SEP Coursework 2 Report

Contents

Programming and Design

In this section, I will explain how I incorporated the command design pattern and the model-view-controller architecture into my design. Furthermore, I will explain how I implemented extensions.

Refactoring and Redesign

I began by researching the command design pattern and started to model classes in accordance with the structure illustrated in Figure 1 by Bates et al. (2004).

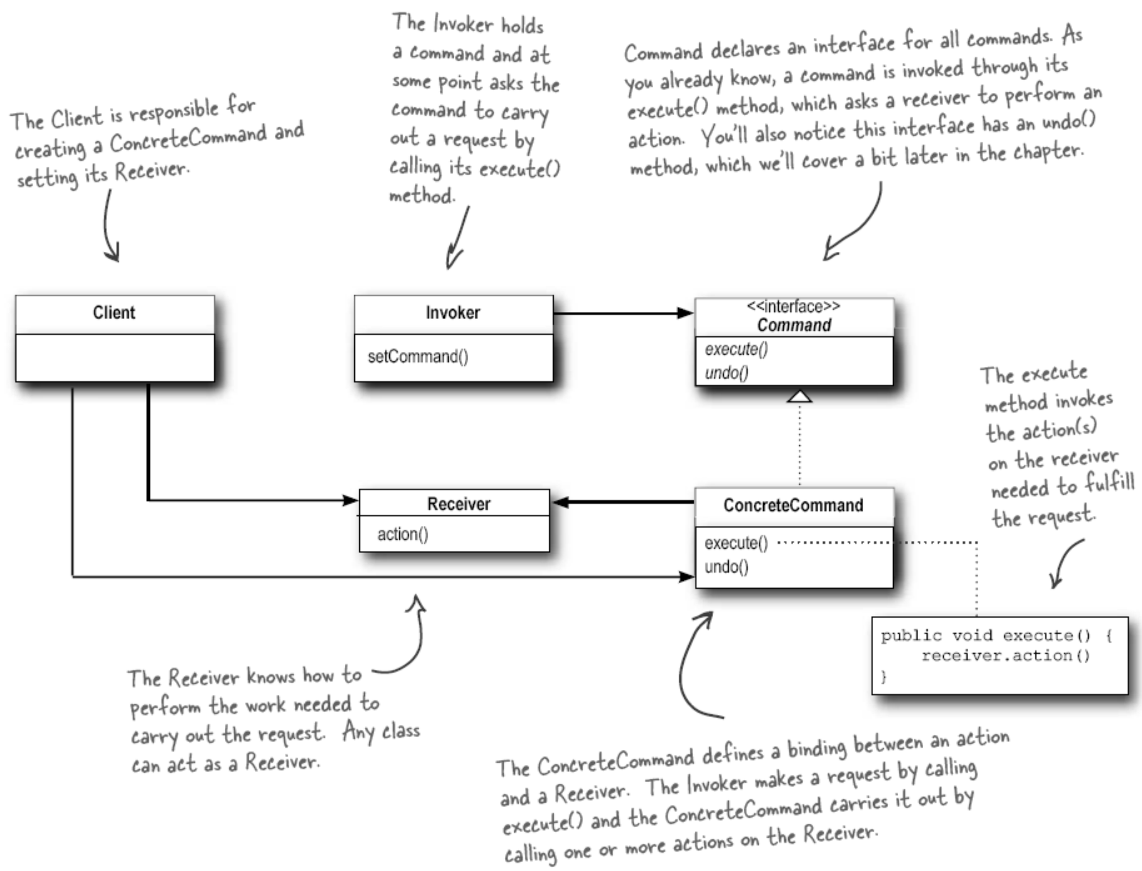

Figure 1. Command Pattern Class Diagram. (Bates et al., 2004, p.207).

To begin with, my Command interface class only contained the execute() method, as undo() was implemented later. Each concrete command's (body, send, fetch etc.) functionality, was slowly extracted out from the original Client class, and placed in their respective Command child classes. At first, it proved difficult to split the Client class into separate classes, as many methods had high coupling and low cohesion. However, once all the commands were extracted from Client, coupling had been lowered and the code became easier to interpret and extend.

