Java Decompiler Testing Analysis

Map to your heart

Team Member: Chih Fan Chao, Chien Ju Lo, Vinnie Hsia

Table of contents

Chapter 1. Introduction and Functional Testing with Partition

Introduction

Java Decompiler is a very interesting and powerful tool that can be used to decompile the Java.class file back to Java.java file. That is, originally, we cannot understand the .class file, but with this tool, we can turn it back before compiling and know what is the original source code. Below is the GUI of this application.

This application, which size is 62.1 MB and uses a model–view–controller (MVC) architecture pattern. Since this application is written by Java and applies to Java files, we decided to use Junit to test it.

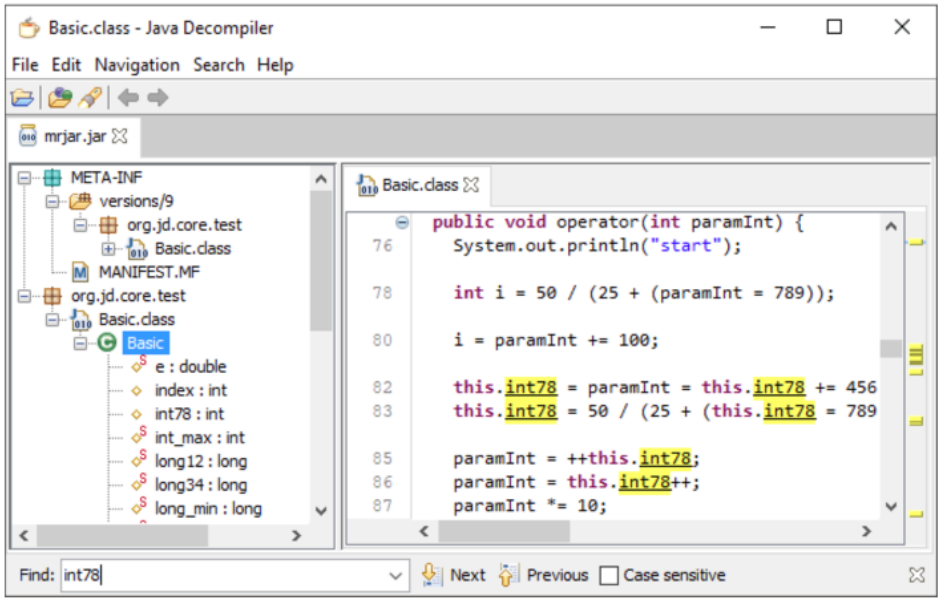


Figure 1. Java Decompiler GUI

Version

JD-GUI: 1.6.6
Java: 11.0.12.7
Junit: 4.12

Set up the environment

After knowing what this Java Decompiler is, we are going to talk about how to set up the environment. The IDE we use is IntelliJ IDEA. This IDE provides a very complete extension for developers and supports cross platforms and multiple languages such as Python, Java, Typescript, etc. You can install Intellij IDEA by referencing this tutorial. To start with, our first step is to fork GitHub from the link: (https://github.com/java-decompiler/jd-gui) to our own repository. Then we can download the code to our local computer with the green button “Code” and Download ZIP or just use “git clone https://github.com/CarolynLo/jd-gui.git “to local as Figure 2.

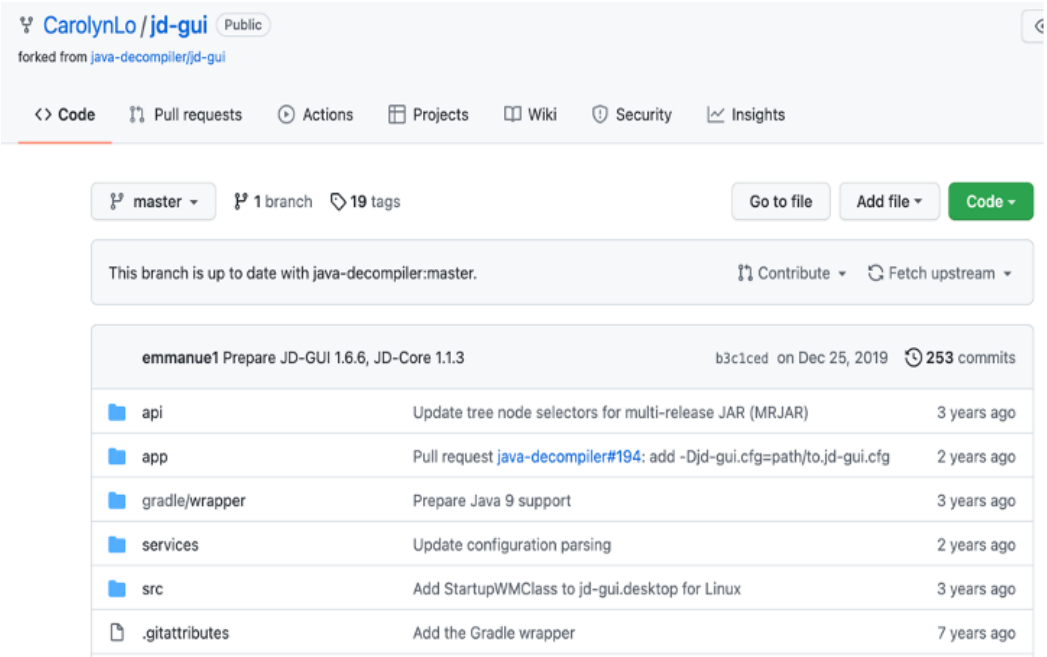


Figure2. Clone Github

When we use the IntelliJ IDEA to open the whole package of the jd-gui, we can press “Build Module jd-gui” with right-click, then this powerful IDE will help us find the Gradle script and build it. After that, we can see the successful building message at the bottom as Figure 3.

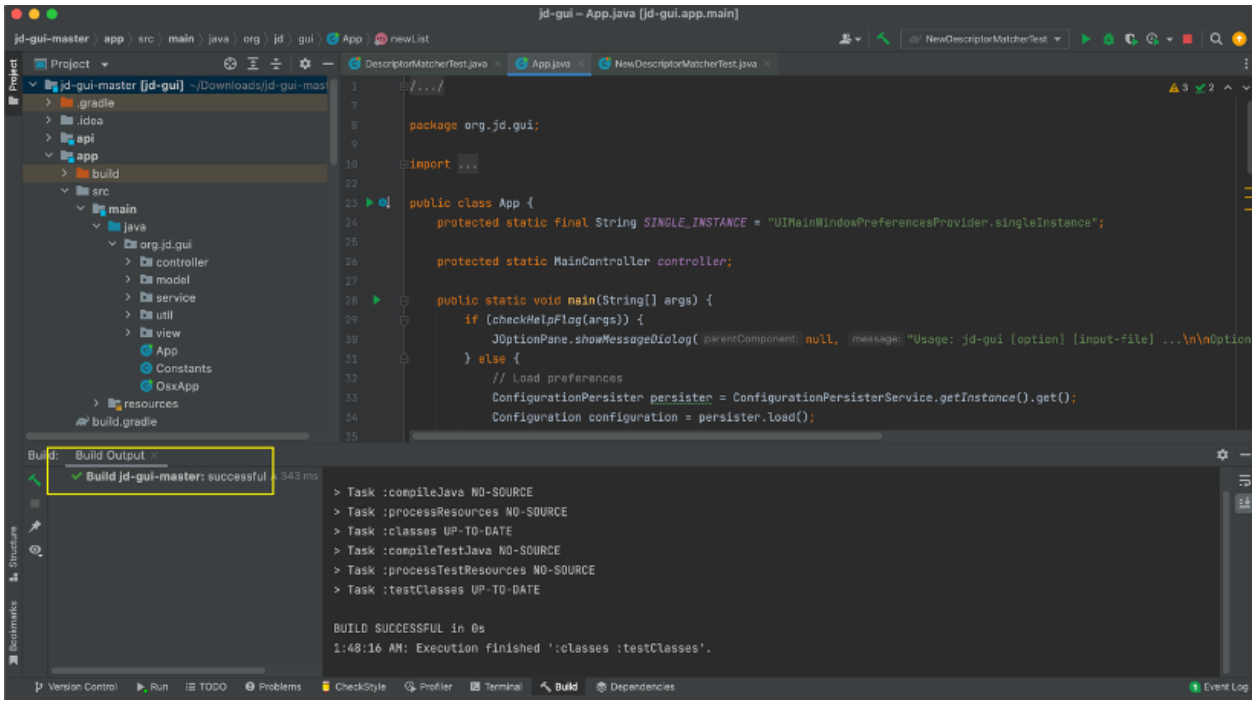


Figure3. Build successful

The last step is to execute the application, which is under the App.java under “org.jd.gui” folder as Figure 4, and then the result will be the GUI as Figure 1.

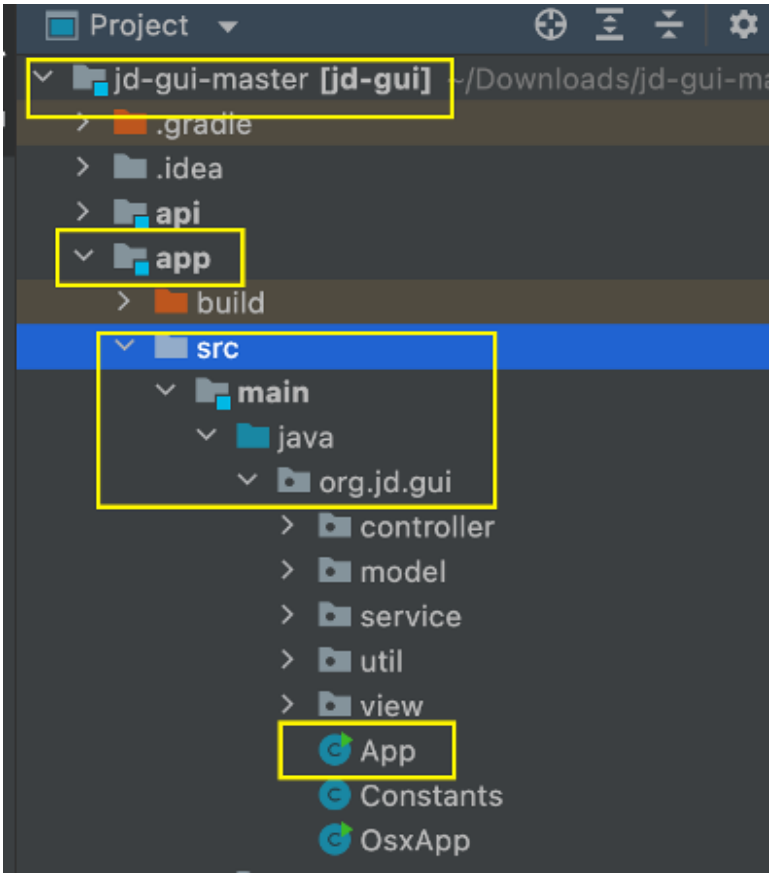


Figure 4. Path to execute the main application

To further confirm the function of the application, we also open a java class file from the local computer, and it does decompile the class file into java code as Figure 5. To be noticed, the .class file which you open needs to have complied with Java SDK older than version 14.

