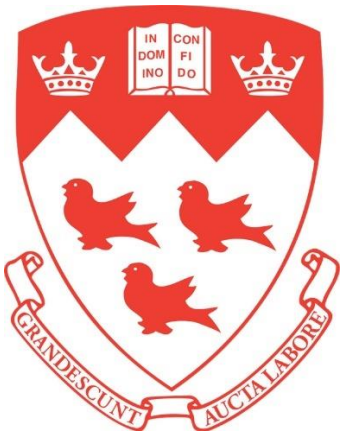
McGill University



ECSE 321 – Project #3

Architecture and Design

Prepared by:

Brent Coutts (260617550)

October 28th, 2016

1 Architectural Styles

1.1 Match each of the following descriptions with the most appropriate architectural style:

i. A style that limits the maximum number of subsystems known to any subsystem in the system to one subsystem.

Repository Architecture.

ii. A style that views the system as a series of sequential or concurrent data transformations.

Pipe-and-Filter Architecture.

iii. A style (or a variant of it) that is commonly used for web applications.

Client-Server Architecture.

iv. A style that decomposes the system into subsystems according to the presentation, interaction, and representation of system data.

Model View Controller (MVC) Architecture.

v. A style that allows redundant functionality to be provided by different subsystems to increase the system's dependability.

Layered Architecture

1.2 List an advantage and disadvantage of each of the five styles.

1.2.1 Model View Controller (MVC) Architecture

An advantage of the Model View Controller architecture pattern is that it is efficient when the project needs to be implemented on multiple platforms. It is simple to reuse the model along with a different view and controller to provide alternate user interfaces.

A disadvantage of this architecture pattern is that changes to the model interface will require both the controller and the view to be updated in order for them to operate with the new model. This relationship causes extra work than would be required by other patterns due to different component separations.

1.2.2 Layered Architecture

The Layered architecture pattern has an advantage of allowing programming teams to each work on a separate layer. The interaction required between the teams is minimal and this allows each to be highly specialized and proficient at their own layer.

One of the disadvantages of the Layered architecture pattern is that performance may be sacrificed. The performance will be affected by the fact that an upper layer must pass through multiple unnecessary layers if it is necessary for it to interact directly with a lower level layer.

1.2.3 Repository Architecture

A Repository architecture pattern provides the advantage of component independencies. The components have no direct interaction with each other. This allows for each component to be updated separately and the changes will be automatically propagated to all of the other components through the repository.

It is difficult to adopt a new data model when using this type of architecture. The new data model has to be applied on the entire repository and this also requires all of the subsystems to be updated.

1.2.4 Client-Server Architecture

The Client-Server architecture pattern has a highly prominent advantage of allowing the servers to be distributed across a network. This means that all clients/users are able to access common functionalities. All services are not required to implement each general functionality.

The reliance of this architecture pattern on a network also leads to one of its disadvantages. It is difficult to predict the performance when using this pattern and it depends on both the performance of the network and the system itself.

1.2.5 Pipe-and-Filter Architecture

The simplicity of the Pipe-and-Filter architecture pattern is a clear advantage. The designer can clearly understand the overall behavior of the system in terms of what the individual filters input and output.

A disadvantage present in this pattern is that data transferred between each of the filter transformations must be consistent. The input and output must be parsed and unparsed to an agreed form. Due to this fact, it is potentially impossible to reuse functional transformations if the transformation does not use a data structure that is compatible in the new system.

1.3 For each of the following systems, list an architectural style that would best represent the core behaviour of the system. Justify your decision!

i. Processing of Insurance Claims

The most representative architecture pattern for the processing of insurance claims would be a Repository architecture pattern. The reason for this choice is that insurance claims have a large number of factors that influence their processing. If the processing was done in one location, it would be found that contradictions exist due to the large amount of different rules and situations for the claims. A repository architecture pattern allows the repository to be the core system for the claims processing. This repository can then have subsystems that can be called if they are relevant to the claim of interest. For example, insurance protocols are often different in each municipality or province. Thus, the repository can call a subsystem containing the relevant claims processing to that province without seeing any other province subsystems. This pattern also allows for simple alterations should the claims in a province change.

ii. Linux

Linux is an operating system. The architectural style that would best represent the core behavior of Linux, is the Layered architecture pattern. In this pattern, the system is organized into layers which are layered on top of each other. Each layer builds upon the last, starting from the most basic functionalities in the innermost layer. This pattern is good when the system is highly complex. Separating the problem into multiple layers allows each layer to be handled in a more systematic way and often by separate specialized teams. The layered format also allows for entire layers to be replaced, if necessary, without affecting the system. There is an enhanced measure of security in this pattern as well, due to the multiple layers. The most important functionalities can be placed in the innermost layers, protected by multiple security checks in the outer layers. Additional layers can be continually added to Linux as further functionalities are required. This allows Linux to constantly improve as an operating system by building upon its existing system. These advantages all highly benefit the Linux system, therefore making this architecture pattern the best for Linux.