Once the command pattern was almost in place, a link between the Client and Command child classes was required, and the usefulness of an MVC architecture became apparent. A Controller class was created, and the methods to parse user input were added to it. The method originally used a switch statement to check if the user's entered command is valid, and if true, a command object was instantiated and invoked by calling it's overridden execute() method. This change meant that I removed the separate Invoker class from the command pattern, as the method within the Controller now took care of its purpose. When the Controller class was finalised, the switch statement was refactored with lambda expressions which stored Command objects inside a HashMap<String, Runnable>. This made it very easy to retrieve and execute commands when needed, reduced the code count drastically and made the class easier to maintain and extend.

Next, the remaining code within Client, including the main program loop, was refactored and added into a View class. Leaving Client with only the main method, and a constructor to initialise the MVC classes. The View class became responsible for running and closing the program as well as most frontend/UI methods, such as taking user input within the while loop, and passing it to the Controller for processing. System.exit(0), originally used to close the program, was removed and replaced with a boolean variable, 'running', which was used to gracefully break the main while(running){} loop and close the program.

A Model class was added which took the role of Receiver in the pattern, and the remaining methods in CLFormatter were added to it, making CLFormatter redundant. Shared command functionality, such as server interactions (ClientChannel.send(Message), ClientChannel.receive()) and input validation, were placed within the Model, leaving concrete command classes encapsulating only their command-specific data and methods. This eliminated any duplication present across the command classes.

At this point, the MVC architecture and the command pattern were in place. Except the program was lacking one thing from the original specification: the state management of the program, drafting and main. The Controller needed to manage which commands could be invoked based on the current state of the program. To solve this, I created two classes, a State interface, containing get and add command methods, and a child enum StateCommands which represented the constants MAIN and DRAFTING. Each constant contained the same overridden methods, and a HashMap to store its associated Command objects. Model stored a State variable to keep track of the active state, and change state, which could be accessed by Controller to determine which commands are valid. For Controller to support this, commands were initialised by its constructor in two separate HashMap collections and passed to the appropriate enum constant (State.MAIN and State.DRAFTING) for storage. Below (Figure 2) shows the method that checks the validity of an entered command based on the State retrieved from the Model. The State HashMap (stateCommandMap) is searched for a command starting with the users input, and if a match is found, code is executed which sets that command to a Class global Command variable for later use.

```java
    public boolean isExecuteCommandValid(String userInput) {
        State state = getModel().getState();
        HashMap stateCommandMap = state.getCommands();
        userC = null;
        extractInput(userInput);

        if ((commandWord == null) || (commandWord.trim().isEmpty())){
            return false;
        }

        stateCommandMap.forEach((k, v) -> {
            if (k.toString().startsWith(commandWord)) {((Runnable) v).run();}});
```

Figure 2. Command method within Controller.

These changes meant that additional commands could easily be enabled by adding a single line pointing to the Command's class to the Controller.addCommands() method, shown below in Figure 3.

```java
    public void addCommands() {
        mainCommands.put(exit, () -> userC = new ExitCommand (model, this));
        mainCommands.put(fetch, () -> userC = new FetchCommand(model, argument));
        mainCommands.put(compose, () -> userC = new ComposeCommand(model, argument));
        mainCommands.put(list, () -> userC = new ListCommand(model));

        draftCommands.put(topic, () -> userC = new TopicCommand(model, argument));
        draftCommands.put(exit, () -> userC = new ExitCommand(model, this));
        draftCommands.put(send, () -> userC = new SendCommand(model));
        draftCommands.put(body, () -> userC = new BodyCommand(model, arguments));
        draftCommands.put(discard, () -> userC = new DiscardCommand(model));

        State.MAIN.addCommands(mainCommands);
        State.DRAFTING.addCommands(draftCommands);
    }
```

Figure 3. Method to initialise Commands within a HashMap in the Controller class.

Extensions

Additional commands: list, discard, topic [topic] and undo were added to the program. The function of these commands was built according to the specification with the aid of unit testing, and they were implemented into the MVC framework and command pattern as described in the previous section.

