University of Hertfordshire
School of Computer Science
BSc Computer Science

Module: Software Engineering Practice

Code Review, Critique, and Use of Software Engineering Practices

Alex Williams

Level 6

(2022-23)

Table of Contents

1.0 Functionality & Structure of the Prototype

This section will describe the general functionality and structure of the Fotoshop prototype.

The prototype starts with the Main class creating an Editor object and initialises it using the `set()` and `edit()` methods. These methods create a Parser object and ask it to get the user's input through the `getCommand()` method. The Parser then handles the user input by checking if the command is valid through the `isCommand()` method in the CommandWords class. A new Command object containing the user's input is returned. The Editor class then processes the returned Command object by checking if the command is equal to a valid command and then executes it by passing it through the corresponding method. Certain command methods, such as `mono()`, convert the image to monochrome and creates a ColorImage object to do that.

Upon execution of a command, the user can enter subsequent commands. This is achieved through iteration in the `edit()` method. The method continues to ask the Parser for the user's input until the user uses the exit command. This then breaks the loop and uses a `System.exit` call to terminate the program.

1.1 Structured Diagram of Prototype

The following is a diagram of the prototype to help illustrate the functionality and structure of the program as described previously.

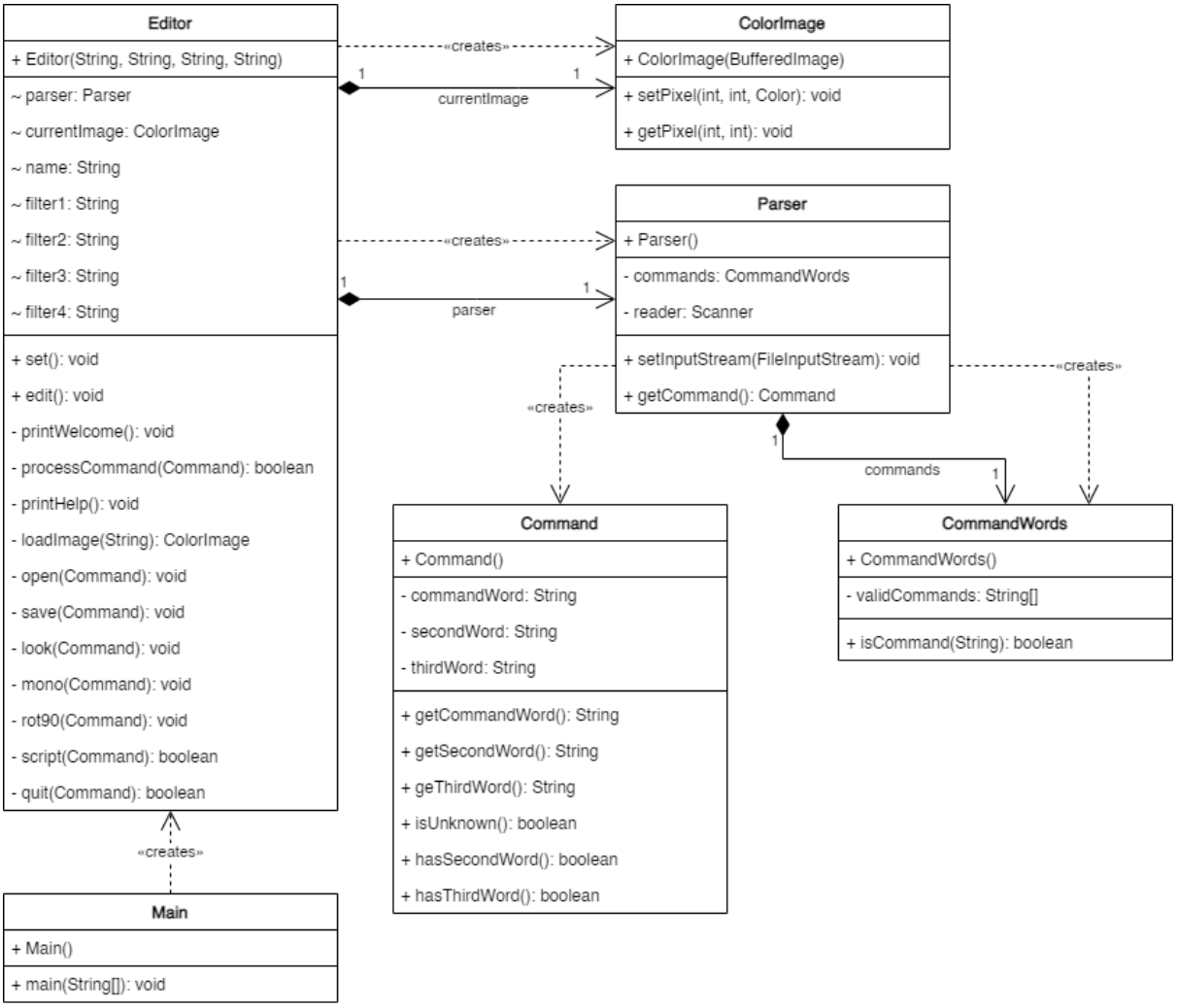

Fig. 1 – Class Diagram of the Program

2.0 Main Critique

This section will detail the main problems with the program and potential solutions to the examples given.

- The structure of the Editor class possesses low cohesion. It contains user interface methods such as `processCommand()`, `printHelp()` & `printWelcome()` that would be better represented in a separate class.
    - These issues could be solved by breaking down the Editor class into smaller classes based on methods with similar functionality, such as a UI class for handling the interface portions of the code.

- The Editor class also has high coupling. This is evident in its interdependence with the other classes and how it directly references their inner workings.
    - For example, when checking for valid command inputs, the class invokes `command.getCommandWord()`. The dependence on the Command class means that if changes are made, each reference to it in the Editor class must now be altered.
    - A way to alleviate this issue would be to create an interface class that acts as a middleman between the two classes. That way, whenever changes are implemented, it won't require refactoring classes that directly reference the other class' methods.
