution Principle, where reference to a base class is made, it can point to any sub-class without distinction (Martin, 2003).. We have utilised this principle extensively. If you look at our monitor classes, we have and aggregation of weatherMoniotor we can therefore update and display any weatherMointor without knowing its type i.e rainfall or temperature. By using this principle , our system could also incorporate adding many more weather monitors for example humidity , surf conditions and wind speed therefore supporting extensibility and OCP .

ADP:
The dependency structure between packages must not contain cycles (Martin, 2003). We have broken cyclic dependencies by implementing ISP and DIP. Every package within the system is accessed through an interface or abstract class within that package. A clear example where we have implemented this is with the weatherData and weatherServices. The weatherData classes contain the interfaces which are implemented in the weatherMonitor classes, therefore, removing the cyclic dependency, using ISP and DIP

Stable Abstractions Principle(SAP)
Most of our packages are stable except the controller package. We should have therefore designed our system better to reduce the instability of the controller package as it depends on many other packages and is only depended on by one package.

Design Patterns Used:

Abstract Factory: In our design we used an abstract factory to apply DIP. An abstract factory is used to create weather monitors for weather data(rainfall, temperature and time), web services and monitors. Therefore, we require only to interact with the Abstract factory interface in order to create concrete objects. Through the Abstract Factory interface, we also provide a creational interface to the packages of weatherData Services and Monitors. By creating a weatherData package for the weather data factory and Services package for the service factory and a monitor package for all monitors we have applied the common closure principle, these classes in the above-mentioned packages are likely to change to be changed outside of the abstraction (i.e the factory classes in order to add more items of the factory) and therefore belong in the same package. Concrete Factories have been made singletons, this had been done as we one we only ever need one instance of a factory throughout the system.

Observer
We have used an observer pattern for updating all the monitors whether it be a graphical or textual monitor. This pattern was chosen as it would allow us to add many different kinds of monitors and new types could also be easily added in the future, therefore, applying the open-close principle. By implementing this pattern we also applied the Liskov Substitutability Principle, i.e. that as long as a monitor implements the observer interface any monitor type can be updated using the same methods.

Adapter

An adapter has been used as the addition of the MelBourneTimelapse service methods did not comply with the web service interface for the service factory we had created as the return type and data type was specified differently i.e. temperature was given in kelvin and getWeather returned a (Martin, 2003)string containing temperature rainfall and time. By using the pattern, we were able to add the new service and conform it to the interfaces specifications without requiring modification to the system. By using the adapter pattern we were able to maintain the OCP principle.

References

Martin, R. C. (2003). *Agile Software Development, Principles, Patterns, and Practices 1st Edition.* Pearson.