The functionality for an undo command was incorporated into the program by adding an undo() method to the Command interface which returned a boolean. True was returned to tell the Controller that the command could be undone, false meant it could not. A Stack commandHistory (see Figure 4) was added to the Controller to keep track of commands that had been executed. If undo were entered, the Controller would call the undo() method on the previously entered Command on the top of the Stack.

```
    /**
     * Invokes a command, if a command is valid and existing.
     *
     * @param userInput the user's entered input.
     * @return True if a command was invoked.
     */
    public boolean invokeCommand(String userInput) {
        if (isExecuteCommandValid(userInput)) {
            userC.execute();
            commandData.put(userC, commandWord);
            commandHistory.add(userC);
            return true;

        } else if (commandWord.equals(undo)) {
                    if(commandHistory.isEmpty()) {
                        System.out.println(rb.getString("undo_empty"));
                        return true;
                    }

                    Command cmd = (Command) commandHistory.pop();
                    if (cmd.undo()) {
                        System.out.println(MessageFormat.format(rb.getString
                        ("undo_successful"), commandData.get(cmd)));
                        return true;
                    } else {
                        System.out.println(MessageFormat.format(rb.getString
                        ("undo_unsuccessful"), commandData.get(cmd)));
                        return true;
                    }
        }
        return false;
    }
```

Figure 4. Invoker method within Controller, showing the command history statements.

I decided that the commands that would be appropriate to be undone were body, compose and topic [topic]. I added a test class to my test suite named UndoCommandTest to test each of the aforementioned commands. In Figure 5, below, the test composes a topic #a, and adds topics #b and #c to the draft via the topic command, it then enters the undo command to remove the previously added topic #c from the draft as shown in the assertion.

```
@Test
public void undoTopic() throws IOException {
    provideInput("compose a\nbody a\ntopic b\ntopic c\nundo\nexit");
    Client.main(super.getClientArgs());

    assertEquals("Drafting: #a, #b", (getOutLine(23)));
}
```

Figure 5. JUnit test for undo topic.

Internationalisation was added by adding a resource bundle package which contained a locale-named (MessageBundle_en_GB) properties file containing Strings. A public ResourceBundle was initialised within Client using the properties file pointed at by the locally set Locale. This bundle was publicly accessible by classes throughout Client's package. All English hardcoded Strings were replaced with a method call to the resource bundle's method getString() which could retrieve the appropriate String according to the set locale as demonstrated below (see Figures 6 and 7) by the body Command. This call would point to the named parameter within the properties file.

```
System.out.println(rb.getString("body_invalid_length"));
```

Figure 6. Method call to ResourceBundle.toString() for String retrieval.

```
# Command-Specific Strings #
send_invalid_empty = Cannot publish a draft with an empty body.
body_invalid_length = Seet body should be non-empty and not \
    longer than 48 characters.
body_invalid_line = Seet body should be a single line, not: {0}
```

Figure 7. Properties file parameter contents, showing body_invalid_length.

To add support for another language would be effortless now, as it would only require adding another properties file containing the same parameter names, but with different values mapped to each parameter key.

Software Engineering Practices

In this section, I will explain the various software engineering practices I used throughout the project.

Testing

I created and ran JUnit tests throughout development. I created a TestSuite parent class to contain methods for setting up and tearing down each JUnit test. This includes setting the Client's parameters and a method, String getOutLine(int line) which returns a String representing a line from the tests output stream.

The first child class I added to TestSuite was OriginalCommandsFunctionalityTest which tested all the functionality of the original commands. To begin with, these tests worked with the original project, but as I added changes to the code, such as additional output to the user, the tests failed and had to be changed to check the correct line of output in the assertion.

Two examples of tests which check fixed bugs are: emptyInputCommand() and composeEmpty(). In the original Client, entering an empty input would cause an exception that crashes the program, attempting to compose a topic without an argument would cause the same problem. This was fixed by letting the user know that input is required and continuing the program. These two tests are shown below (see Figure 8).

```
public class OriginalCommandsFunctionalityTest extends TestSuite {

    private Thread myThread;

    @Test
    public void emptyInputCommand() throws IOException {
        String expectedOutput = "> Could not parse command/args.";
        provideInput("  \rexit");
        Client.main(super.getClientArgs());

        assertEquals(expectedOutput, (getOutLine(6)));
    }

    @Test
    public void composeEmpty() throws IOException {
        String expectedOutput = "> Topic name should be non-empty and"
                + " not longer than 8 characters.";
        provideInput("compose\nexit");
        Client.main(super.getClientArgs());

        assertEquals(expectedOutput, (getOutLine(6)));
    }
```

Figure 8. emptyInputCommand() and composeEmpty() tests.

