# Experiment In Analysis
CPSC-60000-002

I'd like to open this analysis by stating formally that digging through this repository to find the important Java files has been a substantive exercise in perseverance.

The first design pattern that I managed to stumble upon was the DataAccessFactory, representing an obvious example of the Factory Pattern. The various implementations of this Factory included the MongoSchemalessDataStoreFactory, the JDBCOpenSearchAccessFactory, and the SmartDataLoaderAccessFactory. I am familiar with creating an abstract factory to avoid concrete implementations of many similar classes, but unfamiliar with using this pattern to access data. I did some digging into this use case and found that this specific implementation of the factory pattern overlaps with another pattern, the Data Access Object Pattern[1]. "The Data Access Object (DAO) pattern is a structural pattern that allows us to isolate the application/business layer from the persistence layer (usually a relational database but could be any other persistence mechanism) using an abstract API." In essence, each of the data access factory implementations provides access to a different data store, abstracting the logic of accessing data from individual stores. The use of this pattern should help create a more consistent way to interact with data from various sources so I think it probably helps in this project, where the goal is to create a conglomerate of data from various sources.

The next design pattern that I found is the Observer / Listener Design Pattern used in the LoggingStartupContextListener. I haven't worked with servlets up to this point so I had to give them a Goog for the sake of context. The way that I understand things, a servlet is used to add to the functionality of a server[2]. (Sidenote. Looking through the logging code, I found a logging factory that I am going to dig through.) Based on the logging wiki[3] this listener is used to redirect logging from the observed servlet to commons-logging. The listener takes the servlet context as input and attempts to reconfigure its logging. The factories used in the listener class are the CommonsLoggerFactory and the Log4JLoggerFactory; from my understanding of the wiki, both of these likely similar methods to take advantage of commons-logging.

The class WrappingFeatureListener is used to wrap a FeatureListener class and alter the source of the original listener. I don't know that this is a specific design pattern that we've touched on; it looks like a decorator that extends the original listener object to be capable of using different FeatureEvent sources in the future. Although it doesn't directly affect the methods available, it does extend the functionality of the Listener to different event sources, like an adapter. Again, it differs in that it alters the Listener instead of extending it. It seems to me like this was completely unnecessary. The class takes a couple arguments and modifies the

[1] https://www.baeldung.com/java-dao-pattern
[2] https://docs.oracle.com/javaee/5/tutorial/doc/bnafe.html#:~:text=A%20servlet%20is%20a%20Java,applications%20hosted%20by%20web%20servers.
[3] https://github.com/geoserver/geoserver/wiki/GISP-13

original listener that was passed in. It would be just as simple to add a method to the class and make the updates via a call to that instance method.

Next up we have numerous decorator methods that are also frequently paired with a factory: PlacemarkStyleDecorator, PlacemarkDescriptionDecoratorFactory, PlacemarkSelfLinkDecoratorFactory, PlacemarkNameDecoratorFactory, LookAtDecoratorFactory, ExtendedDataDecoratorFactory, LegendDecoratorFactory, and tens more decorator factory classes throughout the repository. Many if not most of these decorator factories are located in the 'kml' subsection of the repo. According to Wikipedia via Google, KML is Keyhole Markup Language, which is used "for expressing geographic annotation and visualization within two-dimensional maps and three-dimensional Earth browsers."[4] From what I gather based on the above, and some more digging into the code, is that this KML is what enables things like legends, labels, ids, and other descriptors to be displayed on a map. The KmlDecoratorFactory is the base class for the other factories, and allows access to a specific feature in the KML. The decorators then each alter a specific feature, like PlacemarkNameDecoratorFactory, which obviously allows access to and edits the Placemark Name feature. This is used directly in and xml file in a <bean> tag, which apparently forms the backbones of an application.[5] These classes allow the programmers to access xml code through the use of Java. The implication here would be that by using these decorator factories and their beans, the developers can alter the annotations and other on-screen information. I don't really see the purpose of each of these sub-factories being a factory itself, as it isn't used by some other class as a way to abstract away the implementation details of another method. That is, unless the class being used in the xml somehow allows future developers to take advantage of the class in some way. The scope of the project and my lack of experience with Java aren't conducive to a great understanding of how each file fits together. The naming of KmlDecoratorFactory does make sense to me, as it is used to abstract away actually accessing and setting the kml property.

Digging through this repo has not been a joyful experience for me. It's clear that the authors were hellbent on using the object-oriented design patterns that they were aware of; all of them, and everywhere. Although there are clearly some areas where it makes sense to have these design patterns in place, such as the listeners on the servlets discussed early on, and potentially the data loader factories that are able to access different data sources, a large number of these implementations were unnecessary. The goal of this course, in my mind, is to have a knowledge of various patterns that might end up being the best solution to an architectural problem that we have. It is unlikely that whatever we are trying to design has not been thought of and solved via some elegant pattern. That being said, some things are simple enough to not need a specific pattern, and following good object-oriented programming practices will do. Although the repo being analyzed likely works just fine, the authors

[4] https://en.wikipedia.org/wiki/Keyhole_Markup_Language#:~:text=Keyhole%20Markup%20Language%20(KML)%20is,originally%20named%20Keyhole%20Earth%20Viewer.
[5] https://www.tutorialspoint.com/spring/spring_bean_definition.htm

introduced unnecessary complexity by insisting upon the use of design patterns for every solution.

Overall this exercise has been a cautionary experience for me in overusing the things that I am learning in this course. When all you have is a hammer, everything may look like a nail, but that doesn’t mean you should hit it.