**COMP0010 Software Engineering, JSH: Reengineering a legacy shell  
Report by: Muna Aghamelu  
Team Members: Nadhirah Rafidz, Celeste Manenc**

I contributed 45%, Nadhirah contributed 45% and Celeste contributed 10%.

The main design principles that I applied to the project were: Separation of concerns and modularisation, Single responsibility principle, and Interfaces. Implementing modularisation made the code more manageable (initially the program was in a single class, we created distinct classes such as Cd.java, Sed.java etc. so we could implement each class in isolation, it also ensured we could keep the application and command part of the code separate), it was easier to find bugs (the first thing I did to understand the code was to split it into smaller functions, my first commit to GitHub was refactoring the additional code by simply ensuring I was creating methods of no more than 5 lines), made identifying similarities between classes more evident (head and tail classes had very similar implementations, when I implemented piping there was a general demarcation between whether the input was coming from the stream or arguments) and finally, object oriented principles such as inheritance (EvalVisitor.java, line 13). The same goes for the single responsibility principle; the Grep.java class was solely responsible for matching strings from file, whilst the Application Factory class was responsible instantiating the Grep class, this made our code cleaner and much more concise. However, a disadvantage of using the Single responsibility rule meant that anytime a bug was found we had to change a class significantly; this led to some classes not being as well written as we had hoped (Head.java).

Moreover, the main design patterns implemented were the: Factory, Decorator and Visitor design patterns. Before deciding on these design patterns, we considered using the Builder pattern as an alternative to the Factory pattern, but we decided that each application only really had on job, so adding granularity to the object would not have any advantages for this project. Implementing the factory pattern helped with the abstraction of the visitor class; initially we had instantiated classes EvalVisitor class, this undermined our initial agreement of the single responsibility principle, so adopting an agile approach we created pseudo code for exactly what we wanted for EvalVisitor to do, redesigning it was much easier now we knew the Factory design pattern was taking care of creating the actual classes. I implemented the Decorator design pattern to deal with unsafe applications, initially I created an abstract Decorator class and had UnsafeDecorator extend from this, but I reconsidered and thought it was an unnecessary abstraction because we only needed to use the Decorator design pattern for one purpose – to output error messages. Before adopting the Decorator design pattern, another team member had tried to handle unsafe by creating duplications of the existing classes and instantiating those classes whenever an unsafe application was going to be used. In one our meetings we decide this was not the best approach, having duplications of the same applications. Therefore, the Decorator design pattern was great for ensuring we avoided bad practice and quality assurance. We used inheritance to implement the visitor design pattern- ANTLR produces its own Visitor class. Out of all design patterns this one had the most impact, it provided us the tools we needed to create commands and it all fell in place from there. We considered using the listener pattern as an alternative to the visitor, it would have made the ordering of applications much easier to handle however we would have had to create a stack to store the instructions as the listener traversed along our generated parse tree. The main conclusion I can draw from my experience of considering design, is just how important it is to plan using pseudo code and UML diagrams before coding – I work more efficiently that way and code is less likely to ‘break’ with a systematic approach.

The initial approach we took to refactoring (we later decided to stop refactoring and to rebuild the project from scratch) entailed using low-level refactoring processes such as: extracting code into methods and removing duplicated code. For example, I began splitting the contents of the switch statement into smaller functions called ‘cdApplication’, ‘pwdApplication’ and the case statement would call each method based on the string name passed. I also made the Jsh class OOP by creating a main class to instantiate a new Jsh shell (this was beneficial for our TDD approach), we soon learnt we needed to implement higher-level refactoring methods to better reengineer the shell. Therefore, we extracted the code using the Design Patterns aforementioned. As we were no longer refactoring existing code, we were vigilant whilst coding to avoid code smells and anti-patterns. I eventually removed all try-catch statements in the individual classes, isolating them to the main Jsh.java class. We avoided the use of platform dependent filenames by using temp directories when Unit Testing and System.getProperty(“line.separator”) when accessing file paths; however, there are still some evidence in our code where I forgot to use this (MkDir.java class, line 24). Magic numbers were extracted using constants variables (Head.java, line 25) and we defined our encodings to avoid undefining encodings (Grep.java, line 44).

About testing, we found many interesting bugs and tedious bugs.

Test Driven Development

Static Analysis and Bugs

Communication between Team

Other Aspects

Development Process

Quality Assurance