Extension work was tested within AdditionalCommandsTest and also UndoCommandTest as previously stated. These classes contain tests that test for correct behaviour. The trickiest part of running these tests were setting up the running of a Server within each JUnit, I ran into many problems trying to successfully run and close the Server so that I could test the commands that require communication with the Server. I overcame this problem by undertaking research and found out that I had to run the Server in a concurrent Thread. The three tests shown in Figure 9, below, show tests for each of the additional commands (except undo, which tests are in a separate class). The assertion statement expects the output line as correct behaviour and will otherwise fail.

```java
    @Test
    public void discardDraft() throws IOException {
        try (Server server = new Server(8888)) {
            myThread = new Thread(() -> server.run());
            myThread.start();
            provideInput("compose mytopic\nbody one\ndiscard\nexit");
            Client.main(super.getClientArgs());
        }
        boolean expected = (getOutLine(14).startsWith("[Main]"));
        printOutputLines();

        assertEquals(expected, true);
    }

    @Test
    public void listTopics() throws IOException {
        try (Server server = new Server(8888)) {
            myThread = new Thread(() -> server.run());
            myThread.start();
            provideInput("compose mytop\nbody aa\nsend\nlist\nexit");
            Client.main(super.getClientArgs());
        }

        assertEquals("Topics: [mytop]", getOutLine(18));
    }

    @Test
    public void composeAddTopic() throws IOException {
        try (Server server = new Server(8888)) {
            myThread = new Thread(() -> server.run());
            myThread.start();
            provideInput("compose one\nbody 1\nsend\n compose one\nbody 2\n"
                    + "topic two\nsend\nfetch two\nexit");
            Client.main(super.getClientArgs());
        }

        assertEquals("       Aston  2", getOutLine(33));
    }
```

Figure 9. Additional command tests.

I applied black box testing techniques to a test class named InputValidationTest. The tests within this class test the lower, mid, upper, and outer bounds of a test case. For example, the length of a body is valid if it is between 1 and 48 characters inclusive. To thoroughly test body length, I created the following tests:

- Outer minimum (0 chars)
- Minimum (1 char)
- Mid (24 chars)
- Maximum (48 chars)
- Outer-max (49 chars)

```java
    /**
     * Tests the lower outer bound of body length (0).
     */
    @Test
    public void emptyBody() throws IOException {
        String expectedOutput;
        try (Server server = new Server(8888)) {
            myThread = new Thread(() -> server.run());
            myThread.start();
            expectedOutput = "> Seet body should be non-empty and"
                    + " not longer than 48 characters.";
            provideInput("compose top\nbody \nexit");
            Client.main(super.getClientArgs());
        }

        assertEquals(expectedOutput, (getOutLine(9)));
    }

    /**
     * Tests the lower bound of body length (1).
     */
    @Test
    public void minBodyLength1() throws IOException {
        String expectedOutput;
        try (Server server = new Server(8888)) {
            myThread = new Thread(() -> server.run());
            myThread.start();
            expectedOutput = "           1  1";
            provideInput("compose top\nbody 1\nexit");
            Client.main(super.getClientArgs());
        }

        assertEquals(expectedOutput, (getOutLine(11)));
    }
```

Figure 10. Lower and Outer-Lower boundary tests for body length.

For illustration, Figure 10, above, shows the lower outer and lower bound tests for body length are shown. With all of the boundary tests set in place, I was confident that the validation of body length was robust. I applied this same technique to all other tests within InputValidationTest.

Following completion of the project, I generated a test coverage report using the JaCoCo plugin. The report indicated that my client package (client, client.command and client.state) had an overall instruction coverage of 89%, and an overall branch coverage of 79%. A screenshot of the html report output is shown below in Figure 11.