- The filter variables have generic names (`filter1`, `filter2`, etc.) And should be defined better. The implementation is also not extensible when new filters are added, which is mentioned in the specification's future plans.
    - A possible solution could be replacing the current system with an array containing each filter.
- The edit method contains a `System.exit` call for when the user is finished with the program. This is bad practice as `System.exit` closes the entire Java runtime, meaning testing approaches such as unit testing will fail when this line is executed.
- The set method creates a Parser object and is called in the Main class after creating the Editor object.
    - As the purpose of a constructor is to handle the setup and creation of objects, it would make more sense to include the method call in the Editor class' constructor.
- The help command never triggers. This is because it uses the == operator that compares the two operands in memory.
    - An `equals()` method should be used instead because it compares character by character when the two aren't the same object.
- The Editor class has variable shadowing in its constructor. The local variables are identical in name to the previously declared filter variables. This means only the values of the local variables will be used when it is accessed.
    - Changing the names of the local variables would solve this issue.

- The strings are declared using `System.out.println`, meaning they cannot be modified. This presents issues with the product's internationalisation. For example, creating translations would be difficult.
    - A potential solution would be to override the class' `toString()` method to modify strings.
- The user interface is command-line based, which goes against the specification's plans to include a graphic user interface in future.
    - An obvious solution would be to rework the interface into an implementation that accommodates a GUI, possibly following a design pattern to do so.

3.0 Static Code Analysis

This section will discuss issues identified through static analysis tools. While the results highlighted several issues with the code, the issues shown were chosen based on their relevance/severity.

3.1 SpotBugs

Fig. 2 – SpotBugs' report on bad practice in the Editor class.

SpotBugs identified issues with the code in the Editor class. One of them related to the use of `System.exit` to terminate the program. This is bad practice as it closes the entire Java runtime. This means that testing methods like JUnits will inevitably fail as it is dependent on the runtime. It's better to throw a RuntimeException instead.

3.2 PMD

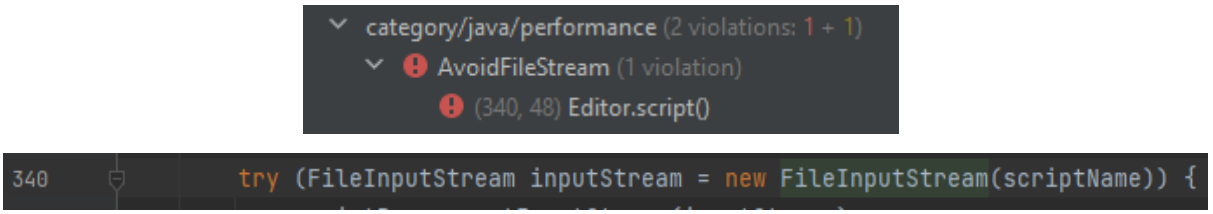

Fig. 3 – PMD & Line 340 of the Editor class displaying a performance error.

