COMP0010 – Software engineering

JSH: re-engineering a legacy Java-based shell
Individual report: Dao Heng Liu [1801203] – Team 32

1. Design principles and design patterns

We decided to build Jsh mostly according to the reference design diagram from the delta debugging tutorial.

One of the first design pattern we employed was the factory method. The factory sits between the Jsh shell logic and the actual program that runs in the shell. The Jsh shell uses the factory method to get an instance of a ShellProgram without knowing in advance which specific ShellProgram it needs, instead of directly calling a class constructor directly. This simplifies the logic in the Jsh shell.

We employed polymorphism for our shell program classes. We had a ShellProgram class with the abstract “execute” method that each of the classes that extend ShellProgram override. This provides a uniform interface between the Jsh shell and each of the programs.

We used something akin to the decorator pattern for the unsafe version of shell programs; instead of creating a new class to handle the unsafe execution of the shell programs, we just added another method to the ShellPrograms class. It effectively acts as a decorator, as it calls the original execute method of the class that extends ShellProgram, and is invariant from class to class (that implement ShellProgram).

In the reference design diagram, it was mentioned that the eval method could be extracted using a visitor pattern. We investigated using a visitor pattern in our Jsh project and found that the visitor pattern introduces complexity that we deemed unecessary. The three classes falling under the Command interface have straightforward interactions between them and we felt that it did not warrant employing a visitor pattern.

Overall, we felt that this Jsh project is quite broad (many classes for all the different programs) and not very deep. Because of this, some design patterns were more important and useful (decorator and polymorphism), while other patterns were less important and more impractical to use (visitor pattern, in this instance).

2. Approaches to refactoring the legacy code

We first decided that it would make the code much more readable by each program into their individual class. That decluttered the Jsh class, which made it easier to work with (abstraction refactoring). As a result of moving the programs out of the main class, the first new bit of code was a factory class for Jsh to use to instantiate the needed application from their respective classes.

In the hopes of keeping complexity down, we tried to reuse and/or readapt the existing JSH code. For example, we kept the original JSH quoting and globbing code for a long time, only moving them to different classes without really modifying their logic too much (with the exception of adding more cases to check for, for example with backticks, pipe characters, etc.)

Then, we started to conforming existing functionality to the specification, as well as implementing new features using the red-green refactoring approach along with TDD for the majority of the remainder of the project.

About 2/3rds of the way through the development of Jsh, we realized that some pieces of code that we kept from the original Jsh were going to be very hard to readapt to meet the specification and that we would spend more time working around the existing code than completely rewriting them. One such part was the code that handled quotes and argument splitting. We decided to drop the exiting code and rewrite it completely from scratch. We learned that sometimes, trying to keep the complexity down by keeping existing code would in fact make the development process more tedious and complex.

3. Test-driven development

I managed to find the password to the acceptance tests during the reverse engineering tutorial. Thanks to the acceptance tests, the vast majority of the development of JSH was accompanied by constantly running a test suite. The only part where we worked on JSH without a test suite was when we were only refactoring the existing code without adding features or fixing bugs.

The original acceptance test written in Python that used Docker took too long to run (on my computer, it took over 400 seconds to run them; on my teammates' computer, it took over a 100 seconds.) We quickly rewrote the tests in Junit and used that for testing moving forward; it usually took less than 10 seconds to run over a hundred tests. IntelliJ made it very easy to selectively run certain tests, as well as and attach a debugger while running a test to track down where in the program a problem was occuring.

Having tests and applying TDD was instrumental to the development of JSH. We ran the tests whenever we made changes to the program. We discovered numerous bugs in the existing JSH code, as well as in code that we wrote using tests. We would often change pieces of code that would break more things than they fixed. Testing makes it immediately obvious that the "fix" actually made everything worse.

As a matter of fact, almost all new features we implemented were done based off of tests. Each time we needed to introduce a new feature, we would first write tests based off of the intended behaviour of the new feature before writing code. That way, we can attempt writing code and run the test suite to see if we did it correctly without having to type in commands manually. This accelerated our workflow and allowed us to stay focused on the task at hand, as we didn't need to think and try to remember how the feature was meant to work; the test suite will clearly point out what aspect of the feature is working and what aspect isn't.

Most of our tests are integration tests. They allowed precise enough targeting to test particular bits of code (thanks to command line switches and manually crafted inputs), while also ensuring that the rest of JSH still worked. For some methods, it was easier to create unit tests instead: when trying to simulate an IOException, it was easier to override the abstract Reader class to make it forcefully throw an exception and run an unit test on the method itself instead of calling JSH and trying to get filesystem permission errors instead.

4. Code quality and static analysis

Static analysis was very useful to us throughout development. We had two main ways of performing static analysis: one was using PMD, while the other one came with the IDE (IntelliJ).

Although I can't recall us finding bugs, the IDE (through static analysis) has pointed out many flaws in our logic. Some of those flaws include unreachable code, unecessarily complex if statements, nested if statements that could be compounded, variable assignment that is never accessed, etc.

Having these issues pointed out in the IDE as we type the code is very useful and accelerates the development process. We have far fewer compilation errors and runtime errors, which saves an enormous amount of time. The IDE will oftentimes point out pieces of code that can potentially cause problems at runtime under certain conditions; if the IDE warnings weren't present and our test suite didn't fully cover every single possible situation, the problem might have gone unnoticed for far too long and become very difficult to fix in the future.

Thanks to this, it has also allowed us to maintain a high standard of code quality. Simpler if statements conditionals and the removal of unreachable code makes the code more readable and makes the logic easier to follow. We also put in reasonable efforts in keeping the code styling consistent, which further helps with code readability.

5. Team communication

The team made a point early on to have good communication throughout the development process: we set up a group chat so that whenever there are updates on the project status, or when taskss need to be handled out, each member is easily reachable.

For the first few weeks of the project, we periodically met face-to-face to go over the existing code and make sure that everyone understands it well, so that we have good foundational knowledge. As term progressed and meeting face-to-face became harder and more inefficient, we started relying more on github for collaboration instead of doing peer-programming in person. To help track progress, we also created a Trello board to reduce the amount of "What have you done since last time?" in the team group chat, as well as assigning tasks for each person to work on.

During winter break, we would often call and use remote desktop software to do remote peer programming.

6. Other software engineering aspects

In order to collaborate remotely and be able to work on the project concurrently, we used git to do our version control. During this project, we learned (through hands-on usage) how to use git in general; creating, using, and deleting branches; solving merge conflicts; getting familiar with git integration in IntelliJ and at the command line; etc.

We were provided with a Dockerfile for deploying and testing Jsh in a Docker container. We made sure that the project still is deployable and works correctly inside the docker container as we worked on it. Testing Jsh inside a docker container also ensures consistency across runs, as it reduces the number of miscellaneous variables that can influence Jsh.

7. Jsh class diagrams