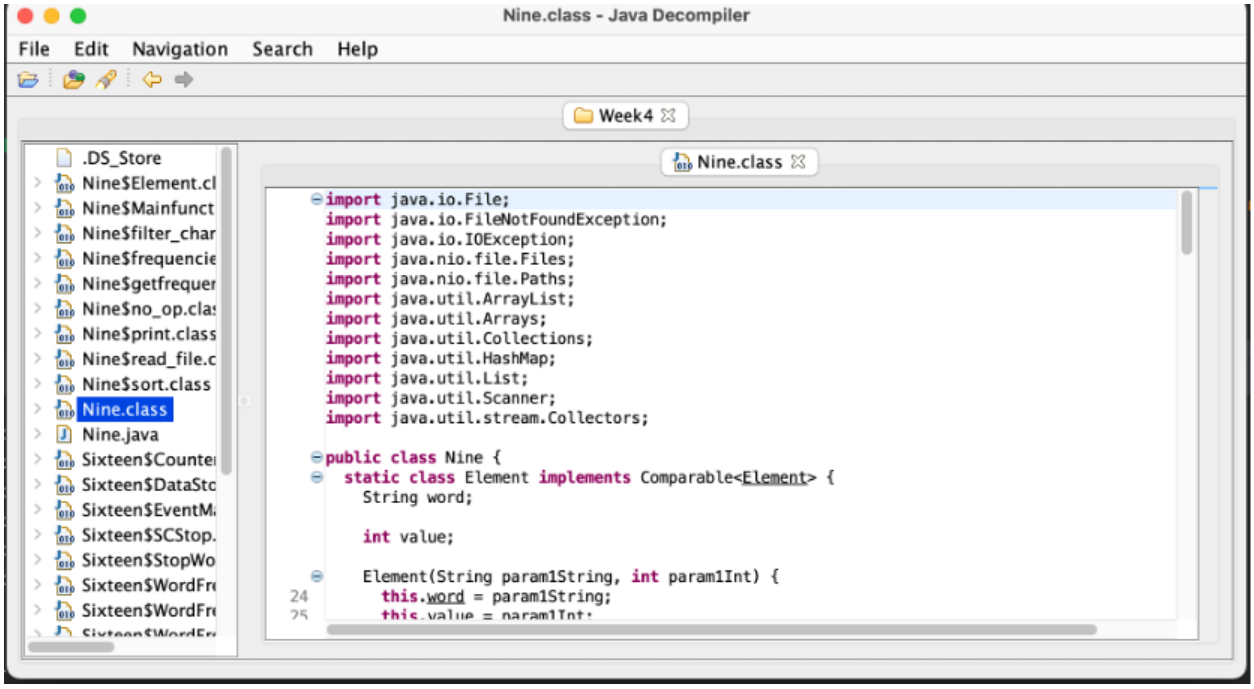


Figure 5. Decompiler result

Explore existing test cases

In order to make sure the tool can work well; we also need to verify the system with existing test cases. For this part, we use the Black Box Testing, which means that the test will ignore the internal mechanism of a system or component and we mainly focus on whether the output is correct. Figure 6 below is the schematic of this kind of testing.

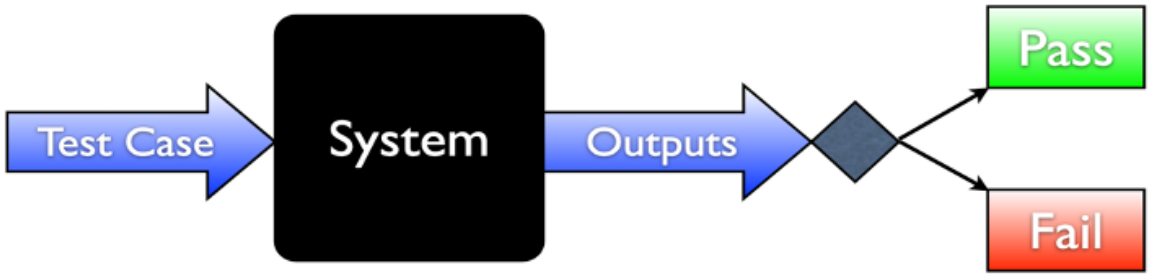


Figure 6. Black Box Testing [2]

For existing test cases, one of the functional tests they have is DescriptorMatcherTest. In this method, they mainly want to do the string comparison. In other words, when we want to test the function, our input will be two strings. After it feeds into our test function, the result will return true if the comparison is the same or fit

the rules, otherwise, it will return false. The schematic diagram is in Figure 7.

Figure 7. Schematic Diagram of the test function

In fact, there are a lot of different conditions when comparing the two strings. If they want to test all the different cases, it would be a huge process. Hence, they also introduce the idea of combinatorial testing. They rearrange the categories of input strings and try to reach the boundary cases. In their rules, the very basic condition is that two strings are the same, then that is the match case, and will assert itself true. Another case is that when the input strings contain “?”, they will have different conditions. If both of the inputs contain “?”, then it is matched. Otherwise, the one that contains the “?” will be skipped and will parse for the next character. Besides “?”, there are also other characters considered as the trigger. While there is “L” or “[“, they will also move to the next character. If at the end of the string is “;”, then it is also true. Otherwise, it will be false just like when ending one line in Java programming, a semicolon is needed. When they test the function, they will test the input cases with those characters or punctuation, then they can directly know what happened when facing those boundaries. Figure 8 shows the sample test cases and it is under the folder of test/java/org.jd.gui/util.matcher.

```
public class DescriptorMatcherTest extends TestCase {
    public void testMatchFieldDescriptors() {
        Assert.assertTrue(DescriptorMatcher.matchFieldDescriptors("?", "?"));

        Assert.assertTrue(DescriptorMatcher.matchFieldDescriptors("I", "I"));
        Assert.assertTrue(DescriptorMatcher.matchFieldDescriptors("?", "I"));
        Assert.assertTrue(DescriptorMatcher.matchFieldDescriptors("I", "?"));
        Assert.assertFalse(DescriptorMatcher.matchFieldDescriptors("I", "A"));

        Assert.assertTrue(DescriptorMatcher.matchFieldDescriptors("Ltest/Test;", "Ltest/Test;"));
        Assert.assertTrue(DescriptorMatcher.matchFieldDescriptors("?", "Ltest/Test;"));
        Assert.assertTrue(DescriptorMatcher.matchFieldDescriptors("Ltest/Test;", "?"));
        Assert.assertTrue(DescriptorMatcher.matchFieldDescriptors("L*/Test;", "Ltest/Test;"));
        Assert.assertTrue(DescriptorMatcher.matchFieldDescriptors("Ltest/Test;", "L*/Test;"));
```

Figure 8. Samples of test cases

Partitioning with new test cases

In this part, we will demonstrate how we develop our own test cases. The feature we want to test is DescriptorMatcher, which is used to do the string comparison. One thing to be noticed in the method is that all descriptors should start with “(“ and when facing “)” means the end of the descriptor. Also, “*” means the wildcard characters which can be matched with anything. After knowing the basic idea of the method, we start to do the partition testing for the function.

We use partition testing, which divides inputs into groups that will show the same results, set up our partition with four different conditions as Figure 9. The boundary values of our partitioning tests are special character “(“, “)” and “*”. After choosing these representatives, which represent the behavior of the method, we can do our test with assertTrue and assertFalse function in Junit to verify the method.

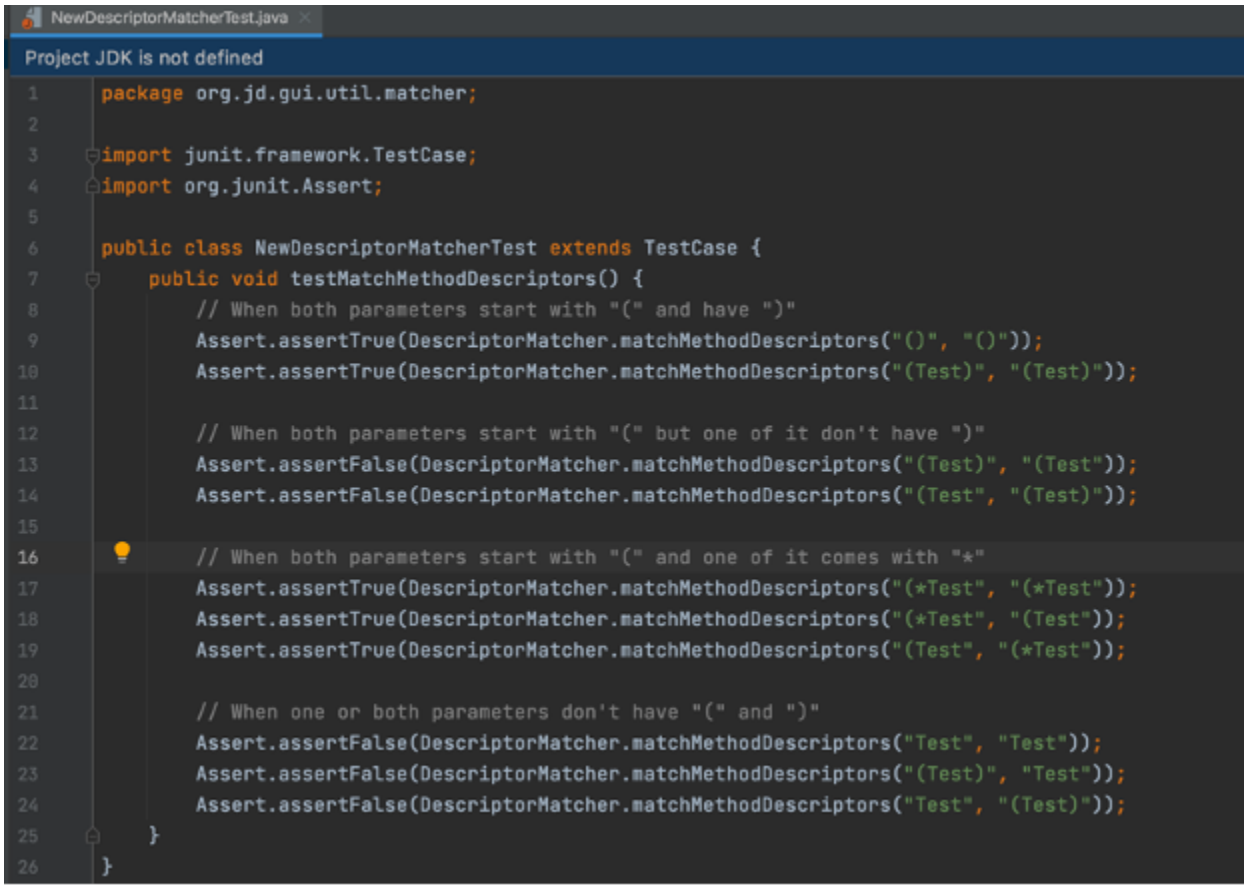


Figure 9. New test java code

1. When both parameters start with "(" and have ")", it will assert true.
2. When both parameters start with "(" but one of them doesn't have ")", it will assert false.

3. When both parameters start with "(" and one of them comes with "*", it will assert true.
4. When one or both parameters don't have "(" and ")", it will assert false.

To further analyze the test cases, we choose systematic testing, which means that we select our inputs that are valuable and not just uniformly. The advantage of the non-uniform test case is that we can avoid missing the logic faults. Finally, Figure 10 shows the new test cases result successfully.

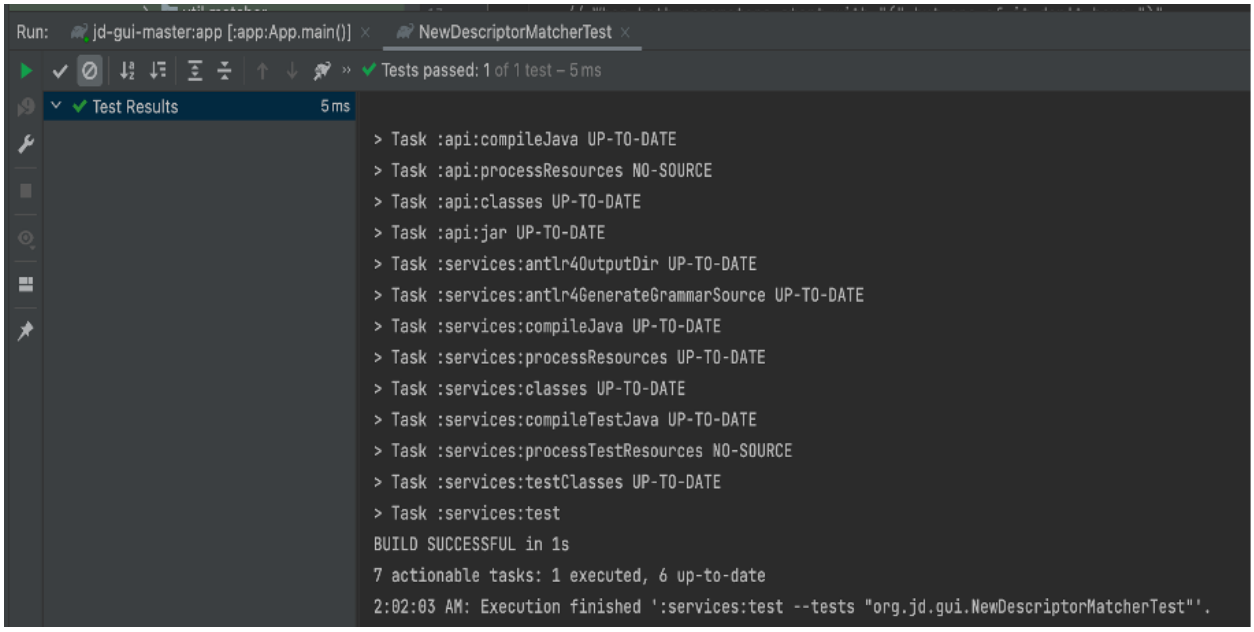


Figure 10. New test cases result

Reference:

1. https://github.com/java-decompiler/jd-gui
2. http://java-decompiler.github.io/
3. 2022-01-13-SWE261P-QA_Principles_and_Testing_Fundamentals_2.pdf
4. https://www.codejava.net/ides/intellij/how-to-download-and-install-intellij-idea

Chapter 2. Functional Testing and Finite State Machines

Introduction

Before we start doing further tests of our methods, we want to take a look at our software's behavior. One very useful technique is to use the finite model. A finite model is a method to describe the system's behavior while this model is compact and predictive. With selected inputs, we can get the results from the model and hence assess whether our software is good or bad. The core idea of the finite model is to draw a finite state machine diagram to do the program analysis and understand the logic and behavior of the software system. If we know how it works, then we can check whether our code meets the requirements. Below we will first introduce how we choose the feature of a finite state machine and then we will do some analysis. The last part will be some JUnit tests to verify our ideas about the finite state machine.