iii. Twitter

Twitter would suggest the Client-Server architecture pattern. This pattern is used when clients desire to access the database from a range of locations. Also, due to the capability of server replication, this pattern allows Twitter to deal with the variable demand exacted upon it. The basic functionality of twitter is to view 'tweets' from other clients around the world. It needs to be possible for the client to access to the server in order to see and post tweets. The server is a large database storing all of the tweets and other data contained within Twitter. The client only requests the data of interest to them from the database and can actually borrow some of the processing power from the server. In this architecture, the functionality is available to any client with an internet connection. However, the client does not need to implement all, or any, of the functionality. Due these reasons, the web based application of twitter would most resonate with the Client-Server architecture pattern.

iv. Event Registration system from Assignment #1

The architectural style of Model View Controller (MVC) would be a wise choice for this event registration system. The justification for this choice is based upon the fact that the program was required to run on multiple platforms (android, desktop, and online) without interactions between each of the platforms. In this pattern, the model is able to be reused on each platform. It is also possible to export a .jar file of the Java code to the android program. This architecture pattern has a larger amount of initial work than some others, but it has the benefit of being relatively simple to transfer to different platforms. The view and controller for each platform were not extensive. This further validates this pattern choice as each separate platform only requires the reasonably minor task of updating the view and controller.

2 Design Patterns

See eclipse zip file for java code. Various assumptions were necessary in order to solve the problem. These are listed as follows.

2.1 Assumptions

- The price was rounded down to the nearest integer before the points were applied. This means that in the case of threshold points the price, if over $100 was spent, was truncated to the integer below before being multiplied by two.
- Only one points method can be invoked (not a summation of each method). This led to the program choosing the best method for the user.
- The program in charge of determining the total price based upon item price and quantity of items was not required to be included. It was assumed that the price and customer name were known before using this point determining program.
- All grocery store points systems reveal the points total to the customer instantly. It is therefore possible to assume that the date of points application is the current date (i.e. the date of purchase).

It is reasonable to assume that another point of sale program would have already calculated the total price as well as a loyalty card program that would have determined the customer name. Allowing the user to input the name and price values provides a quick method of utilizing the program instead of updating the physical code each time a different price is used.

3 Design Anti-Patterns

a) In your own words, explain what "over-engineering" is and why it should be avoided.

Over-engineering is the process of designing a solution to a problem that is unnecessarily complicated. Though it is difficult to quantify exactly when a solution is over-engineered or not, a reasonable clue is when the design solution is more complicated than the problem itself. The slippery slope that leads to over-engineering is often a situation where the programmer is trying to code for as many possible future requirements that they can think of. It also occurs when the programmer attempts to keep their code more general or extendable than is reasonable within the current requirements specification.

While noble in its intentions, over-engineering is bad practice because this leads to wasted time spent on the project and an overly complicated solution. This over-complication can lead to the programmer having much more difficulty reaching the solution than necessary, potentially leading to missed deadlines or inadequate solutions. Many of the future requirements assumed by the programmer would likely prove to be unnecessary, making these sections of the code are of no use to the system. Furthermore, the extra code makes it difficult for other programmers to understand the logic or purpose behind the system. The code is more difficult to maintain and this leads to further wasted resources in the future attempting to maintain, or redo, the system. It is clearly wise to be cognisant of when a programmer may be over-engineering and to take steps to avoid it.

b) Provide an example of a design anti-pattern and explain why it is considered bad practice.

An example design anti-pattern is ‘The Blob’. This anti-pattern occurs when one class is very large and executes most of the processes of the program (i.e. the blob class). Other small programs may exist, but they likely only hold data or execute small processes. This anti-pattern is more synonymous with a procedural architectural style rather than an object oriented one and is often caused by the programmer not implementing an architectural method.

There are multiple reasons for why this anti-pattern is bad practice. When the anti-pattern occurs, it is likely that there is a lack of cohesion between the attributes and operations. The attributes and operations, though unrelated, may be bundled together from a lack of clarity. The system solution may be excessively complex for the provided problem statement. As almost all of the functionality is present in one class, it is difficult to modify the system without affecting the functionality of other objects within the class. Altering one section of the code may lead to errors in another section. Since the class is so large it is difficult to follow the logic of the system clearly for modifications or debugging. The size of the class may also lead to the system being inefficient. Finally, the produced class will be too complicated and specific for reuse. This is poor programming practice, as code refactoring is an extremely valuable tool for future projects.