JaCoCoverage analysis of project "SEP" (powered by JaCoCo from EclEmma)

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
sep.seeter.server		58%		40%	28	54	45	146	5	21	0	3
sep.seeter.net.channel		39%		32%	10	17	39	62	1	6	0	1
sep.seeter.net.message		73%		54%	15	41	13	75	3	28	0	7
sep.seeter.client		92%		82%	12	83	23	220	0	49	0	5
sep.mvc		0%		0%	13	13	25	25	9	9	3	3
sep.seeter.client.command		84%		75%	15	44	17	109	6	26	0	8
sep.seeter.client.state		92%		n/a	1	10	0	9	1	10	0	3
Total	750 of 2,850	74%	92 of 219	58%	94	262	162	646	25	149	3	30

Figure 11. JaCoCo Report Output.

Static Code Analysis

I ran static analysis tools EasyPMD and FindBugs to aid in the development of the project. Figure 12, below, shows the EasyPMD ruleset I used.

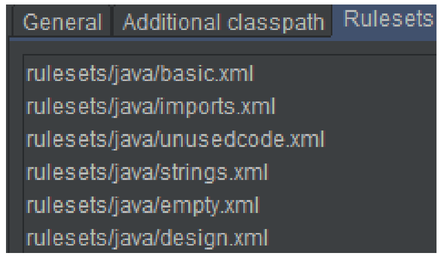

Figure 12. EasyPMD ruleset.

Three warnings/errors encountered from static analysis:

1. Two interface classes State and Command had public access modifiers on the methods, however, EasyPMD detected the modifiers were useless as they would be implicitly set as public. Removing the modifiers made the code less bulky. (see Figure 13).

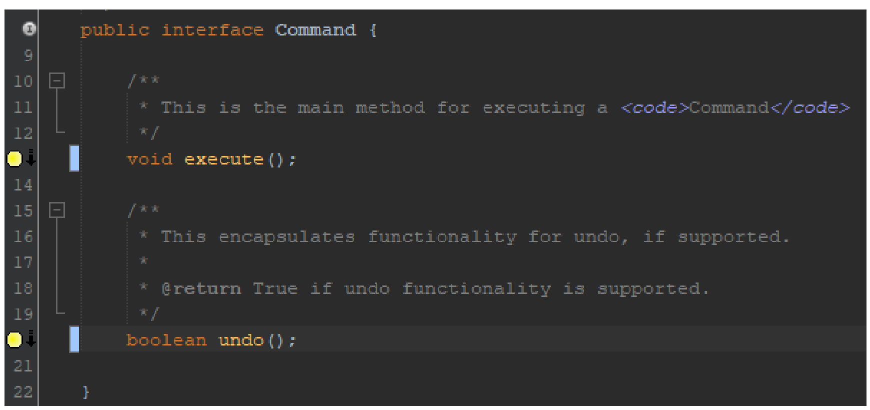

Figure 13. State interface access modifiers removed.

2. EasyPMD indicated that the statement at line 218 (see Figure 14, below) could cause confusion as it uses boolean negation operator (!). I corrected this warning by removing the operator and switching the if and else statement bodies around. I believe this change improved the readability of the code.

```
217     public boolean removeDraftLine() {
            if (!draftLines.isEmpty()) {
219             draftLines.remove(draftLines.size()-1);
220             return true;
221         } else {
222             System.out.println(rb.getString("empty_draft_lines"));
223             return false;
224         }
225     }
```

Figure 14. Boolean negation operator on line 218, before removal.

3. EasyPMD warned me that a default label in State switch case was missing. I fixed this warning by adding a default label (see Figure 15, below). Without this label, problems could occur if the state were neither MAIN nor DRAFTING.

```
304     public String getStateHeader() {
305         switch (state) {
306             case MAIN:
307                 return rb.getString("main_state_header");
308             case DRAFTING:
309                 return getDraftingOutput();
310             default:
311                 return rb.getString("state_not_set");
312         }
313     }
314 }
```

Figure 15. Switch case default label added on line 310.

4. In addition to the above. I corrected many small warnings in the Client class. Namely setting the constructor as private, the class as final, and the BufferedReader as final also. This could prevent any potential accessibility/security problems from arising.

