Design

Knowing which design patterns produce the most eloquent solutions, using available tools effectively and teamwork all require hard work, grit and perseverance. We made these tasks slightly easier for ourselves by planning well ahead, and good communication. Knowing where to begin was also not trivial, but Mukilan and I decided to start with the parser, which we built using Antlr since it is the mainstream parser-generator in the industry. When I began integrating the parser's generated classes into Java, I realised what shape the code needed to take; I needed to refactor the Jsh code into distinct classes with generic methods, and after thorough research, I found that the visitor design pattern would allow me to do this. I found a plethora of resourced to aid my understanding as the visitor is one of the main patterns from the reputed "Gang of Four" book. As ideas for future design patterns started naturally developing, we wrote them down in a UML diagram (see figure 1), I also added some applications inspired from the bash shell, which I was personally eager to develop.

The visitor design pattern allows you to add methods to objects of differing types, thus giving you a lot of scope with polymorphism. We were working with three different command types (Call, Pipe and Sequencing), so it made sense to create a "Command (visitable)" interface that each command implements. The visitable interface only had one function (the accept method) and the visitor interface only had three (a visit method for each command), so initially, it seemed counter-intuitive to create two interfaces for just four methods. However, using the visitor pattern, you start reaping the rewards of the laborious design as the project grows (decreasing your technical debt). If in the future, more commands were needed, we could conveniently define external classes that extend the visitor interface and the project would then support extra functionality without the need to edit existing code. Our classes naturally had distinct encapsulated sections, enforcing the "separation of concerns" principle and we followed good programming practice in that we had private instance variables with public getters and setters.

Since I viewed applications as data types, I split them up into distinct classes, which allowed me to get application objects that I would need when implementing the decorator pattern as we had planned. Consequently, I needed a way to group applications into one `umbrella` with some shared methods, so they implemented an "Application" interface, allowing them to be used by each command-type generically (since this interface also had a method that works on all applications, this doubles as an example of ad-hoc polymorphism). It was only due to this design pattern that I knew how to begin my code; I created my UML diagram after beginning the visitor implementation, and the rest of the patterns came to me naturally, almost by "wishful thinking". This pattern was a fundamental part of my code and I would struggle to think of alternative ways of using the same programming paradigms such as the separation of concerns, polymorphism, abstraction and encapsulation.

Next on my UML diagram was the factory pattern, which encapsulates and unifies the instantiating process of classes. For our project, I used a factory pattern to instantiate application objects so that only the factory is responsible for creating new applications. This meant that the Call class only needed to communicate with the factory to get an instance of an application; because I could simply write code that needed the factory to get an application object, I was able to make my task easier by programming by "wishful thinking". The factory design pattern also enforced the separation of concerns principle, so if any application became obsolete, one could simply delete the class and remove all relevant lines from the factory without impacting any other code, meaning we keep our code concise and prevent bit-rot. During the design phase, I did question whether there was a better way of implementing this application selection, so I investigated the strategy pattern. This would have meant that instead of returning an instance of an application from a factory, the strategy

pattern would simply run an application algorithm, which would have been enough for safe applications, however, our design also needed to cover unsafe applications. The factory pattern made it significantly easier to implement a decorator pattern to allow unsafe applications to alter application behaviour and so the design we had planned, worked better with the factory pattern.

Having separated applications into distinct classes and using the factory pattern, made the decorator pattern a no-brainer to implement unsafe applications. This was because the decorator pattern made it infinitely easier to attach new functionality to applications at runtime, without breaking pre-existing code that used the safe application objects. Again, I did think about slightly deviating from my design, and considered viable alternatives, one of which was creating an abstract application factory, and with separate factories for safe and unsafe applications. This, however, was unnecessarily verbose and gave me no functional advantage, which is why I stuck with my design. This temptation to throw in different exciting design patterns was a reoccurring pattern for me, I was constantly tempted to use all the new design patterns that I had not used before to come up with creative solutions. However, Paul Graham said it best in that "The shape of a program should reflect only the problem it needs to solve". After studying many design patterns in detail, we ensured that our UML diagram was perfect for our purpose and we followed it closely, making sure to only consider changes when they were really needed. Good programming design and practice genuinely made me write cleaner code and allowed my coding style to evolve; I seldom suffered from "writer's block" because I always had a design guiding my trajectory.