PMD highlighted a potential performance issue in the Editor class. `FinalInputStream`'s reliance on a finaliser method will cause a pause in garbage collection. Instead of instantiating a `FileInputStream`, `Files.newInputStream(Paths.get(scriptName))` should be used to solve this issue.

Fig. 4 – PMD's warning and the offending method.

Another problem was found in the cyclomatic complexity of the `processCommand()` method. PMD reported the method as having a high complexity of 10. Methods with high cyclomatic complexity should instead be broken down into smaller methods.

3.3 Built-In Static Analyser (IntelliJ)

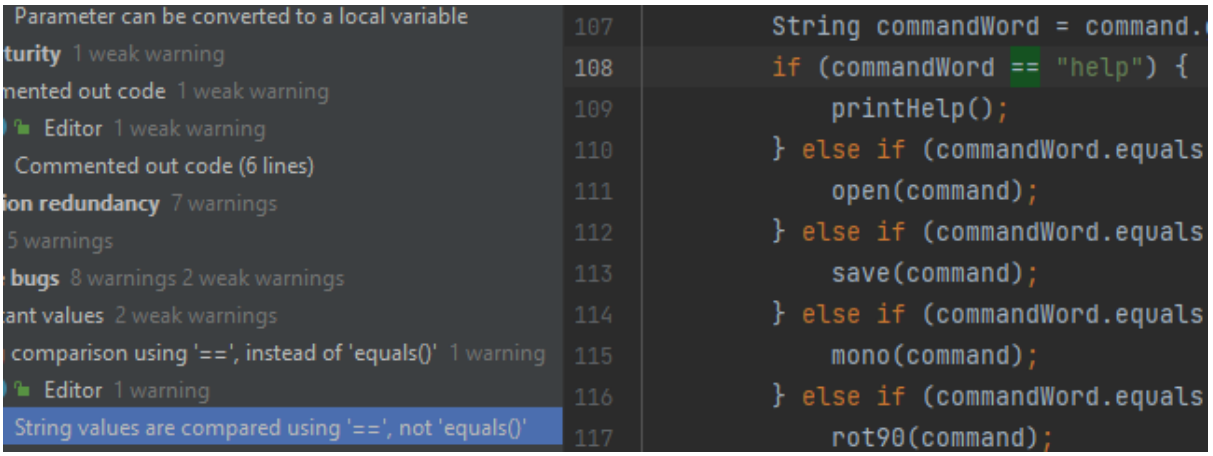

Fig. 5 – The built-in analyser shows an incorrect method for comparing values.

The built-in analyser revealed several issues with the code. One point showed that some String values were compared using '==', which compares the two values in memory. It will always return false as the two don't represent the same object. The proper fix would be to use the `equals()` method, which compares character by character when the two aren't the same object.

4.0 Design Pattern to Refactor the Prototype

Due to the poor implementation of the command functionality, this section will discuss a tangible design pattern for maintaining commands. The obvious choice would be a command design pattern because the prototype already uses commands for its interface.

The purpose of this design pattern is to encapsulate the data required for performing a command in an object (Baeldung, 2022). This allows the interface to be decoupled from the business logic, making it more extensible. This would also make future plans to add a GUI much easier.

The basic pattern consists of an Invoker that passes a request to the commands `execute()` method. An abstract/interface Command class. 'Concrete commands' that inherit the Command class' `execute()` method. And a receiver that carries out the actions/business logic. Not all components apply to a given situation (Digital Ocean, 2022).

Below is a class diagram showing how this pattern could improve the prototype.