Analysis of Finite State Machines

To begin with, we want to introduce how we choose the feature for finite state machines. A finite state machine is a set of states and a set of transitions, and it can be a directed graph. That is, to represent the software with a finite state machine, we first need to define the states of our software method, then we can have different actions to help us change one state to another state. Hence, to choose a good feature, we want the method to have states that can be transformed. After going through the code and the GUI, we think History.java is a good choice because it can be separated into five states and can use add, backward, and forward to make the valid transition. Figure 1 shows the finite state machine diagram and we will describe details below.

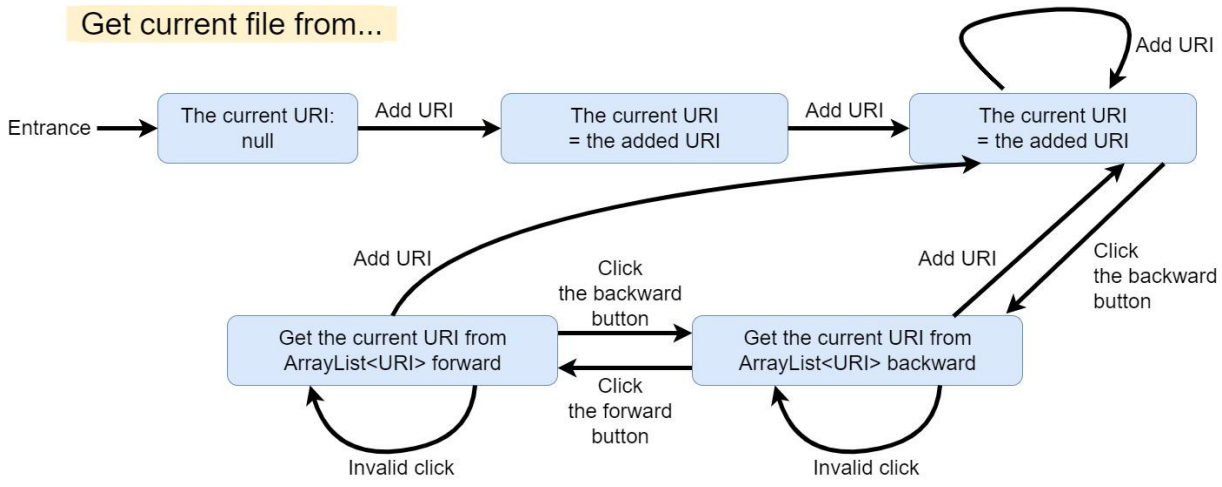


Figure 1. Diagram of Finite State Machines for History.java

The method we choose is “History” or you can reference at this link. This method is used to record all the files that were opened in order. That is, there are two lists, forward list and backward list, to save the URI of the files. For the backward list, it will record the newest URI at the last element, which is like a queue. It means that when we open a new file after opening another file before, then this new file will be at the last element in this backward list. The other list is a forward list, which is used for recording the previous URI. These lists will become our nodes in the finite state machines. To change our states in the finite state machines, we will have different statements to help us do the transition.

In our example, we will have “add”, “backward” and “forward”. When we “add”, it means we will have a new file opened, and the URI of this file will be stored into the backward list. When we click “backward”, it means that we will move back to our previous file. Hence, we will change our state from backward to forward, and then the console will show the last URI in the forward list. Another condition is that when we click “forward”, we will move our state from previous to current, which means that we will go back to the backward list again, and the console will show us the newest URI. Other actions such as “invalid click” will not change the current state.

JUnit test for Functional Models

After understanding our finite state machine, we want to use some JUnit tests to check if our thoughts of states and transitions are correct. The feature we want to test is History.java, which determines the displaying file. We will demonstrate how we develop our test cases and test each statement of finite state machines as Figure 2 below.

```
public void testCurrentFileUri(){
    History history  = new History();
    // Current history will be null at the beginning
    Assert.assertNull(history.current);

    // Add the first uri
    URI uriA = new File("src/test/files/Five.class").toURI();
    history.add(uriA);
    Assert.assertEquals(uriA, history.current);
    Assert.assertFalse(history.canBackward()); // no backward
    Assert.assertFalse(history.canForward()); // no forward

    // Add the second uri after the first uri
    URI uriB = new File("src/test/files/Four.class").toURI();
    history.add(uriB);
    Assert.assertEquals(uriB, history.current);
    Assert.assertFalse(history.canForward()); // no forward
    Assert.assertTrue(history.canBackward()); // can backward
    Assert.assertEquals(uriA, history.backward()); // backward get the previous uri
    Assert.assertTrue(history.canForward()); // can forward after go back to the first uri
    Assert.assertEquals(uriB, history.forward()); // forward get the next uri

    // Add the third uri after the first uri, the second uri will be cleared
    history.backward();
    URI uriC = new File("src/test/files/TermFrequency.class").toURI();
    history.add(uriC);
    Assert.assertEquals(uriC, history.current);
    Assert.assertFalse(history.canForward()); // no forward
    Assert.assertTrue(history.canBackward()); // can backward
    Assert.assertEquals(uriA, history.backward()); // backward get the previous uri
    Assert.assertTrue(history.canForward()); // can forward after go back to the first uri
    Assert.assertEquals(uriC, history.forward()); // forward get the next uri
}
```

Figure 2. Test cases for History.java

For case1: We add uriA

Initially, we do not have any files. Then, after adding uriA, we can get our URI as uriA. Besides, ArrayList<URI> backward and ArrayList<URI> forward are all empty, so canBackward() and canForward() should return false which means that there is nothing to move forward or move backward.

For case2: We add uriB.

Now, the state becomes the second “The current URI = the added file”. We can click the backward button, but we cannot click the forward button. Hence, canBackward() should return true and canForward() should return false. Besides, when we click the backward button, the state becomes “Get the current URI from ArrayList<URI> backward”. Thus, backward() should return uriA, which just saves in ArrayList<URI> backward. Then, uriA will be removed from ArrayList<URI> backward and uriB will be stored to ArrayList<URI> forward. After that, we can click the forward button, then the state becomes “Get the current URI from ArrayList<URI> forward”. At this moment, we can use forward() to get the uriB, which is just saved in ArrayList forward.

For case3: We add uriC after backward to uriA.

Now, the state goes back to the second “The current URI = the added file”. We can click the backward button, but we cannot click the forward button. It is because we add uriC when the current file is uriA. It means that the uriB, which is stored in ArrayList<URI> forward, will be cleared. Hence, canBackward() should return true and canForward() should return false.

Besides, when we click the backward button, the state becomes “Get the current URI from ArrayList<URI> backward”. Thus, backward() should return uriA, which is the previous URI. After that, we can click the forward button, then the state becomes “Get the current URI from ArrayList<URI> forward”. At this moment, we can use forward() to get the next URI, which should return uriC.

Conclusion

In our finite state machines, when we do not have anything in the URI, we cannot click forward or backward. However, after we open files, we can have the records in the forward list and backward list. As long as the lists are not null, click actions such as “Forward” or “Backward” can be executed. The state will change to the URI depending on the latest time we have in our list, just like the concept of cache. With these states and transitions, we can know how our system works when opening and reading files. Finally, Figure 3 shows our Finite State Machine works successfully and it meets our expectations as the diagram.

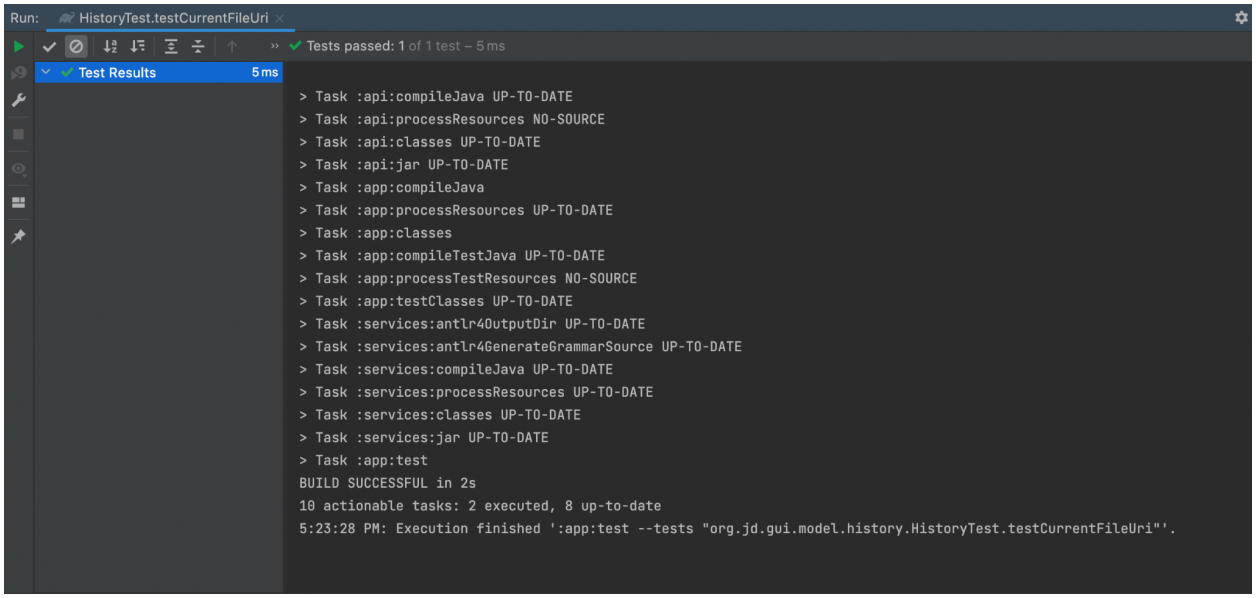


Figure 3. Final result for the JUnit test

Reference

1. Finite State Machine idea: 2022-01-27-SWE261P-Finite_Functional_Models.pdf
2. Definition of Finite Model for introduction part: https://www.guru99.com/model-based-testing-tutorial.html?fbclid=IwAR2yf1JwQfK3JphQe-8pKTQ_M8Vy5vVYb2OP61mEPB4XJdMe8mt8X0Em7Ic

Chapter 3. White Box Testing and Code Coverage

Introduction

This time, we want to introduce the idea of Structural testing. Structural testing is also known as code-based testing. Compared to functional testing, it focuses more on the structure of the code, instead of the correctness of function. That is, we will use different methods to analyze whether the test suites cover the function and which part is missing. The reason that we want to have this kind of testing is that sometimes even if you have the right test result, it does not mean that all the parts in the code are verified. There may exist some conditions that you do not take into consideration and structural testing can help us find what can be improved. After a rough introduction, following we will talk about the current coverage condition and new improvement testing. First of all, Figures 1, 2, and 3 show five flowcharts of five methods in History.java.

