**Test Coverage**

In this part, we will look at the process of our softwate testing. The main purpose of the test is to get the maximum branch coverage possible, and in order to do this we have to make sure that all the branches in the conditional statement are working correctly.

**● Test requirement**

The goal is to set a maximum of 100% and achieve the maximum branch coverage as possible.

**● Coverage metrics**

1. Use Junit test to achieve the maximum branch coverage.

|  |
| --- |
| package test;  import org.junit.Test;  import static org.junit.Assert.\*;  import edu.handong.se.markdownconverter.\*;  public class MDParserTest  {  @Test  public void textTest1() {  Text t = new Text();  String testText = "Hello";  t.setValue(testText);  String getTestText = t.getValue();  assertEquals(testText, getTestText);  }  } |

To write tests for Junit, we need to use the the methods of junit.assert. So we imported junit.Assert and junit.Test. And, as in the example code above, we used the assertEquals method to check whether the get method works well.

2. Use an Ant build script to unit-test project

|  |
| --- |
| <project name="2017-SE-Team-8" default="compile" basedir="." xmlns:jacoco="antlib:org.jacoco.ant">  <target name="test" depends ="build">  <junit showoutput="true" printsummary="on" enabletestlistenerevents="true" fork="true">  <classpath path="${build.dir}" />  <classpath path="lib/junit.jar" />  <classpath path="lib/hamcrest-core.jar" />  <formatter type="plain" usefile="false" />  <test name="test.MDParserTest">  </test>  </junit>  </target> |

Ant is a Java base tool for automating the build process. We made an Ant file ‘build.xml’ and it tasks compile, build, and so on. In the above code, we set the target name to test and specify the classpath and the file to be executed. Through this, if the user enters ‘ant test’ at the terminal, the program will proceed the test.

3. Use Jacoco to measure the branch coverage of the Junit test

|  |
| --- |
| <target name="cov-test" depends ="build">  <jacoco:coverage>  <junit showoutput="true" printsummary="on" enabletestlistenerevents="true" fork="true">  <classpath path="${build.dir}" />  <classpath path="lib/junit.jar" />  <classpath path="lib/hamcrest-core.jar" />  <formatter type="plain" usefile="false" />  <test name="test.MDParserTest">  </test>  </junit>  </jacoco:coverage>  </target> |

We use Jacoco to measure the branch coverage. It reports the coverage measurement in HTML file. We can add code coverage to these tasks by simply wrap the codes like an above examples

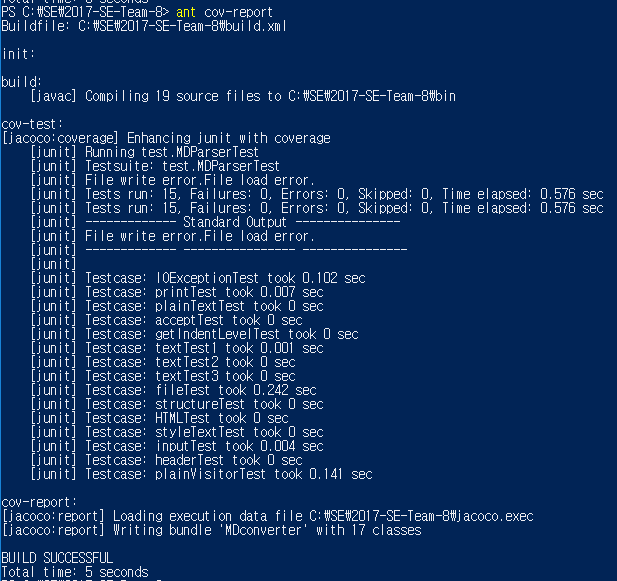


Figure 1. Execute JaCoCO test and get report.

**● Coverage**

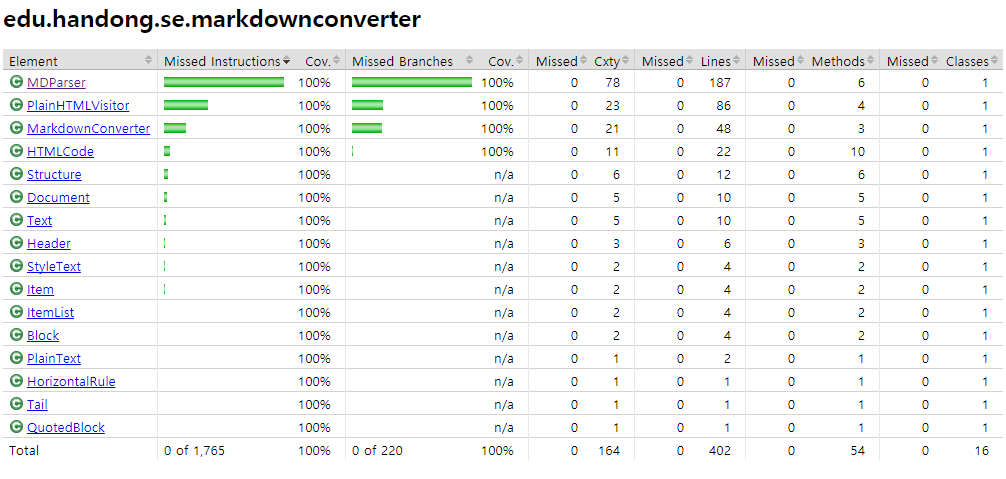


Figure 2. JaCoCo code coverage report

As we can see, the total number of branches in our source code is 220. Most of them are MarkdownConverter, which is the main class of code, MDParser, which provides parsing, and PlainHTMLVisitor, which acts as a visitor.

**● Coverage criterion**

We can ignore false branches of assertion statements.

**● Example test cases**

1. An user input a wrong command.

John wants to convert the MD file to HTML. However, when he first ran the program, he did not know how to use the program, so he entered the wrong command. In this case, the program determines whether the command entered by the user is a command for proper operation, and outputs a help message if it is not.

1. Wrong file name input

Jane created the MD file to be converted and entered the filename into the program. However, she made the mistake of typing the name of a file that did not exist, unlike the MD file she created. In this case, the program checks whether a file matching the input file name exists in the path, and if there is no file, it catches IOException and prints an error message.

1. Case that works normally

Mary entered the MD file normally into the program, and the program added the link and heading from the existing file contents to the new HTML file according to the procedure. And eventually she got the converted HTML file successfully.

**● Issues that occurred during testing**

Most of the problems that were difficult to test were things that happened when you were testing assuming that the user gave the wrong input. Especially, it seems to be more difficult because there are so many unpredictable possibilities. So, when we made the code, we took some defensive code to prevent the wrong input, which was rather difficult part of the test. For example, if an exception occurs, a conditional statement such as if causes the program to stop and exit. However, if a program has already exited the function by these conditions at the beginning of the code, then no such exception occurred in the middle of the function. It took a lot of time to find these misses.

In addition, in the case of conditional statements, there were many conditional statements with more than two conditions. In these cases, if 'or' is used in the condition, the first condition is satisfied, and the following condition does not occur, so branch coverage is missed. To clear all of these misses, the number of inputs that we have to enter has varied, and this has also been a very complicated problem.

And at the beginning of the test, there was a difficulty with inputting null values or entering nonexistent files. in terminal I was able to throw an exception through a method that gave no value, but I was not very good at it because I did not know how to make this case when I was writing test code. However, in the end, I was able to solve the problem by finding a way to create a new empty string array through distress and search.