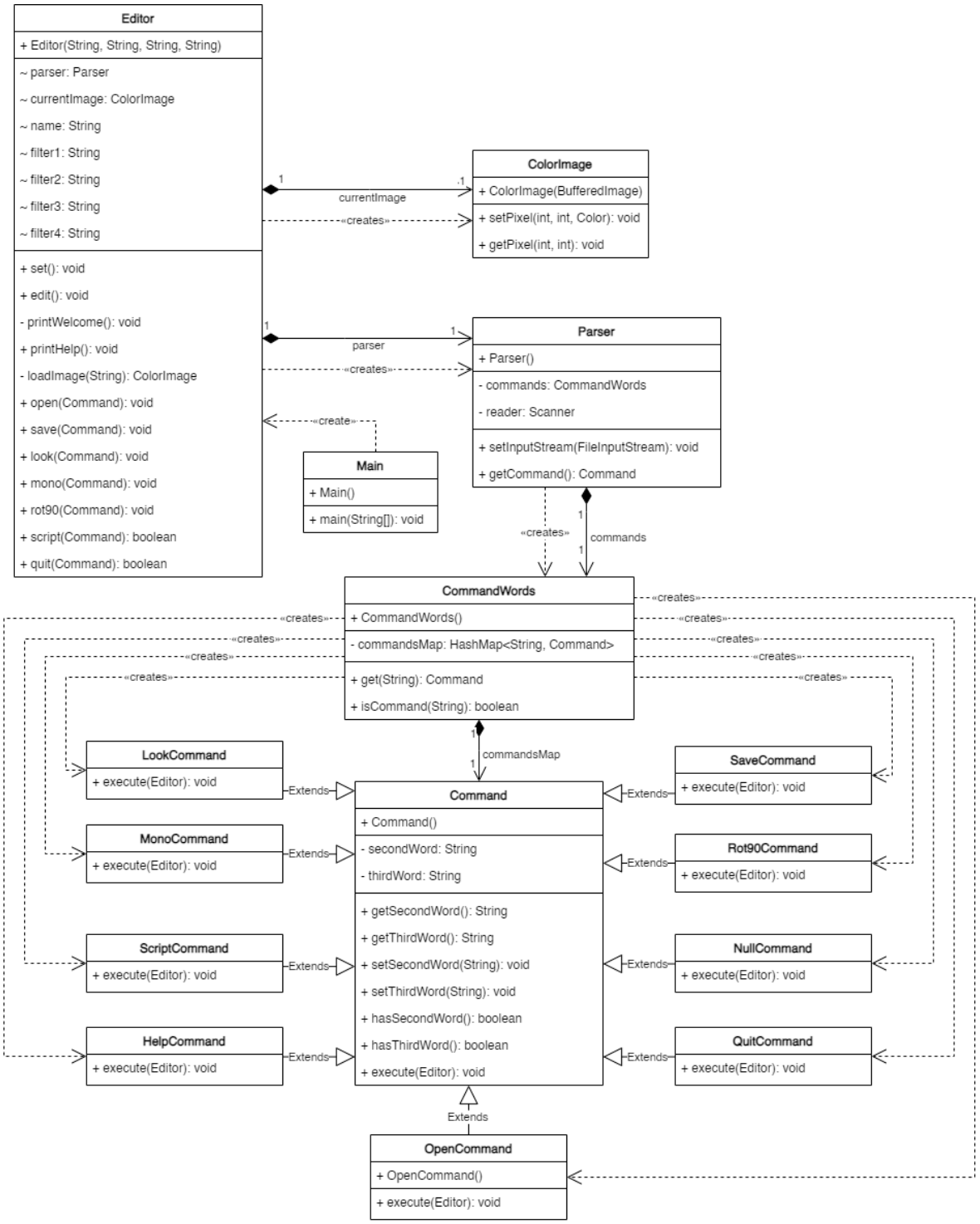

Fig. 6 – Class Diagram depicting the implementation of the Command design pattern.

The CommandWords class acts as the invoker and stores each command in a HashMap. The commands inherit from an abstract Command class and override its `execute()` method. The `execute()` method will then contain method calls to the receiver, which in this case is the Editor.