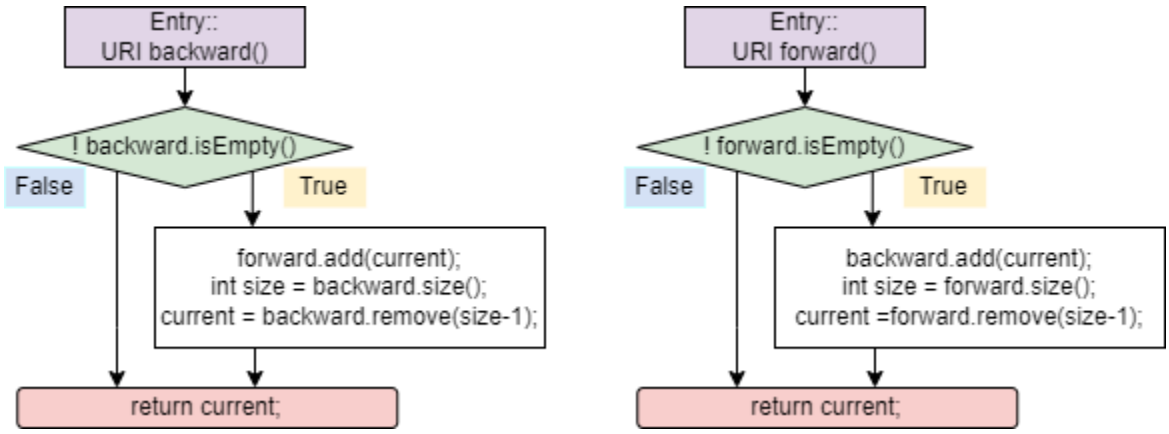


Figure 1. Flowchart of the functions: URI backward() and URI forward()

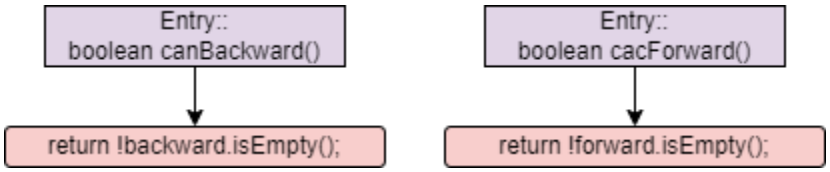


Figure 2. Flowchart of the functions: boolean canBackward() and boolean canForward()

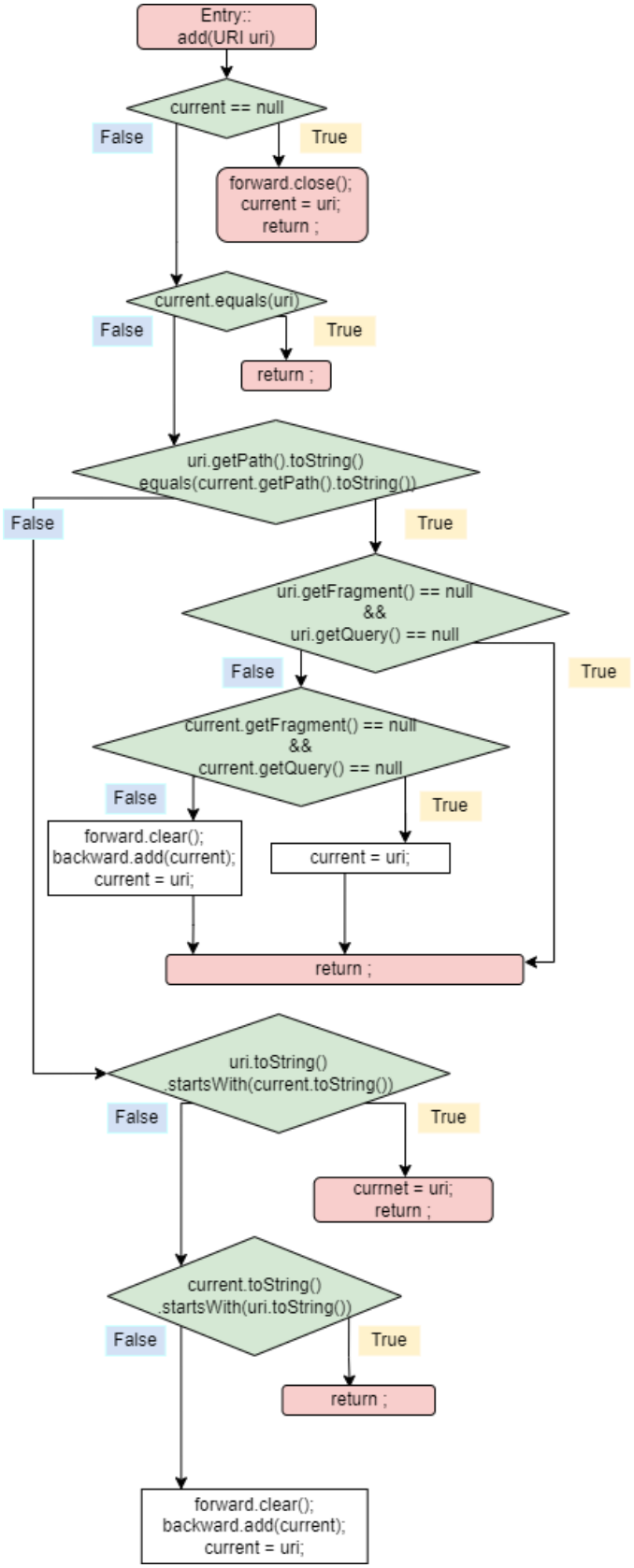


Figure 3. Flowchart of the function: add(URI uri)

Code Coverage with current test suite

In this part, we have three kinds of code coverage analysis: line, branch and method. We will introduce it one by one and we will analyze the method in the code with our original test suite written here. To start with, we first analyze the code coverage with lines.

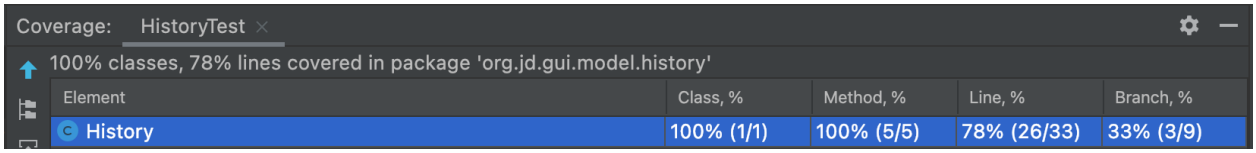


Figure 4. Code coverage in IntelliJ

1. Line coverage

The definition of line coverage is how many lines are covered in the code when doing the testing. Our total lines in History.java is 33 and when running the HistoryTest.java, it is covered with 26 lines, and the percentage is 78%. Figure 5. shows the result for Line coverage. The red bar means it does not cover and the green bar means it covers the code.

```
/.../

package org.jd.gui.model.history;

import java.net.URI;
import java.util.ArrayList;

public class History {
    protected URI           current = null;
    protected ArrayList<URI> backward = new ArrayList<>();
    protected ArrayList<URI> forward = new ArrayList<>();

    public void add(URI uri) {
        if (current == null) {
            // Init history
            forward.clear();
            current = uri;
            return;
        }

        if (current.equals(uri)) {
            // Already stored -> Nothing to do
            return;
        }

        if (uri.getPath().toString().equals(current.getPath().toString())) {
            if ((uri.getFragment() == null) && (uri.getQuery() == null)) {
                // Ignore
            } else if ((current.getFragment() == null) && (current.getQuery() == null)) {
                // Replace current URI
                current = uri;
            } else {
                // Store URI
                forward.clear();
                backward.add(current);
                current = uri;
            }
            return;
        }
```

```
45
46
47          if (uri.toString().startsWith(current.toString())) {
48              // Replace current URI
49              current = uri;
50              return;
51          }
52
53          if (current.toString().startsWith(uri.toString())) {
54              // Parent URI -> Nothing to do
55              return;
56          }
57
58          // Store URI                                          ⚠ 3  ✓ 1
59          forward.clear();
60          backward.add(current);
61          current = uri;
62      }
63
64      public URI backward() {
65          if (! backward.isEmpty()) {
66              forward.add(current);
67              int size = backward.size();
68              current = backward.remove( index: size-1);
69          }
70          return current;
71      }
72
73      public URI forward() {
74          if (! forward.isEmpty()) {
75              backward.add(current);
76              int size = forward.size();
77              current = forward.remove( index: size-1);
78          }
79          return current;
80      }
81
82      public boolean canBackward() { return !backward.isEmpty(); }
83      public boolean canForward() { return !forward.isEmpty(); }
84  }
```

Figure 5. Result of line coverage

2. Branch coverage

The definition of branch coverage is how many branches are executed and the branch represents a labeled edge in the CFG. Our total branches are 9 in History.java and our test suite covers 3 branches. It can be shown in Figures 6 and 7. Figure 7. shows the functions URI backward() and URI forward(), since we check the lists every time before we call the function, the False branches of these two functions are not tested.

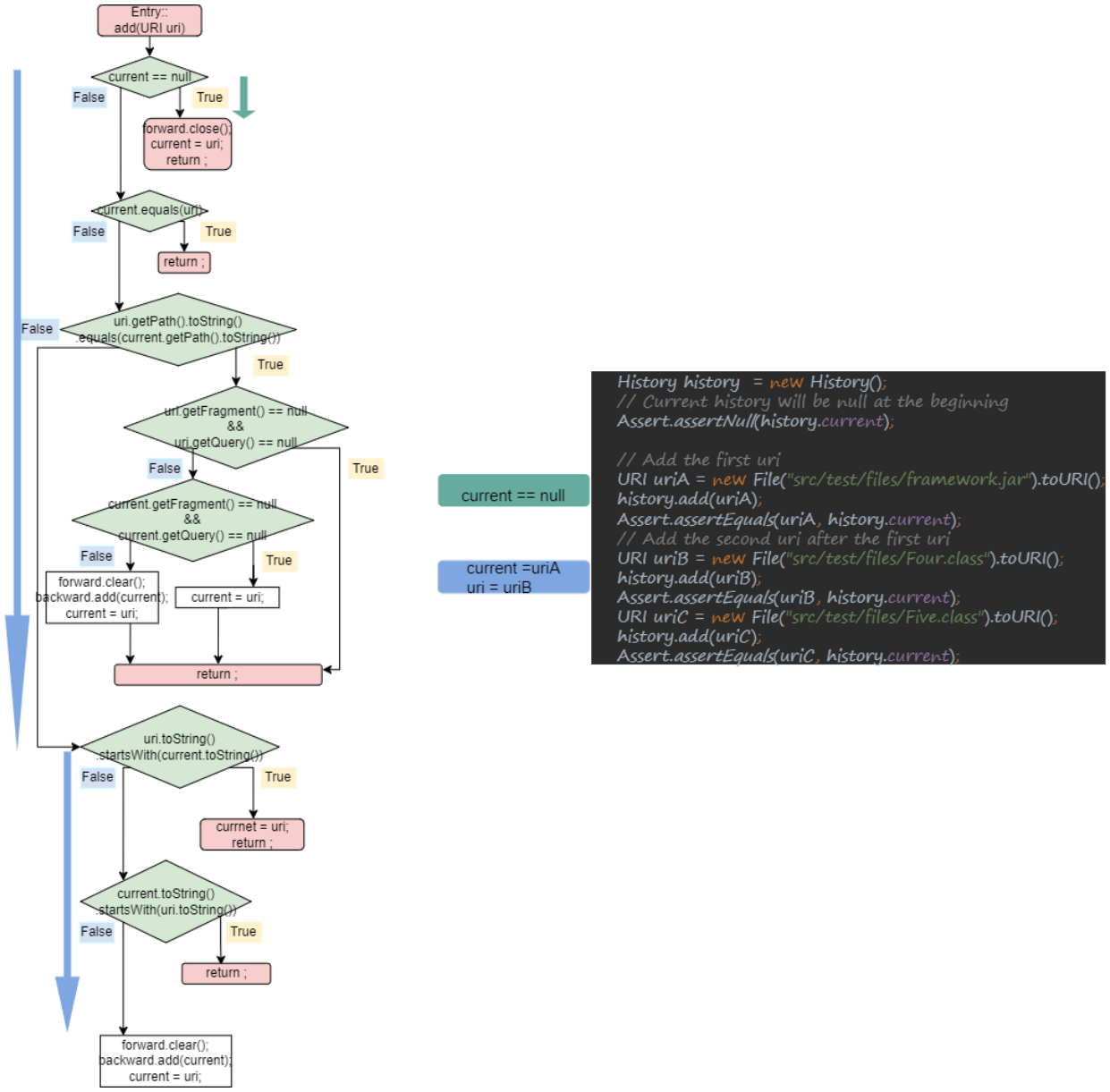


Figure 6. Result of branch coverage - add(URI uri)

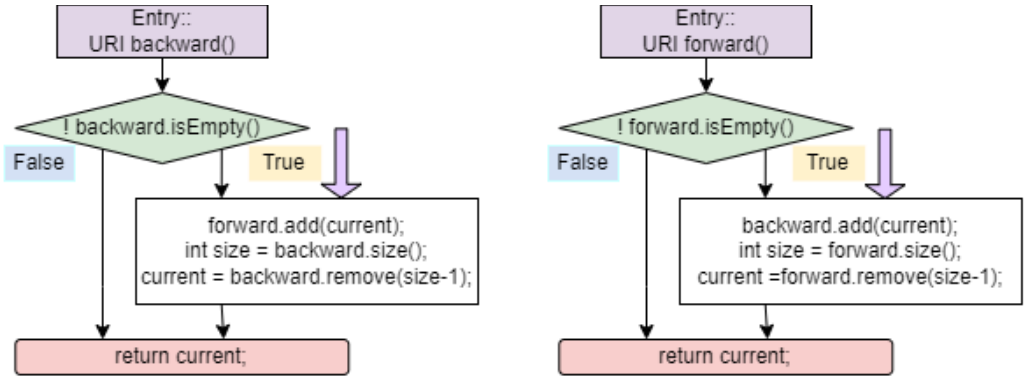


Figure 7. Result of branch coverage - URI backward() and URI forward()

3. Method coverage

The definition of method coverages is how many methods are called and executed at least once during the testing. From Figure 4, we can see that our testing already covers all the methods in the History.java, which means we have 100 percent to call the five methods: add(), backward(), forward(), canBackward() and canForward()and execute it at least one time in our test cases.

