JSH SHELL INDIVIDUAL REPORT

Student id: 19077928

Introduction

JSH shell is a command interpreter program responsible for interpreting commands from users and performing specified tasks in the command lines. This report is to discuss how our group refactored and redesigned the shell to reinforce its stability and enrich its functionality.

Design principles and patterns

JSH shell comprises two parts. One is command parser and interpreter. It checks the validity of commands by using the parser generated by Antlr and interprets and performs them.

Visitor Pattern

To achieve the later function suggested above, we chose to apply visitor patterns for both commands and calls, on basis of Antlr-generated visitor classes.

The reason is that there are three different forms of commands (Call, Pip, and Sequence, all implement the Command interface), where Pip and Sequence consist of more than one Call. The three types of commands should be dealt with differently, whereas, the operation for each individual one is static. This is also the case for calls. Call command consists of a list of sub-calls (BaseCall, Input/OutputRedirection, and Substitution, all implement the Sub_Call interface), where quoted arguments and redirection should be specially evaluated. Visitor pattern perfectly suits the case as it can separate algorithms from the objects they operate on. As visitor pattern allows, visiting a complicated command from users can lead to visiting its constituent parts, i.e., Sub_Calls, so that the visitor can traverse down the parse tree and evaluate backward from atomic elements, the whole command can be evaluated provided all its sub-elements were correctly evaluated and executed.

Observer Pattern

If JSH shell fails to parse a command, it will throw an exception telling near which token there is an error. This is achieved by an observer class extending ‘BaseListener’ by Antlr so that once a syntax error is observed from the command line, it will be reported, and its position can be located.

The other part of JSH shell is application executor. The shell supports twelve applications, all implement the Application interface. Each application can be executed by creating the corresponding application object and calling its executing method.

Factory Pattern

Factory pattern is applied to create required application objects from commands. It will return any concrete application instantly by taking the name of the application. Instead of using switch cases to create objects as traditional factory does, we implemented a HashMap with application name and application object pairs, where the objects can be reused once created, which improves code performance.

Decorator Pattern

JSH shell supports both safe and unsafe applications. The unsafe behavior is achieved by using a decorator pattern. The unsafe decorator implements the Application interface, with its

constructor taking a safe application object. The decorator wraps the executing method of the safe application into a try-catch block to catch exceptions and print the error message instead, without contaminating the existing functionalities of the method.

Apart from the two main parts, the shell has a toolkit package, which contains classes with useful methods.

Singleton Pattern

One of the classes is WorkingDr, which keeps data of JSH shell's current working directory. Since a shell can have and keep track of only one current working directory, the class must be designed to have unique instantiation and well-guarded. Singleton pattern is therefore applied to ensure its uniqueness and security.

Refactoring

The shell has been refactored to a large extent from its original code.

Refactor applications

The original code had everything in one class, which was badly organized. We started by extracting pieces of codes into separate classes in different packages according to their usage. Specifically, all applications in the old version were mixed together, each of them was to be executed by switch statements, which was a bad design. We extracted applications into individual classes and used an application factory to create and execute them.

Looking into the code of each application, we found the internal code structure was a mess. They were refactored so that each application now has a method to check the correctness of its arguments, the main method to execute it, and a method to write results to output if necessary. It is worth mentioning that tail can be considered the reverse of head and the two have many similarities, so we let tail extend head to reduce the code quantity.

Refactor command evaluation

Initially, the shell had limited functionalities. In order to implement pip, redirection, and substitution, we decided to rewrite the command evaluation mechanism by using a more powerful visitor pattern. Now there is a separate class to parse command lines, the visitors to process commands and calls, and the evaluation class for call, pip, and sequence.

Add useful methods

While refactoring, we found that some pieces of code were repeatedly used, so we made a package called toolkit to contain classes with useful methods, such as Globing (to glob calls), InputReader (to read from files and stdin), and PatternMatcher (to check or find matched patterns). These classes guarantee code reuse, lowering the potential risk of introducing bugs.

Refactor error handling

We created a custom exception named JshException and wrapped exceptions that may occur, including IOExceptions, syntax errors and wrong argument errors, with proper error messages. This maintains the consistency of the shell design and protects the software and user data from the security risk that raw Java exceptions may leak confidential information.

Testing

Our development of JSH shell followed TDD principle. The shell was tested in two ways, one is integrated tests written in python, i.e., tests.py and acceptance-tests.py. During the

development, we repeatedly used these tests to ensure the system worked correctly as a whole and all requirements were met. Any failed test would be looked into and the problem exposed would be fixed immediately.
The other way is unit tests. They tested each individual class and covered as many branches as possible (95% covered), which aided in discovering hidden problems and further refactoring.

Code quality and static analysis

We managed to enhance code quality and readability in the following ways:

1. When trying to achieve certain functionality, we would first consider java built-in methods and APIs, since they are safe and of fair performance. For frequently used functionalities, we created classes and methods for them to avoid duplicated code and for code reuse.
2. The code was well-formatted. Variable and method names were deliberately chosen. Magic numbers were strictly avoided.
3. The code passed PMD analysis and error-revealing tests generated by randoop, which proves its conciseness and safety.
4. SpotBugs was used to examine the code, the bugs detected were looked into and fixed.

Team communication and project management

We divided our tasks reasonably, where two people (Zixuan Ding and Zihan Zhu) wrote the main code of JSH shell, with the other (Coco Liu) viewing the code simultaneously, giving suggestions, and writing unit tests accordingly. To reinforce team communication, our group held meetings biweekly and frequently communicated with and helped one another online. Occasionally, we would code in pairs remotely, when one person shared the screen and did coding, with the other being the consultant.
In terms of project management, two weeks was spent on understanding how the original code worked, four weeks was spent on implementation of new features and general refactoring. Since then, we had been continuously refining our code with reference to the specification, python tests, unit tests, and knowledge gained from lectures.

Extra contribution

Advanced tools are used:

- Antlr is used to generate the parser for JSH commands. Our customized visitor and observer classes also extend those Antlr generated.
- Randoop was used to generate automated tests. Our shell passed the error-revealing unit tests it generated based on JDK specifications.
- SpotBugs maven plugin was used to detect bugs in our code.

Extra functionalities are provided:

- Applications can read not only from files and by redirection, but also directly from standard input if there is no argument of file name specified.
- The shell can take care of syntax errors from parsing commands. We enabled the listener function of Antlr plugin (inside the .xml file) and wrapped the error message to provide more information about why JSH shell fails to parse a command.