4.1 Example Implementation in Java

This section shows example commands implemented using the command pattern.

```
public class Rot90Command extends Command {

   @Override
   public void execute(Editor editor) {
       editor.rot90();
   }
}
```

Fig. 7 – Implementation of the Rot90Command class.

```
public class LookCommand extends Command {

   @Override
   public void execute(Editor editor) {
       editor.look();
   }
}
```

Fig. 8 – Implementation of the LookCommand class.

```
public class OpenCommand extends Command {

   @Override
   public void execute(Editor editor) {
       editor.open(this);
   }
}
```

Fig. 9 – Implementation of the LookCommand class.

```
public class MonoCommand extends Command {

   @Override
   public void execute(Editor editor){
       editor.mono();
   }
}
```

Fig. 10 – Implementation of the LookCommand class.

```
public class CommandWords
{
   private HashMap<String, Command> commandsMap = new HashMap();

   /**
    * Constructor - initialise the command words.
    */
   public CommandWords()
   {
       this.commandsMap.put("help", new HelpCommand());
```

```
        this.commandsMap.put("look", new LookCommand());
        this.commandsMap.put("rot90", new Rot90Command());
        this.commandsMap.put("open", new OpenCommand());
        this.commandsMap.put("save", new SaveCommand());
        this.commandsMap.put("mono", new MonoCommand());
        this.commandsMap.put("script", new ScriptCommand());
        this.commandsMap.put("quit", new QuitCommand());
        // Hashmap setup
    }

    /**
     * Get a command from the commandsMap HashMap.
     * @param command The command to retrieve.
     * @return The corresponding command from the HashMap.
     */
    public Command get(String command) { return commandsMap.get(command); }
```

Fig. 11 – Refactoring of the CommandWords class. A HashMap was chosen as it conveniently matched the user's input to the command's keys.

```
public void edit() {
    printWelcome();

    // Enter the main command loop.  Here we repeatedly read commands and
    // execute them until the editing session is over.
    boolean finished = false;
    while (!finished) {
        Command command = parser.getCommand();
        command.execute(this);
    }
    System.out.println("Thank you for using Fotoshop.  Good bye.");
    System.exit(0);
}
```

Fig. 12 – Refactored `edit()` method, removing the reliance on the archaic `processCommand()` method.

The implementation involved making the Command class abstract so each concrete command could inherit from it. Each concrete command overrides the Command class' `execute()` method to tailor it to each command.

Instead of cluttering `execute()` with the command's functionality, method calls are made to the Editor class, which carries out the corresponding action. This separates the interface from the business logic, increasing the prototype's extensibility.

Some refactoring also had to be accounted for, as also documented above.

5.0 Unit Testing

This section will show the use of Unit Testing to expose errors in the system. The test code and the concrete test output of each test will be provided.

5.1 Unit Test #1

```
@Test
public void shouldDisplayHelpOnHelpCommandEntered() throws
       UnsupportedEncodingException {
   String input = "help\nquit\n"; // What commands to run

   ByteArrayInputStream in = new
          ByteArrayInputStream(input.getBytes(StandardCharsets.UTF_8));
   System.setIn(in);
   ByteArrayOutputStream out = new ByteArrayOutputStream();
   System.setOut(new PrintStream(out, true, StandardCharsets.UTF_8));

   Editor instance = new Editor("", "", "", "");
   instance.set();
   instance.edit();

   String output = out.toString(StandardCharsets.UTF_8);
   assertTrue(output.contains("You are using Fotoshop."));
}
```

Fig. 13 – Unit Test asserting the help command will return the appropriate string.

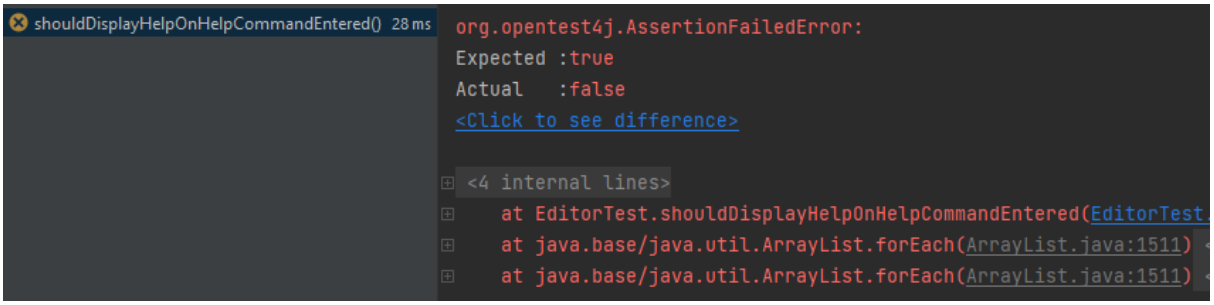

Fig. 14 – Concrete test output exposing an error from using the help command.