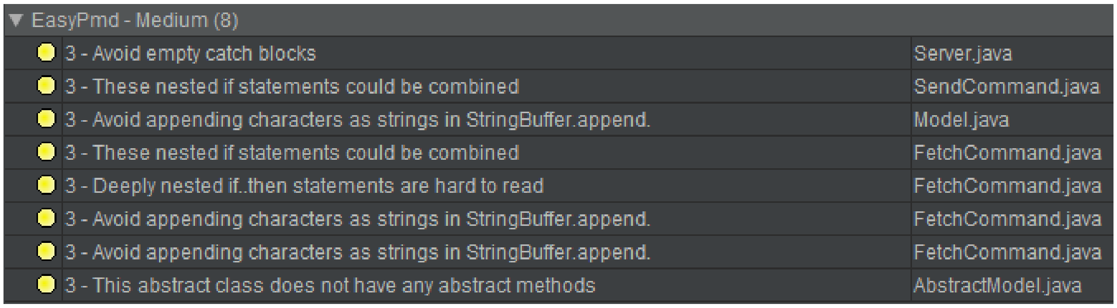

Figure 16. Final EasyPMD output.

Figure 16, above, shows the final output from EasyPMD. There are 8 medium level 3 warnings. The reasons I could not fix them are listed as being:

- I am not permitted to change the Server.java and AbstractModel.java files.
- The three nested if statements could not be changed without changing the functionality of the program, as the first if checks if a topic is valid, and upon entering the nested if, the topic is sent to the server.
- The StringBuffer warnings are related to original code that formats Strings. I do not have knowledge of StringBuilder and did not want to attempt to change it and break seemingly perfectly functioning methods.

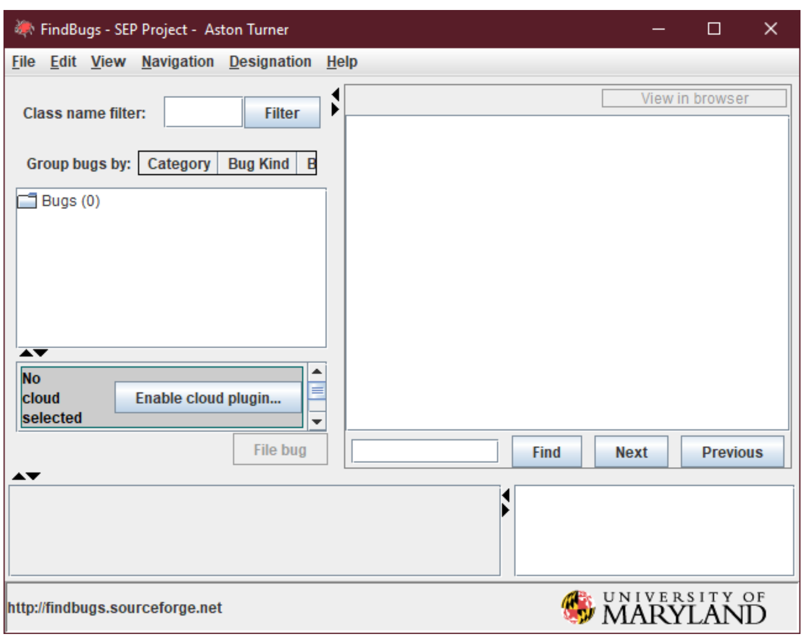

Figure 17. Final FindBugs analysis output.

Running an analysis of the final project in FindBugs found 0 bugs as shown in the post-analysis screenshot above (Figure 17).

Version Control

I committed to GitHub frequently with meaningful comments using GitHub desktop. I had almost never used branches before this project, so it was a learning curve for me, but I appreciate the benefits of using them now. Throughout development, I used four branches including the master branch. The other three branches I used were for adding the additional commands (list, topic [topic] and discard), adding the undo command and lastly a branch for internationalisation. When adding the first branch (additional-commands), the master branch was in a stable state which provided me with a clean slate to start adding additional functionality to. This provided me with reassurance that I could rollback to the stable master branch if problems occurred. I also recognised the advantages branches provided to teams working collaboratively on a project.

The CHANGELOG.txt file can be found inside the project folder.

Documentation

I generated Javadoc comments for the project via the NetBeans menu, and then added a readable and meaningful message to each comment and tag to facilitate future modifications and the overall readability of the code. On completion of adding Javadoc comments, I generated the html files that represent the Javadoc. I have highlighted some areas of the Javadoc which I thought were helpful, below.

