**CST-361 Design Patterns in Java**

**Student** **Name**: Michael Weaver

**Project** **Title**: Activity 8: Antipatterns and Best Practices

**Date**: 02/17/2019

**GitHub URL**: <https://github.com/battousairurik/CST361_Design_Patterns_In_Java>

Antipattern – The God Object

The God object is basically a single class that has more and more functionality as the project progresses. Typically beginning as a utility class, a developer would continue to add odd forms of utility to that class until it does everything from concatenating strings to merging files (Daxx Team, 2016).

The God Object is an antipattern because it violates the rules of object oriented programming. The God Object violates encapsulation because ***everything*** is part of the god object. It violates the idea of inheritance, because it is all inherited classes in itself. The God Object is not polymorphic, it simply is all methods ever possible. It also violates abstraction since you need direct access to the object to do whatever the object does.

The best way to work around the God Object to is to design models specific to what they are intended to do. If you need utility functions for two separate classes, then use two utility classes. This also assist in making the code more readable as you know exactly the function of the class by its name and functions.

Antipattern – The Stovepipe System

*“The Stovepipe System AntiPattern involves a lack of coordination and planning across a set of systems.The key problem in this AntiPattern is the lack of common subsystem abstractions, Subsystems are integrated in an ad hoc manner using multiple integration strategies and mechanisms.All subsystems are integrated point to point, thus the integration approach for each pair of subsystems is not easily leveraged toward that of other subsystems.Furthermore, the system implementation is brittle because there are many implicit dependencies upon system configuration, installation details, and system state. The system is difficult to extend, and extensions add additional point-to-point integration links (Sourcemaking.com, 2019).”*

A few means of circumnavigating the stovepipe antipattern are to utilize abstraction, use of common unified architecture throughout the project, and use of loose coupling of components.

Best Practice - MVC

MVC is the separation of business logic (Java beans and EJB components), controller logic (servlets/Struts actions), and presentation (JSP, XML/XSLT) (Botzum, 2007).

Model-View-Controller (MVC) is fundamental to the design of good Java EE applications. It is simply the division of labor of your programs into the following parts:

* *Those responsible for business logic (the Model -- often implemented using Enterprise JavaBeans™ or plain old Java objects).*
* *Those responsible for presentation of the user interface (the View).*
* *Those responsible for application navigation (the Controller -- usually implemented with Java servlets or associated classes like Struts controllers).*

MVC is a technique that I have been practicing since nearly day one, so it is something that I fall back on more often than not. Seeing how effective this practice is, I am likely to maintain its use going forward.

Best Practice - Develop to the specifications, not the application server

Know the specifications by heart and deviate from them only after careful consideration. Just because you can do something doesn't mean you should.

For instance:

* *We find developers dig themselves into a hole by trying something that they think will work "a little better" than what Java EE allows, only to find that it causes serious problems in performance, or in migration (from vendor to vendor, or more commonly from version to version) later (Botzum, 2007).*

Developing to specifications is absolutely necessary. I’ve experienced this in almost every one of my classes. You will always be on a deadline and if you attempt to create something far grander than needed, you will pass that deadline and fail to submit the project. This practice also circumnavigates a number of other problems such as the use of new technologies not fully supported or custom services that have migrational risks.

References

Daxx Team. (2016). *Software Architecture AntiPatterns Explained*. Retrieved from <https://www.daxx.com/blog/development-trends/software-architecture-antipatterns>

Sourcemaking.com. (2019). Stovepipe System. Retrieved from <https://sourcemaking.com/antipatterns/stovepipe-system>

Botzum, Keys. Brown, Kyle. Willenborg, Ruth. Wong, Albert. (2007). The top Java EE best practices. Retrieved from <https://www.ibm.com/developerworks/websphere/techjournal/0701_botzum/0701_botzum.html>