5.2 Unit Test #2

```
@Test
public void shouldReturnFailedCommandString() throws
UnsupportedEncodingException {
   String input = "rot90\nquit\n";

   ByteArrayInputStream in = new
           ByteArrayInputStream(input.getBytes(StandardCharsets.UTF_8));
   System.setIn(in);
   ByteArrayOutputStream out = new ByteArrayOutputStream();
   System.setOut(new PrintStream(out, true, StandardCharsets.UTF_8));
```

```
    Editor instance = new Editor("", "", "", "");
    instance.set();
    instance.edit();

    String output = out.toString(StandardCharsets.UTF_8);
    assertTrue(output.contains("I don't know what you mean..."));
}
```

Fig. 15 – Unit Test asserts an error/unknown command string will be returned when trying to use `rot90` before loading an image into `currentImage`.

```
java.lang.NullPointerException: Cannot invoke "ColorImage.getHeight()" because "this.currentImage" is null
    at Editor.rot90(Editor.java:300)
    at Editor.processCommand(Editor.java:117)
    at Editor.edit(Editor.java:61)
    at EditorTest.shouldReturnFailedCommandString(EditorTest.java:59) <29 internal lines>
    at java.base/java.util.ArrayList.forEach(ArrayList.java:1511) <9 internal lines>
    at java.base/java.util.ArrayList.forEach(ArrayList.java:1511) <27 internal lines>


Process finished with exit code -1
```

Fig. 16 – Concrete test output exposing a `NullPointerException` error from trying to use `rot90` before loading an image into `currentImage`.

5.3 Unit Test #3

```
@Test
public void shouldRunFlipHCommand() throws UnsupportedEncodingException {
    String input = "open input.jpg\nflipH\nquit\n";

    ByteArrayInputStream in = new
            ByteArrayInputStream(input.getBytes(StandardCharsets.UTF_8));
    System.setIn(in);
    ByteArrayOutputStream out = new ByteArrayOutputStream();
    System.setOut(new PrintStream(out, true, StandardCharsets.UTF_8));
    Editor instance = new Editor("", "", "", "");
    instance.set();
    instance.edit();
    String output = out.toString(StandardCharsets.UTF_8);
    assertFalse(output.contains("I don't know what you mean..."));
}
```

Fig. 17 – Unit Test asserts the `flipH` command will run because "`I don't know what you mean…`" only appears on unknown commands.

```
org.opentest4j.AssertionFailedError:
Expected :false
Actual   :true
<Click to see difference>

 <4 internal lines>
    at EditorTest.shouldRunFlipHCommand(EditorTest.java:77) <29 internal lines>
    at java.base/java.util.ArrayList.forEach(ArrayList.java:1511) <9 internal lines>
    at java.base/java.util.ArrayList.forEach(ArrayList.java:1511) <27 internal lines>


Process finished with exit code -1
```

Fig. 18 – Concrete test output exposing that `flipH` does not work, although `printHelp()` outputs this as an available command.

6.0 Version Control

To improve the code's maintainability, version control was implemented using Git.