Code Coverage with improvement testing

In this part, we wrote new test cases to improve the coverage. The original test suite only determines the current opening files that their URIs should store in the right places. For example, the latest adding URI should equal the current URI, and if there is a previously opened file, its URI will be stored to a backward list. If we press the backward button, the latest added URI in the backward list will become the current URI and be removed from the list. Then, the previous current URI will be added to the forward list. Same as the backward button, if we press the forward button, the latest added URI in the forward list will become the current URI and be removed from the list. Then, the previous current URI will be added to the backward list.

However, the original test suite can only test when the test cases are in different references. Therefore, we add new test cases as Figure 8. to check how it works when adding files in the same reference. When we add a new compress data file first, then add a specific file, which is included in the compress data file, the URI of the compress data file will not be added to the backward list.

```
/*
 * If just open a compress data file, the uri will be added to the history.
 * But if navigate/search/open the file in compress data files,
 * it will only add the file instead of the compress data file
 */

// Navigate specific file in compress data file
history.backward();
Assert.assertEquals(uriA, history.current);
URI uriD = new URI("/User/Test/framework.jar#Framework");
history.add(uriD);
Assert.assertEquals(uriD, history.current);
Assert.assertFalse(history.canBackward()); // no backward because the previous uri is
its compress data file
// Do nothing if not navigating a specific file
history.add(uriA);
Assert.assertNotEquals(uriA, history.current);
Assert.assertEquals(uriD, history.current);
// Navigate another specific file in same compressed data file
URI uriE = new URI("/User/Test/framework.jar#Main");
history.add(uriE);
Assert.assertEquals(uriE, history.current);
Assert.assertFalse(history.canForward()); // no forward
Assert.assertTrue(history.canBackward()); // can backward
Assert.assertEquals(uriD, history.backward()); // backward get the previous uri
Assert.assertTrue(history.canForward()); // can forward after go back to the first uri
Assert.assertEquals(uriE, history.forward()); // forward get the next uri
```

```
// Search specific file in compress data file
URI uriF = new URI("/User/Test/interface.jar");
history.add(uriF);
Assert.assertEquals(uriF, history.current);
URI uriG = new URI("/User/Test/interface.jar?IWord");
history.add(uriG);
Assert.assertEquals(uriG, history.current);
// Do nothing if not searching a specific file
history.add(uriF);
Assert.assertNotEquals(uriF, history.current);
Assert.assertEquals(uriG, history.current);
// Search another specific file in same compressed data file
URI uriH = new URI("/User/Test/interface.jar?IFrequency");
history.add(uriH);
Assert.assertEquals(uriH, history.current);
Assert.assertFalse(history.canForward()); // no forward
Assert.assertTrue(history.canBackward()); // can backward
Assert.assertEquals(uriG, history.backward()); // backward get the previous uri
Assert.assertTrue(history.canForward()); // can forward after go back to the first uri
Assert.assertEquals(uriH, history.forward()); // forward get the next uri

// Open specific file in compress data file
URI uriI = new URI("/User/Test/example.jar");
history.add(uriI);
Assert.assertEquals(uriI, history.current);
URI uriJ = new URI("/User/Test/example.jar!/main.class");
history.add(uriJ);
Assert.assertEquals(uriJ, history.current);
Assert.assertNotEquals(uriI, history.backward()); // the compress data file will not
be store to backward list
```

Figure 8. New Test cases for History.java

For case 1: We navigate the specific file in the compressed data file.

For now, the current URI is uriA, which is a jar file. If we navigate uriD, which is a class name inside the jar file, the current URI will become uriD, and uriA will not be added to ArrayList<URI> backward since uriD is one of the files in uriA. If we then add again uriA, it will do nothing since the current URI is in uriA. But if we navigate uriE, another class in uriA, the current URI will change to uriE, and uriD will be added to ArrayList<URI> backward.

For case 2: We search the specific file in the compressed data file.

We add a new jar file, which is uriF, first. If we search uriG, which is a class name inside the jar file, the current URI will become uriG, and uriF will not be added to ArrayList<URI> backward since uriG is one of the files in uriF. If we then add again uriF, it will do nothing since the current URI is in uriF. But if we search uriH, another class in uriF, the current URI will change to uriH, and uriG will be added to ArrayList<URI> backward.

For case 3: We open the specific file in the compressed data file.

We add another jar file, which is uriI, first. Then, if we open a specific file inside the uriI, which is uriJ, the current URI will become uriJ, and uriI will not be added to ArrayList<URI> backward since uriJ is one of the files in uriI.

Conclusion

The new test cases work successfully and meet our expectations as the diagram in Figure 9. With the new test cases, we improved the branch coverage from 3/9 to 9/13, reaching out to 69%, shown in Figure 10. Also improved the line coverage to 100%, shown in Figure 11. We found that if we just open a compressed file, the uri will be added to the history. But if we open, navigate, or search for a specific file after we open the compressed file, the URI of the compressed file will not be recorded. Finally, Figure 12 shows the result of line coverage in IntelliJ after the new test cases.

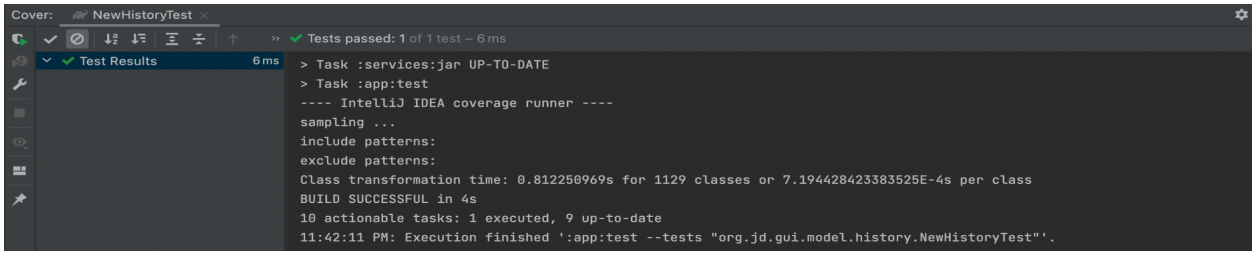


Figure 9. Test cases passed

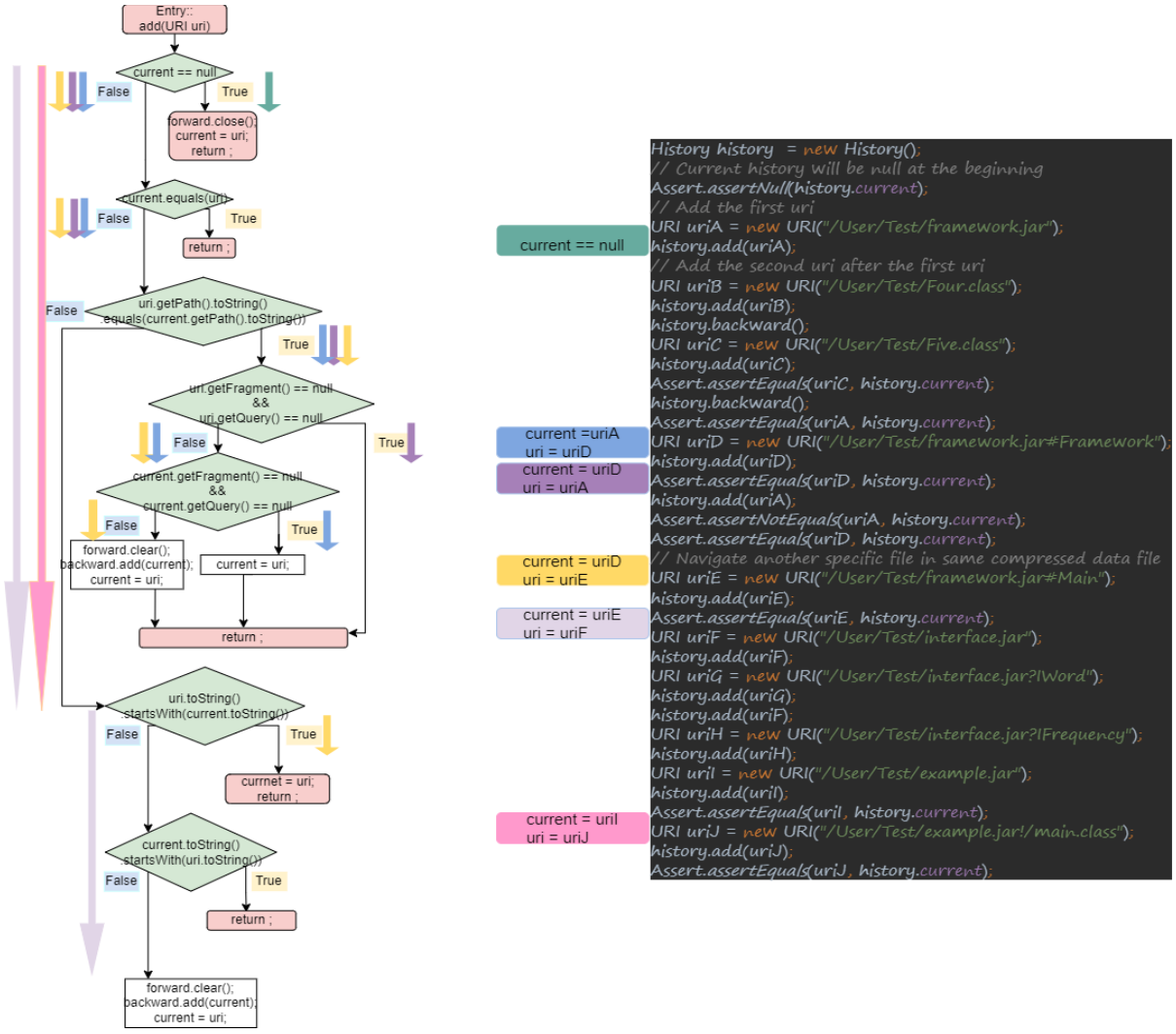


Figure 10. Result of branch coverage after improved IntelliJ

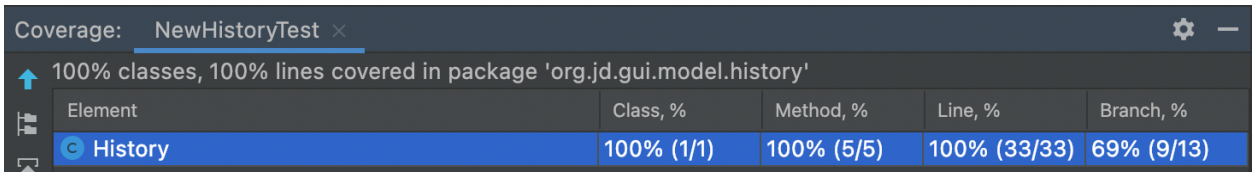


Figure 11. Improved code coverage in IntelliJ

```
31      if (uri.getPath().toString().equals(current.getPath().toString())) {
32          if ((uri.getFragment() == null) && (uri.getQuery() == null)) {
33              // Ignore
34          } else if ((current.getFragment() == null) && (current.getQuery() == null)) {
35              // Replace current URI
36              current = uri;
37          } else {
38              // Store URI
39              forward.clear();
40              backward.add(current);
41              current = uri;
42          }
43          return;
44      }
45
46
47      if (uri.toString().startsWith(current.toString())) {
48          // Replace current URI
49          current = uri;
50          return;
51      }
52
53      if (current.toString().startsWith(uri.toString())) {
54          // Parent URI -> Nothing to do
55          return;
56      }
```

Figure 12. Result of line coverage after improved

Chapter 4. Continuous Integration

Introduction of continuous integration

Continuous integration is a smart way for people to manage and maintain their codes, especially when the projects are large or there are lots of people co-work in the projects. With the idea of continuous integration, people can see when the codes are committed and can build, test, and deploy the codes automatically. This helps integrate the projects and the codes.