Class Summary

Class	Description
Client	This class contains the main method for running the program, as well as the initialisation of the MVC classes and locale.
Controller	In charge of initialising commands in a List and returning Command objects.
Model	This Class acts as a receiver within the Command Pattern to implement all operations on commands including server interaction and drafting methods.
View	In charge of running the main program loop for taking user input and outputting (most) UI output using the initialised Readers.

Figure 18. Javadoc Class summary.

The class summary shown above in Figure 18, will allow someone unfamiliar with the project to quickly understand the role of each of the main classes within the client package. If they wanted to extend the project by adding more commands, it is quite straight forward from the comments to know which classes require investigating.

Class Client

java.lang.Object
    sep.seeter.client.Client

```
public class Client
extends java.lang.Object
```

This class contains the main method for running the program, as well as the initialisation of the MVC classes and locale.

Figure 19. Javadoc breakdown of the Client class.

Another example shown in Figure 19, above, shows my concise breakdown of the purpose of the Client class to any readers.

Deployment

The client can be ran from an IDE capable of compiling Java, the main arguments can be set via the project configuration. Required arguments are : user name, host name and port number. E.g. Aston localhost 8888. The server can be concurrently ran via a command line by navigating to the root project folder, and entering '-cp build\classes sep.seeter.server.Server [port number]'.

Built With

- NetBeans IDE 8.2 - The IDE used.
- JaCoCo - Java Code Coverage library.
- FindBugs™ - Java static code analysis tool.
- EasyPmd - NetBeans Java static code analysis plugin.
- Lizard - Code complexity analyser.

Figure 20. README.MD hosted on the project's GitHub repository page.

Furthermore, I created a readme file on GitHub that was added to the project folder (see Figure 20, above). A highlight of this readme explains how to deploy the project, and the technologies used to build it so far. This may be useful to provide an insight into the project by any future collaborators.

Bonus extensions

From the bonus extension section in the specification, I added the topic [topic] command. This command allows additional topics to be added to the current draft. I added functionality so that the execution of this command can be undone by using the undo command. I created many JUnit tests to test many aspects of this command, e.g. to ensure that the command can only be undone if there is more than one topic in the current draft.

Figure 21, Below, shows the test that adds a body from another topic to an existing topic, with the use of the topic command. The input is run twice to ensure robustness and check for body duplication bugs.

```java
@Test
public void composeAddBodyToExistingTopic() throws IOException {
    try (Server server = new Server(8888)) {
        myThread = new Thread(() -> server.run());
        myThread.start();

        // Run the same input twice to check for body duplication bugs.
        provideInput("compose one\nbody 1\nbody 2\nsend"
                + "\ncompose one\nbody 3\nsend\n"
                + "\ncompose two\nbody two\ntopic one\nsend"
                + "compose one\nbody 1\nbody 2\nsend"
                + "\ncompose one\nbody 3\nsend\n"
                + "\ncompose two\nbody two\ntopic one\nsend"
                + "\nfetch one\nexit");
        Client.main(super.getClientArgs());
    }

    // Output of topic one should be: 1\2\3\two\1\2\3\two.
    // Output of topic two should be: two\two.
    assertEquals("       Aston  1", getOutLine(94));
    assertEquals("       Aston  2", getOutLine(95));
    assertEquals("       Aston  3", getOutLine(96));
    assertEquals("       Aston  two", getOutLine(97));
    assertEquals("       Aston  1", getOutLine(98));
    assertEquals("       Aston  2", getOutLine(99));
    assertEquals("       Aston  3", getOutLine(100));
    assertEquals("       Aston  two", getOutLine(101));
}
```

Figure 21. Unit test for the topic command.

The pseudocode for this command's input is roughly as follows:

1. Create a new topic draft named "one".
2. Add three lines of body to it: "1, 2, 3" and send.
3. Create a new topic draft named "two".
4. Add a line of body to it "two" and add the existing topic "one" to the draft and send.
5. Repeat steps 1-4.
6. Check that the body output from fetching topic one includes "two" in the correct places from where it was created inside the topic "two" draft.

References

Bates, B., Freeman, E., Freeman, E. and Sierra, K., 2004. *Head First Design Patterns*. 1st ed. Sebastopol, CA: O'Reilly, p.207.