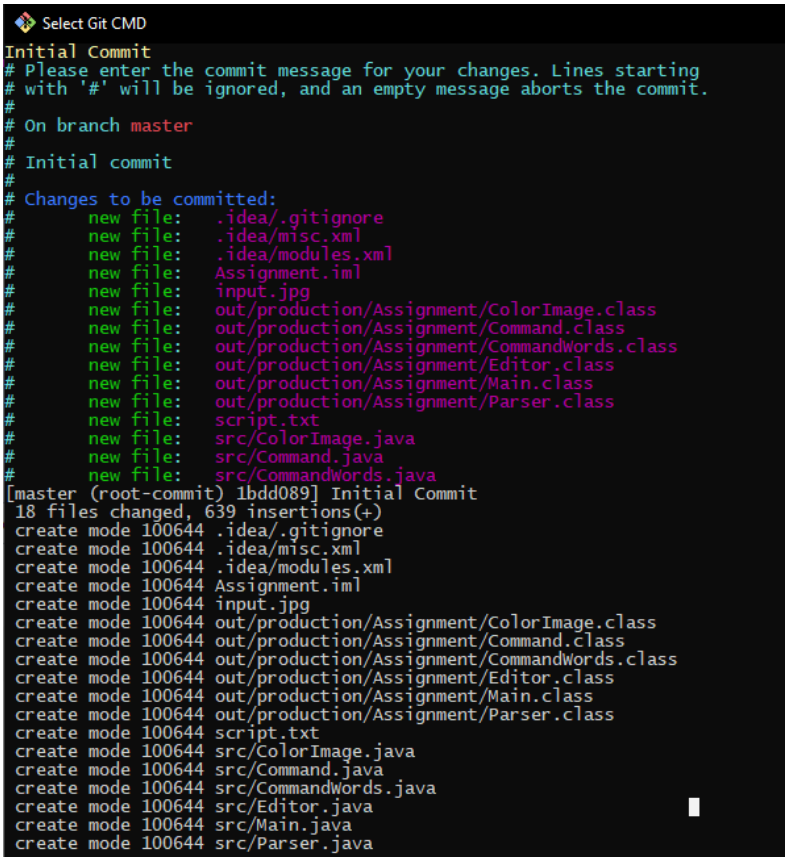

Fig. 19 – Using Git CMD to commit the changes to a repository.

```
D:\Projects\University\6COM\Software Engineering Practice\Assignment>git remote add origin ht

D:\Projects\University\6COM\Software Engineering Practice\Assignment>git branch -M main

D:\Projects\University\6COM\Software Engineering Practice\Assignment>git push -u origin main
Enumerating objects: 25, done.
Counting objects: 100% (25/25), done.
Delta compression using up to 16 threads
Compressing objects: 100% (21/21), done.
Writing objects: 100% (25/25), 407.15 KiB | 33.93 MiB/s, done.
Total 25 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/ascwnyc/SEP-Assignment.git
 * [new branch]      main -> main
Branch 'main' set up to track remote branch 'main' from 'origin'.
```

Fig. 20 – Using Git CMD to push the changes to a private GitHub repository.

A repository was initialised using the Git CMD. Changes were staged and committed to the repository, then pushed to an online private GitHub repository, where future development team members can access it.

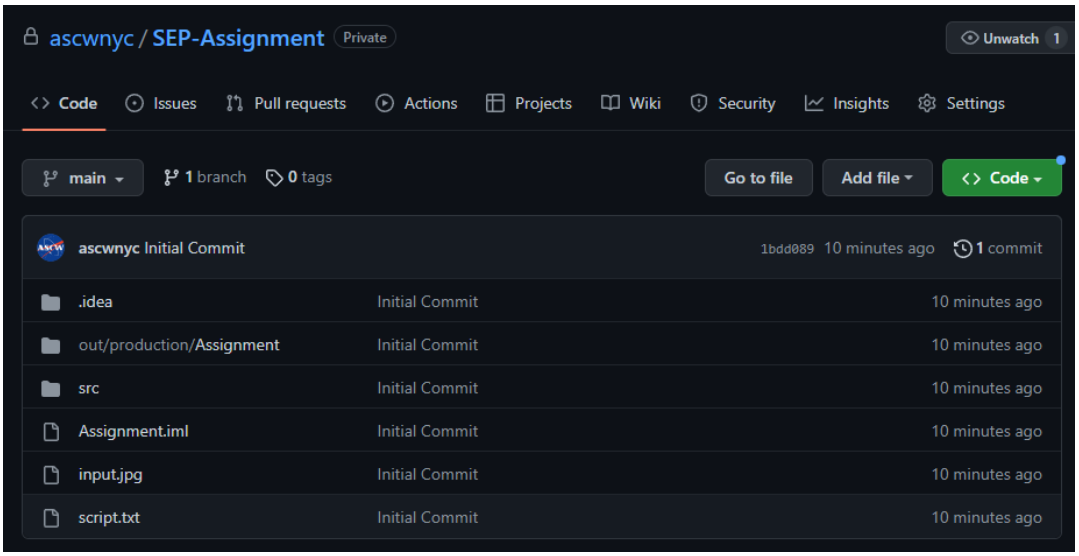

Fig. 21 – The private GitHub repository the changes have been pushed to.

7.0 References

1. Baeldung (2022) *The Command Pattern in Java*. Available at: https://www.baeldung.com/java-command-pattern (Accessed: 10th March 2023).

2. Digital Ocean/Pankaj (2022) *Command Design Pattern*. Available at: https://www.digitalocean.com/community/tutorials/command-design-pattern (Accessed: 11th March 2023).