Refactoring
We did our best to adhere to programming standards during development, so when writing and refactoring code we made sure to remove code smells (See Figures 2 and 3). This included the avoidance of things like magic numbers, duplicate code, long parameter lists, unidentified encoding, and using platform-dependent filenames/paths. We also tried as far as possible to refrain from using global variables as they indicate poor modularity and are prone to bugs. This wasn't a huge issue in Java (where everything is part of a class and global variables technically don't exist) but we still had to be careful with variable scope and access modifiers. We also understood good refactoring meant making readable code, so I wrote detailed Javadoc comments for mainly public, but also some private functions whose purpose was hard to decipher. Within most applications, generic comments were made by Ben as he argued that Javadoc comments for all methods in all applications would be overkill and would make code illegible. Another important lesson we learnt was towards the end of our project, whilst writing some tests. We realized we had misread the specification slightly and that "globbing" should only work for unquoted inputs, we came up with a solution immediately, but it was unclean and required us to change the signature of a method in an interface, slightly deviating us from our design. We would usually have spent time changing the structure of our code to normalise the change, Mukilan had a complex idea to integrate globbing into the visitor pattern; however, at this stage, the productivity gain from refactoring and fixing this minor design issue would appear after the deadline so we thought it prudent to leave our code in a working state.

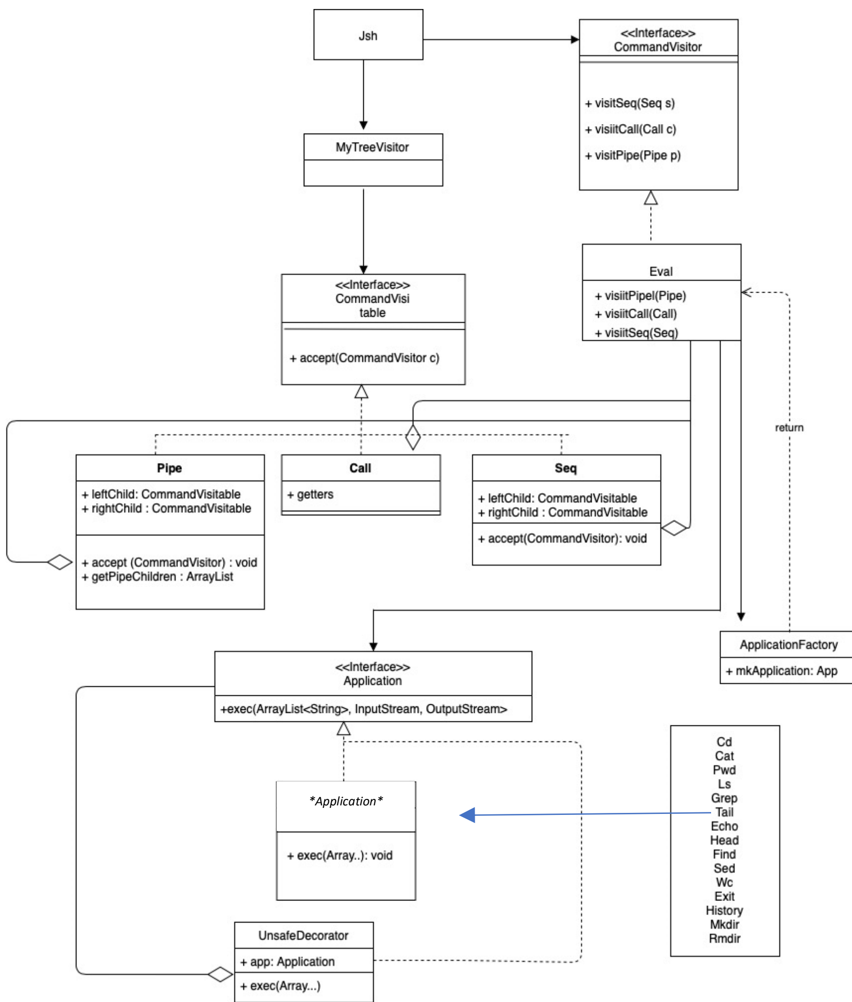
Testing
When making changes to code, I used (unstructured) smoke and explanatory tests to find any obvious flaws and ensure my code ran and was not fully dysfunctional. However, these tests make it difficult to track how much has been tested (coverage is subjective). Further to this, the effectiveness of unstructured tests is largely dependent on the skill of the tester, so although unstructured tests were useful to me during the development stage, I knew we had to employ a more rigorous technique for the formal tests. Mukilan, who took the lead on testing, wrote structured tests using Junit in a separate package, fixing minor bugs himself and informing me of

serious issues. We used Junit to give us a uniform development cycle so that we could correctly undertake test-driven development. Mukilan wrote failing tests, I wrote the code required to pass said tests, then Ben and I would refactor and clean up this code, ensuring all tests still pass; this cycle continued until we finished writing new code, which is when we began continuous testing. Since we were writing code in response to tests, our code was very concise, mirroring only exactly what was asked of us in the requirements. We were following the agile programming framework, so the cyclical nature of this test-driven development tied in very well with our sprints. Using TDD also helped me to write higher quality and cleaner code that was less susceptible to bugs, I found it very useful as a developer to know what code I had to write by simply trying to pass tests. As we found bugs, we also added regression testing, which meant that we kept tests for specific bugs in our code, even after their removal, in case they came back down the line.