Tool-GitHub Actions

GitHub Actions is one of the Continuous Integration Tools, which can be used in GitHub. Thus, we do not need to worry about the environment of our program. It can build, test, and deploy our code from our GitHub repository [1]. That is why we chose this tool instead of Circle CI and Travis.

Figure1. The interface of Github Actions

Process

Followings are the steps about how to use Github Actions:

- Sign in to our GitHub account
- Click Actions
- Since our project is a Java Gradle project, we choose “Java with Gradle”
- After choosing “Java with Gradle” as Figure2, it automatically generates a “workflows” folder
- Now, we can add more files to the folder [2], as Figure3

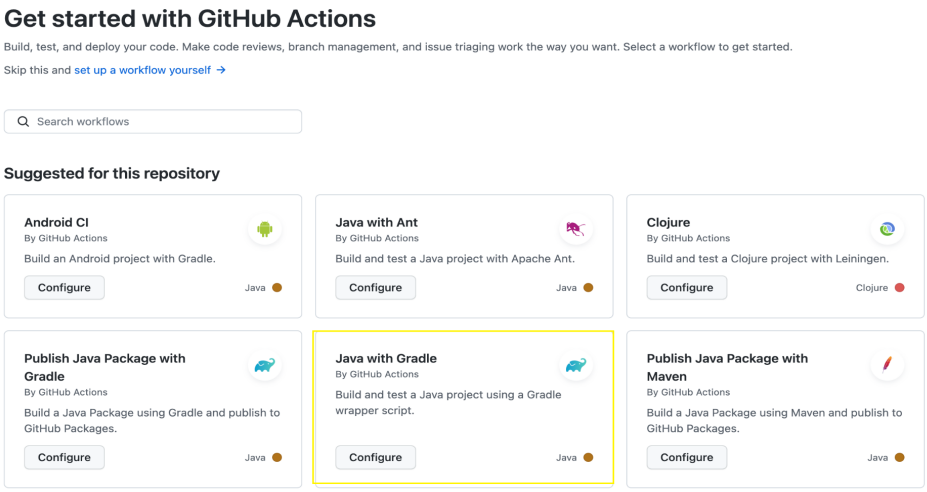


Figure2. Java with Gradle

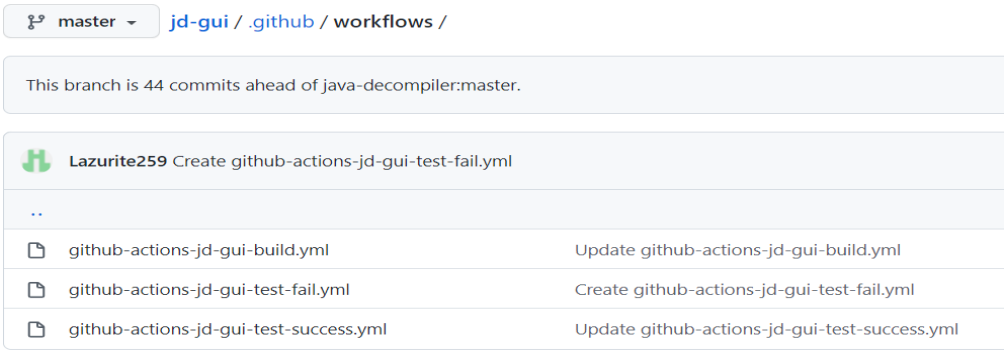


Figure3. Three .yml files in the folder “workflows”

Followings are some conditions that we test:

- Condition1: Build
  - To build the project, we can use “./gradlew build --info” to build.
  - Issue: We notice that as long as there is a fail test in the project, we will fail on the build.

```yaml
name: GitHub Actions for jd-gui build
on: [push]
jobs:
  Explore-GitHub-Actions:
    runs-on: ubuntu-latest
    steps:
      - name: Check out repository code
        uses: actions/checkout@v2
      - name: Test Build
        run: ./gradlew build --info
```

Figure4. "Build” command in the file

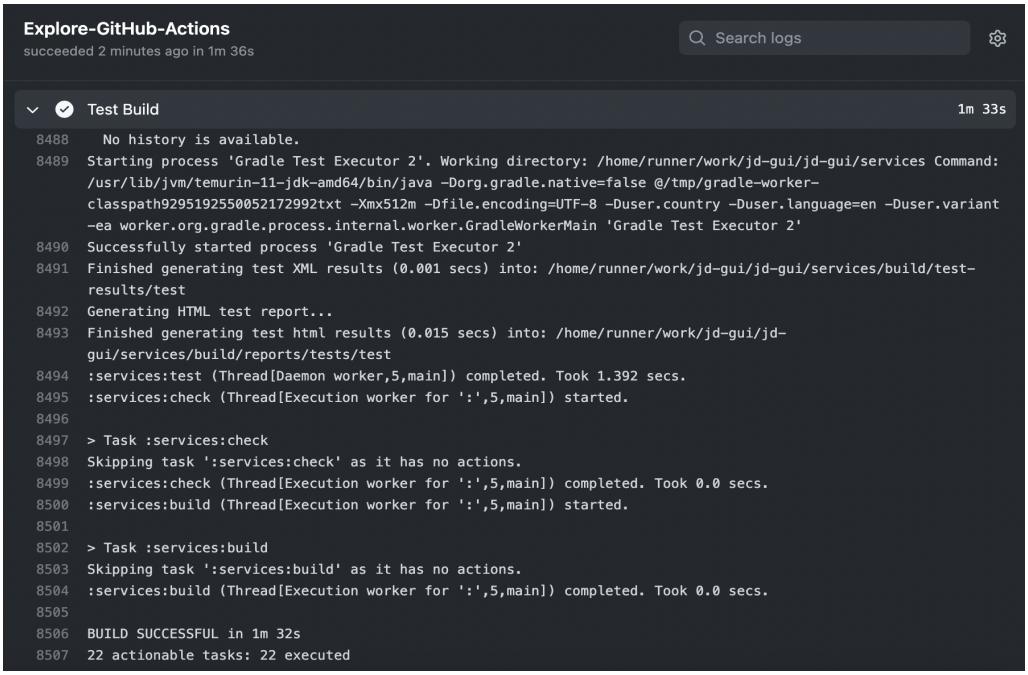


Figure5. Result of the command - build

- Condition2: Run successful test case
  - To test for a successful case, we run with “./gradlew :app:test --tests org.jd.gui.model.history.NewHistoryTest --info” [3].
  - Issue: When we try to test our JUnit file, it cannot find our file because of the wrong path. The solution is to use Repository root in the parameter.

```yaml
name: GitHub Actions for jd-gui newHistoryTest
on: [push]
jobs:
  Explore-GitHub-Actions:
    runs-on: ubuntu-latest
    steps:
      - name: Check out repository code
        uses: actions/checkout@v2
      - name: Test NewHistoryTest
        run: ./gradlew :app:test --tests org.jd.gui.model.history.NewHistoryTest --info
```

Figure6. Run “NewHistoryTest” command in the file

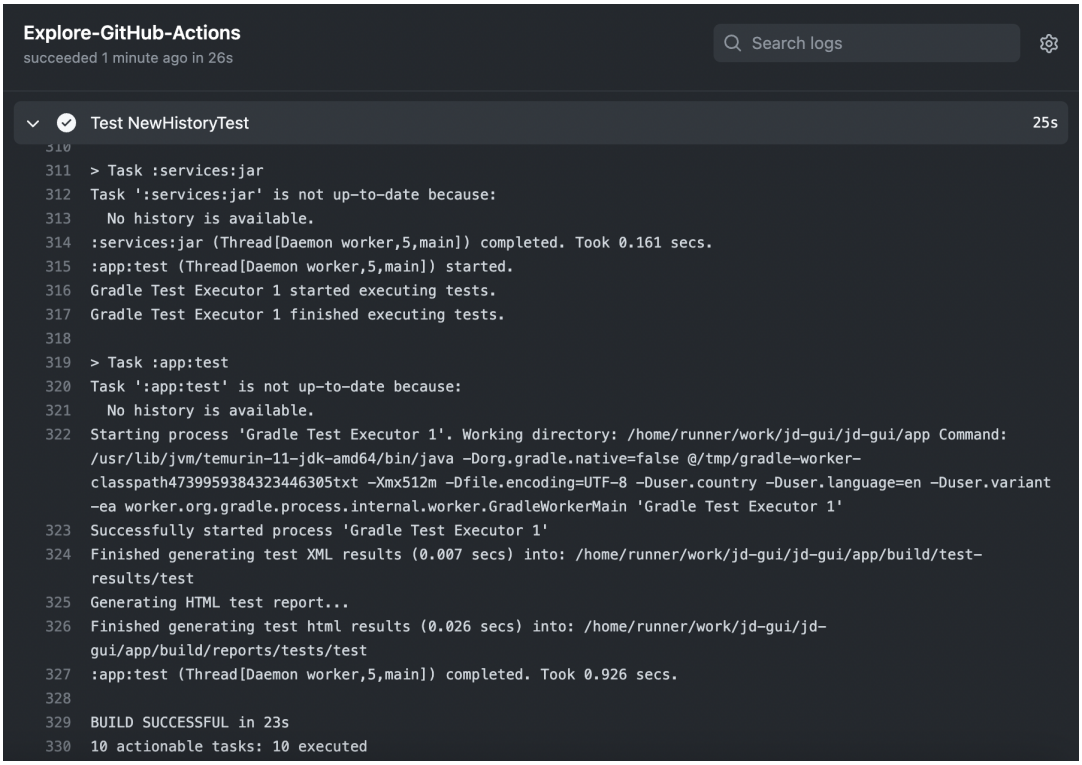


Figure7. Result of the command - successful test case

- Condition3: Run failed test case
  - To test a failed case, we run with “./gradlew :app:test --tests org.jd.gui.model.history.HistoryTest --info” and we add a line “Assert.assertEquals(0, 1);” in our code to make sure the case will be failed.

- Issue: In Gradle, it does not support “System.out.println()” on the console, which means when we try to print some log in our code, it cannot show on the Github actions workflow.

```yaml
name: GitHub Actions for jd-gui HistoryTest
on: [push]
jobs:
  Explore-GitHub-Actions:
    runs-on: ubuntu-latest
    steps:
      - name: Check out repository code
        uses: actions/checkout@v2
      - name: Test HistoryTest
        run: ./gradlew :app:test --tests org.jd.gui.model.history.HistoryTest --info
```

Figure8. Run “HistoryTest” command in the file

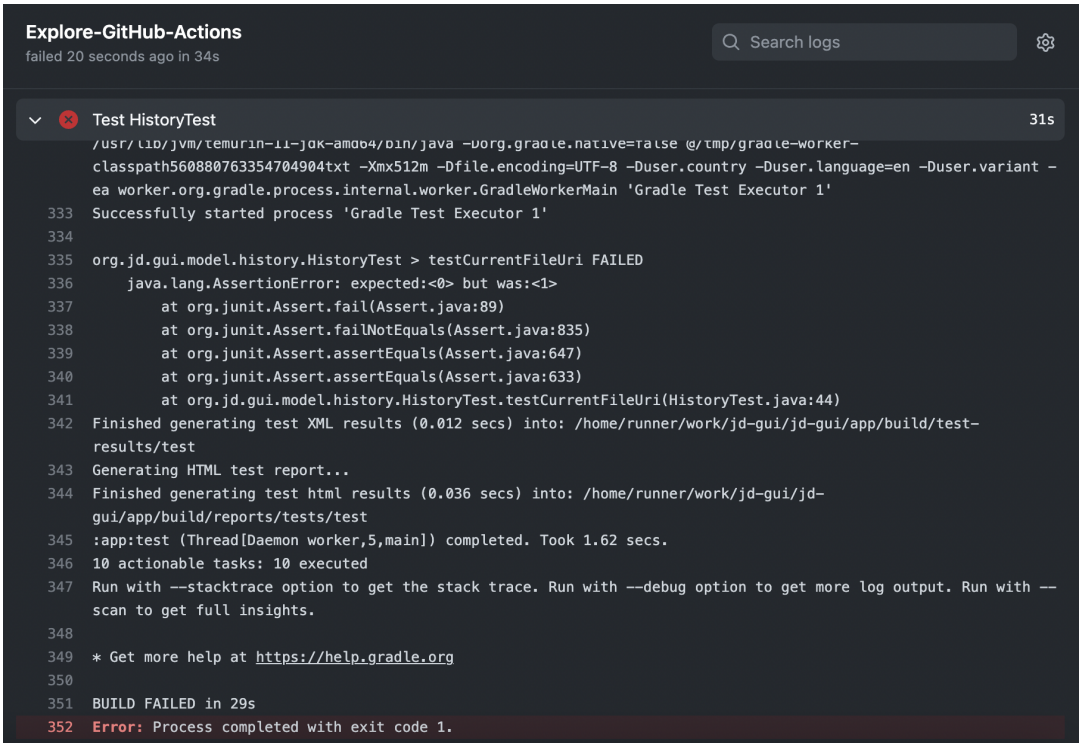


Figure9. Result of the command - failed test case

Conclusion

We use GitHub Action to do continuous integration and we test for three different conditions, such as build, successful test case and failed test case. During editing the files in workflows, we met some issues, but we successfully solved them. From this experience, we realized that continuous integration is important and that GitHub Action is a convenient tool. It automatically tests our project whenever we have committed.