We used a combination of Black box and white box tests, as they both gave a much better notion of code coverage; in white box, it was the fraction of code exercised in tests, and in black box it was the fraction of specification requirements satisfied. Black box testing comes from an external perspective so you can miss edge cases (inputs that a typical user wouldn't enter). White box tests work backwards, starting from the source code, so it is easier to spot edge cases but since you can't test for missing functionality, white box testing alone would also not suffice. At the end of the project, I helped Mukilan write tests and I learned how important having both white box and black box tests was. 100% coverage does not mean 100% tested code, you can hit every branch/statement, but you aren't hitting them with every combination possible, so we used a natural balance of both. Black box tests were especially powerful to us during integration and software testing after our parser was finished because we were able to test commands exactly as if the user was typing them into the Jsh shell, whereas white box tests allowed us to start deeper into the code and test isolated classes. No form of testing was more useful than the other intrinsically, they were all crucial at certain points in the development process, so it was the amalgamation of all of them that made our tests specifically useful.

During this coursework, I constantly made use of static analysis. This outlined some important things I needed to remove from my code such as unnecessary brackets, unused imports and even modifiers from interfaces however no major bugs were ever found. Another reason we used Junit was that it gave us access to tools such as "Temporary Folders" which created an isolated environment for us to create test files on. It also allowed us to perform fault injection, so we were able to test our shell's behaviour when faulty inputs were entered. This was to ensure that (for safe applications) our program does not do anything unsafe- throws the correct exception and terminates as required.

Appendix

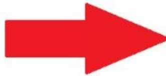


Sources:

- https://www.youtube.com/watch?v=TeZqKnC2gvA
- https://www.youtube.com/playlist?list=PL_DQiOR0jhbU3ZKyYIV9oxfcqXTpbK4Le
- https://www.youtube.com/watch?v=EcFVTgRHJLM
- https://www.youtube.com/watch?v=j40kRwSm4VE
- "Head First Design Patterns" by Eric Freeman and Elisabeth Robson
- "Gang of four" by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides.
- COMP0010 Lectures/Slides

Figure 1: UML Diagram. Note: The class labelled 'Application' is not an actual class, but a placeholder for the list of applications listed in the stray box on the bottom right (We did this to save space).

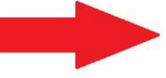
```
if (useIS) {
    String line = new String(input.readAllBytes());
    String[] lines = line.split(System.getProperty("line.separator"));

    if (lines.length <= headLines) {
        headLines = lines.length;
    }

    for (int i = 0; i < headLines; i++) {
        writer.write(lines[i] + System.getProperty("line.separator"));
        writer.flush();
    }
}
else {
    File headFile = new File(currentDirectory + File.separator + file);
    writeOutput(file, headFile, writer, currentDirectory);
```



```
if (!useIS) {
    File headFile = new File(currentDirectory + File.separator + file);
    writeOutput(file, headFile, writer, currentDirectory);
}
else {
    String line = new String(input.readAllBytes());
    String[] lines = line.split(System.getProperty("line.separator"));

    if (lines.length <= headLines) {
        headLines = lines.length;
    }

    for (int i = 0; i < headLines; i++) {
        writer.write(lines[i] + System.getProperty("line.separator"));
        writer.flush();
```

F

```
@Override
public void exec(ArrayList<String> args, InputStream input, OutputStream output, ArrayList<Boole
    BufferedWriter writer = new BufferedWriter(new OutputStreamWriter(output, StandardCharsets.U

    int numOfFiles = validateArgs(args);
    for (int i=0; i<numOfFiles; i++) {
        String path = currentDirectory + System.getProperty("file.separator") + args.get(i);
        File file = new File(path);
        if (file.exists()) {
            throw new RuntimeException("mkdir: File already exists, choose different name");
        }
        else {
            if (file.mkdir()){
                writer.write("Folder created sucessfully");
                writer.write(System.getProperty("line.separator"));
```



```
public void exec(ArrayList<String> args, InputStream input, OutputStream output, ArrayList<Bo
    BufferedWriter writer = new BufferedWriter(new OutputStreamWriter(output, StandardCharset

    int numOfFiles = validateArgs(args);
    for (int i = 0; i < numOfFiles; i++) {
        String path = buildString(Jsh.getCurrentDirectory(), args.get(i));
        File file = new File(path);
        if (file.exists()) {
            throw new RuntimeException("mkdir: File already exists, choose different name");
        }

        file.mkdir();
        writer.write("Folder created sucessfully");
        writer.write(System.getProperty("line.separator"));
        writer.flush();
```

```
*/
private String buildString(String currentDirectory, String arg){
    StringBuilder sb = new StringBuilder();
    sb.append(currentDirectory);
    sb.append(System.getProperty("file.separator"));
    sb.append(arg);
    return sb.toString();
}
```

Figure 1: Use of String builders instead of concatenation in loops to prevent build-up of garbage.