Reference

1. *Features • GitHub Actions · GitHub*. (n.d.). GitHub. Retrieved February 21, 2022, from https://github.com/features/actions
2. *Quickstart for GitHub Actions*. (n.d.). GitHub Docs. Retrieved February 21, 2022, from https://docs.github.com/en/actions/quickstart
3. *How to run only single test cases with Gradle build*. (n.d.). Cloudhadoop. Retrieved February 21, 2022, from https://www.cloudhadoop.com/gradle-one-testcase-execution/

Chapter 5. Testable Design and Mocking

Introduction of testable design

The idea of testable design is that people can design a unit test for their codes easily. Hence, following some rules can help them write and develop the test cases better. For instance, try to avoid complex logic in private methods, avoid static methods, avoid logic in constructors, or avoid singleton patterns because all of these can increase the difficulty of testing the codes.

Example for not appropriate for the testable design

In the original code here, we see there is logic in the constructor. Besides initializing the constructor, it also has “if” and “else” to decide what value should be assigned to “lineSeparator”.

```
public class NewlineOutputStream extends FilterOutputStream {
    private static byte[] lineSeparator;

    public NewlineOutputStream(OutputStream os) {
        super(os);

        if (lineSeparator == null) {
            String s = System.getProperty("line.separator");

            if ((s == null) || (s.length() <= 0))
                s = "\n";

            lineSeparator = s.getBytes(Charset.forName("UTF-8"));
        }
    }
```

Figure1. Constructors with logic inside

New test cases

To avoid directly using the logic in constructors, we decide to move out the “if” and “else” parts to another method outside. Figure2 shows how we handle this and Figure3 shows how we rewrite new test cases.

```java
private static byte[] lineSeparator;

public NewlineOutputStream(OutputStream os) {
    super(os);

    initializeLineSeparator();
}
/** fixed code */
protected void initializeLineSeparator(){
    if (lineSeparator == null) {
        String s = System.getProperty("line.separator");

        if ((s == null) || (s.length() <= 0))
            s = "\n";

        lineSeparator = s.getBytes(Charset.forName("UTF-8"));
    }
}

protected byte[] getLineSeparator() { return lineSeparator; }
/** end */
```

Figure 2. New method to handle logic

```java
public class NewLineOutputStreamTest extends TestCase{
    public void testInitializeLineSeparator(){
        try {
            NewlineOutputStream newlineOutputStream
                    = new NewlineOutputStream(Files.newOutputStream(Paths.get(new File( pathname: "src/test/files/Four.cl
            Assert.assertEquals(newlineOutputStream.getLineSeparator()[0], (int)'\n');
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Figure 3. Test case for new testable code

Introduction to Mocking

Mocking is to focus on the code being tested without considering the behavior of external dependencies. It creates a fake object to unit test and sees whether it could pass or fail. One important usage for mocking is to do interaction testing. When using mocks, it can help test the interaction between the unit tests and methods quickly and test whether it reaches our expectations.

Feature selection for Mocking

History.java is one of several external dependencies of MainController.java. We can add a URI, and we also can call the methods, such as backward() and forward(), to get the current URI in the History.java, We use Mockito to create an object that mocks History.java. Then, we can test the behavior of History.java and the interaction between them and without affording the real History.java.

New test cases

First, we use Mockito to create an Object “spyHistory” to mock History, then add three URIs into “spyHistory” in the MockitoHistoryTest.java. Second, we use the when().thenReturn() method to stub forward() method.

After that, we use Assert to test the return value from “spyHistory”. Since we add three URIs, there should be one current URI and two URIs in the backward list. Thus, backward.size() should be 2 and forward.size() should be 0.

We also use System.out.println() to observe what is in the backward list. Shown in Figure 5. Although after calling twice for the backward() method, the size of the forward list should be 2, when we print out the URI return from the forward() method, it shows “/User/Test/forward.jar”. That’s because the forward() method is a stub method.

Finally, we pass the test case in MockitoHistoryTest.java, shown in Figure 6.

```
public class MockitoHistoryTest {
    @Test
    public void testVerify() throws URISyntaxException {
        // mock creation
        History history = new History();
        History spyHistory = spy(history);

        // mock object
        spyHistory.add(new URI("/User/Test/framework.jar"));
        spyHistory.add(new URI("/User/Test/interface.jar"));
        spyHistory.add(new URI("/User/Test/method.jar"));

        // stubbing
        when(spyHistory.forward()).thenReturn(new
URI("/User/Test/forward.jar"));

        Assert.assertEquals(2, spyHistory.backward.size());
        Assert.assertEquals(0, spyHistory.forward.size());

        // print real backward() element
        System.out.println(spyHistory.backward());
        System.out.println(spyHistory.backward());

        Assert.assertEquals(2, spyHistory.forward.size());

        // print stubbed method
        System.out.println(spyHistory.forward());

        // verification
        verify(spyHistory).add(new URI("/User/Test/framework.jar"));
    }
}
```

Figure 4. Test case for mocking History.java

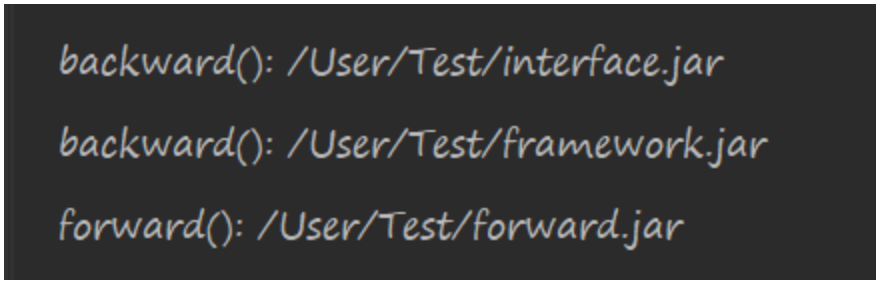


Figure 5. Output for MockitoHistoryTest.java

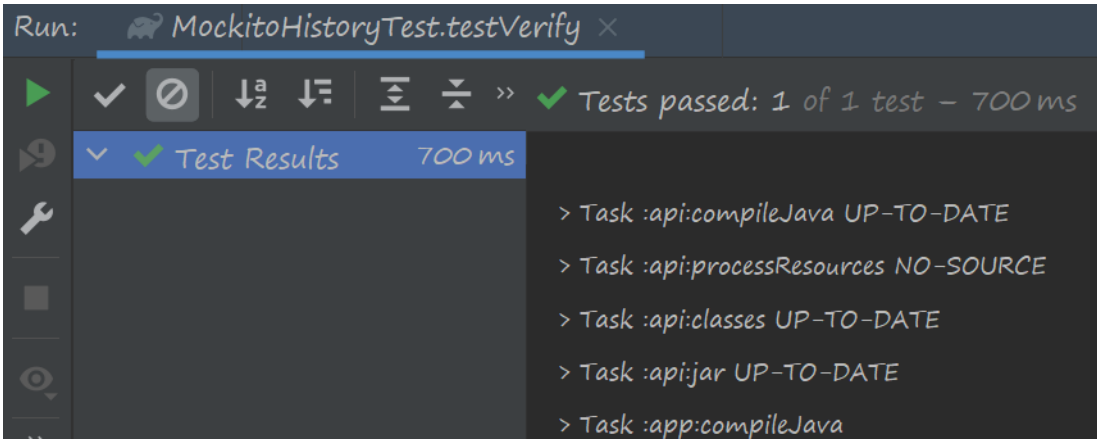


Figure 6. Test Results for MockitoHistoryTest.java

Chapter 6. Static Analyzers

Introduction to Static Analyzer

Static analyzers help developers find the bugs easier. Sometimes even the codes can be compiled and run successfully, there are some invisible bugs inside. Using static analysis tools, which define the different code rules, can help find those bugs before running and give some suggestions on how to fix them, which is very useful for developers and saves time for debugging.

Static Analysis

The static analyzer tools we choose are mainly CheckStyle and PMD. FindBugs is for reference. The file we choose to analyze is History.java.

1. Checkstyle

Checkstyle can check code layout and formatting issues. It can also find class design and method design problems. Figure 1 is the results of using the Sun Checks rule in Checkstyle to analyze Hisory.java.

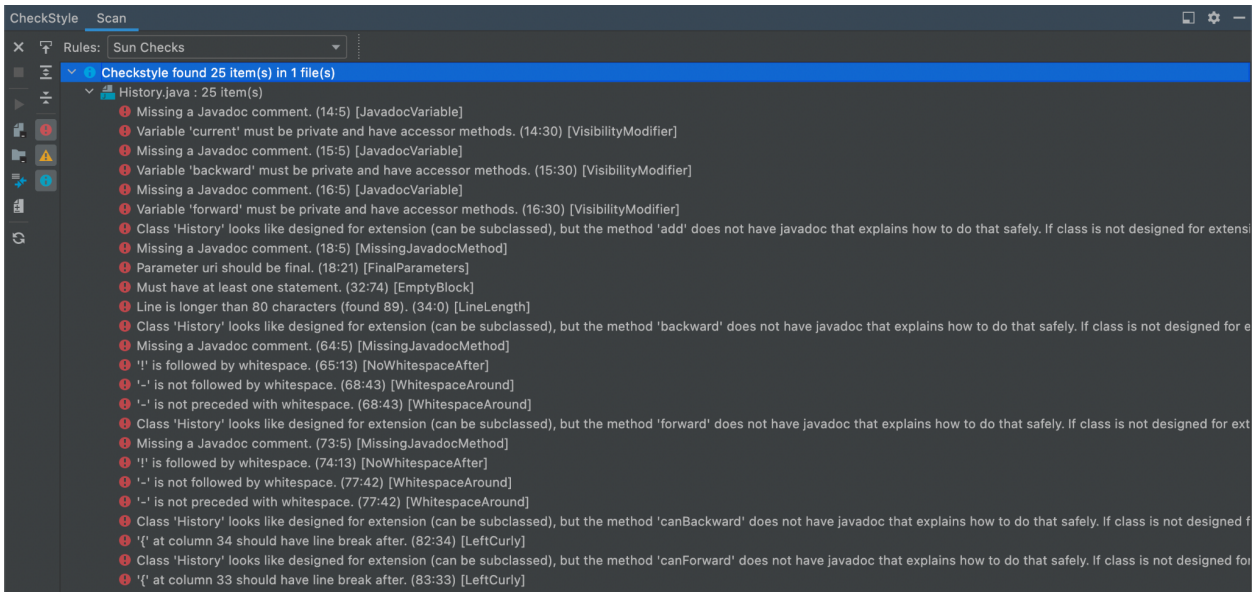


Figure 1. Checkstyle results

In the Checkstyle report, there are 25 issues found and can be divided into 5 categories.

- Javadoc Comments - the variables and methods are missing the javadoc comments to explain their functionality.
- Visibility - the variables in the class should be private and have their own accessor methods.

- Final Parameters - the parameter in the method should be final since it will not be reassigned.
- Empty Block - in conditional statements, there should be at least one statement in the block.
- Formatting - the line length should not be too long, there are some additional whitespaces or missing whitespaces, and the left curly should have a line break after it.

For “Javadoc Comments”, “Empty Block”, and “Formatting” issues, they are not actual problems to the code since they won’t affect the result of the code. But for some cases, “Javadoc Comments” can be important to make others use the code correctly. Moreover, with a good format can make the code cleaner and have a better expression. Then, for “Visibility” and “Final Parameters”, they seem to be more important because they will change the usage of some codes.

2. PMD

PMD is a code analyzer. We use the PMD plugin in IntelliJ to analyze our History.java automatically. After running PMD, we get the analyzed report, as Figure 2. We classify those warnings into several types of warnings.

After classifying, we can find that some of the warnings are related to code structure. These warnings are more important because they can prevent the code from unexpected errors, especially for a project with large code. For example, "a class should have at least one constructor” and “Potential violation of Law of Demeter (method chain calls)”; some of the warnings are about code readability, such as, “Class comments are required” and “Useless parentheses”.

We list our classification below:

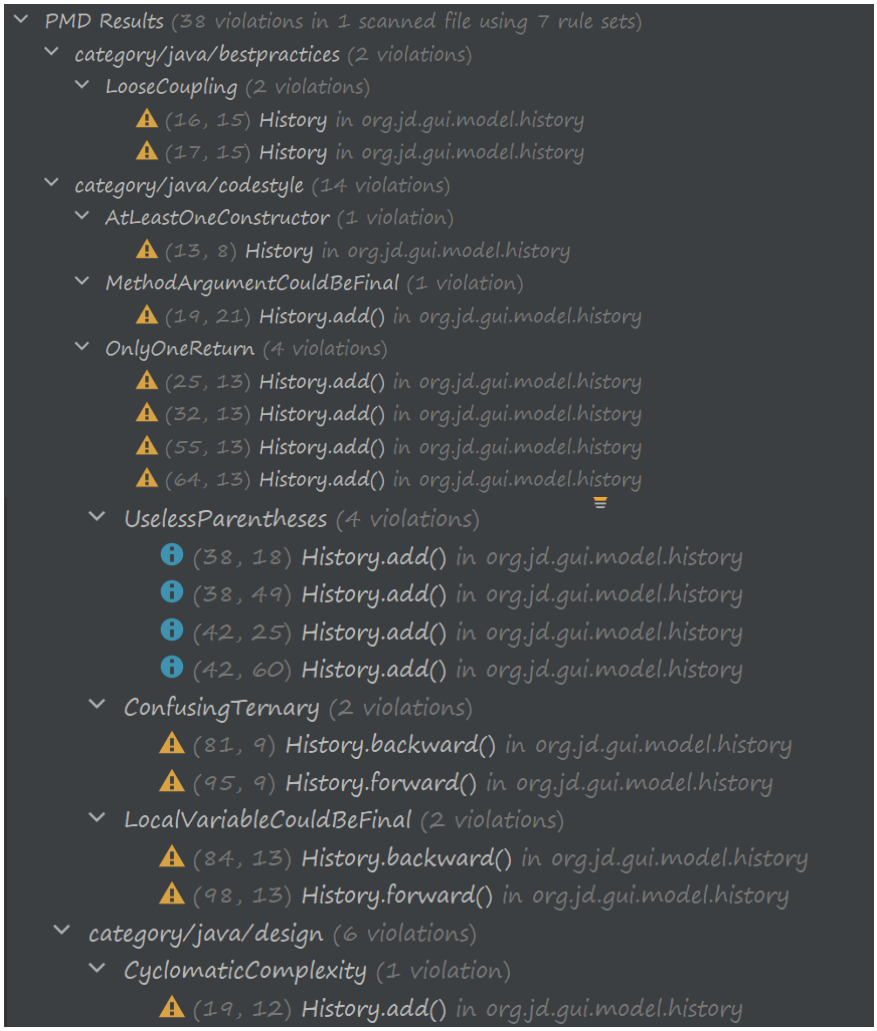
Important:

- Requirements of OOP code structure
    - Each class should declare at least one constructor
    - Avoid using implementation types like 'ArrayList'; use the interface instead
    - Found non-transient, non-static members. Please mark as transient or provide accessors.
- Code style
    - A method should have only one exit point, and that should be the last statement in the method
    - Avoid empty if statements
    - Avoid if (x != y) ..; else ..;
    - Potential violation of Law of Demeter (method chain calls)
- Name
    - Field ‘backward’ has the same name as a method

    - Field ‘forward’ has the same name as a method
- Performance
    - The method 'add(URI)' has a cyclomatic complexity of 10.

Less important:

- Lack of comments
    - Class comments are required
    - Field comments are required
    - Public method and constructor comments are required
- Variable type
    - Parameter 'uri' is not assigned and could be declared final
    - Local variable 'size' could be declared final
- Redundant
    - Avoid using redundant field initializer for 'current'
    - Useless parentheses



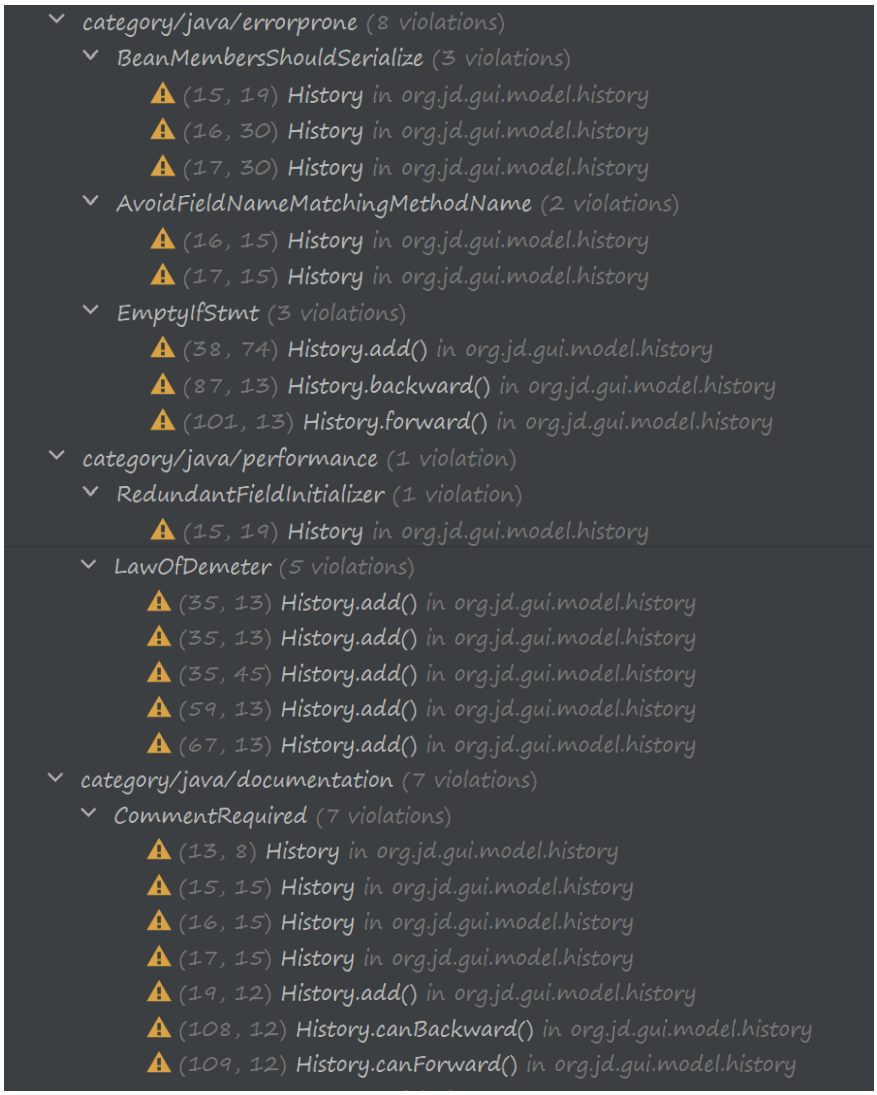


Figure 2. History.java result for PMD tool

3. FindBugs

In FindBugs, it does not find the bugs in History.java as Figure 3. However, when we use FindBugs to analyze the whole project as Figure 4, it finds the bugs on some other files. We can see from the results that FindBugs checks the code with higher levels. Unlike the other two tools, it does not really care about the specific usage of code, but cares whether it is more readable or whether it can be improved for performance's concern.

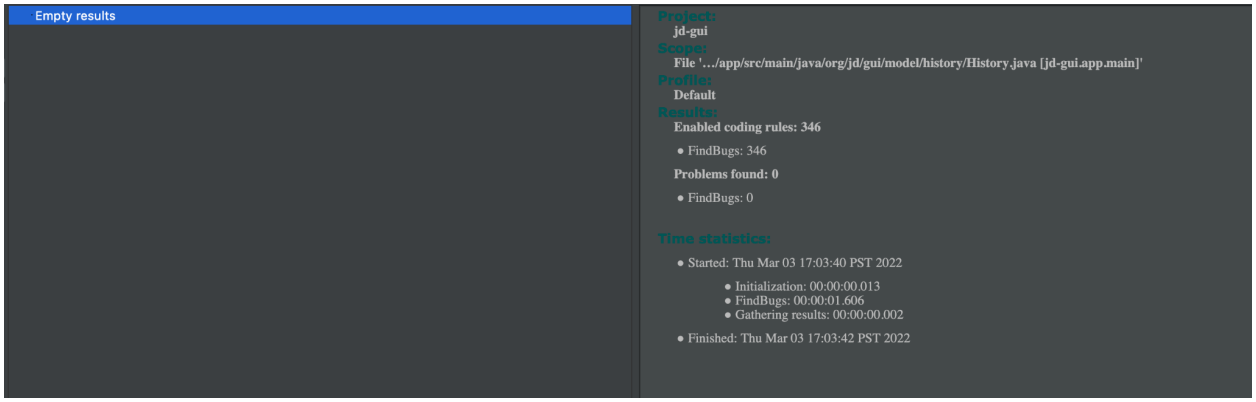


Figure 3. History.java using Findbugs

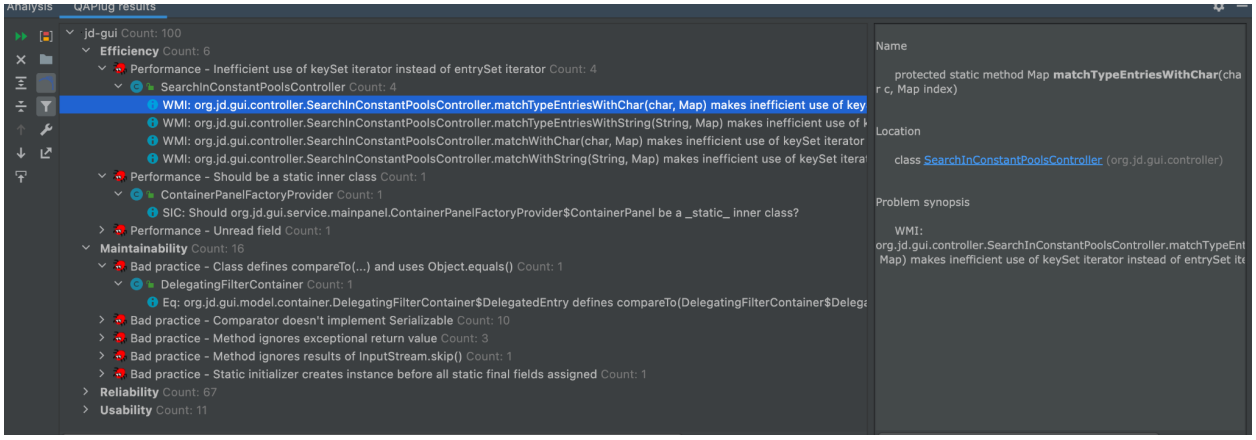


Figure 4. All projects using FindBugs

Comparison

From the results of CheckStyle and PMD, we can see the similarities and differences below.

Similarity:

1. Miss the comments: These two tools all suggest that we should include the comments for each function in codes. This is very important to maintain the codes, because with the comments, we can know what the codes mean in each section.
2. Final Parameters: These two tools both suggest that those parameters will not be reassigned in the method should be final type.
3. Empty Block: These two tools all suggest there should be at least one statement in the block.

Difference:

1. Visibility of variables: In CheckStyle, it suggests that variables in the class should be private and have their own accessor methods. However, this bug does not show in PMD. We think it is reasonable because it is a safer way to access the variables.
2. Redundant: In PMD, it suggests avoiding using the redundant field for initialization, but in CheckStyle we do not see this. We think it is because we are using Java. While using C or C++, we should initialize the parameter. For this suggestion, we agree it is okay not to initialize the current parameter but it is not that important.
3. Formatting: In CheckStyle, it has many suggestions about code formatting, such as line length, whitespaces or line break. But in PMD, it doesn't have these kinds of suggestions. It seems CheckStyle focuses more on the code formatting than PMD.

Reference

1. *checkstyle – Checkstyle 10.0*, https://checkstyle.sourceforge.io/