JSoup, a tool you need to parse HTML

Part I: Introduction, Set Up and Partitioning

Introduction to JSoup

`JSoup` is a very useful Java library for working with real-world HTML/XML. During the real world process, we usually meet with HTML or XML files. In Python, we mostly use `beautifulSoup` to handle these messy and dummy HTML files into neat and clear data structures. In Java, we use `JSoup`, converting HTML or XML into neat and clear data structures by its internal `parser` function. Its speed is much faster than Python and one of your best choice for Java projects.

Our Github link is https://github.com/duke326/SWE261. Our collaborators are Lai Wang, Xinyi Hu and Yu Sun. We have already added TAs and Professor Jones into our project.

Useful Functions of JSoup

It uses the best DOM method of HTML5 and CSS selector to provide a very convenient API for obtaining URLs and extracting and processing data.

jsoup implements the WHATWG HTML5 specification, and parses HTML to the same DOM as modern browsers.

- scrape and parse HTML from a URL, file, or string
- find and extract data, using DOM traversal or CSS selectors
- manipulate the HTML elements, attributes, and text
- clean user-submitted content against a safe-list, to prevent XSS attacks
- output tidy HTML

jsoup is designed to handle all kinds of HTML/XML found; from raw and verified to invalid tag soup; jsoup will create a wise parse tree.

Detailed analysis into JSoup

To dive deep into some data of `JSoup`, we have counted the lines of code for `JSoup` and it appears to be 32887.

In ubuntu system, I used this command to calculate the number of code except the comments.

```
find . -type f -name '*.java' | xargs cat | wc -l
```

Also, we have counted number of java files, and the result is 132.

In ubuntu system, I used this command to calculate the number of java files.

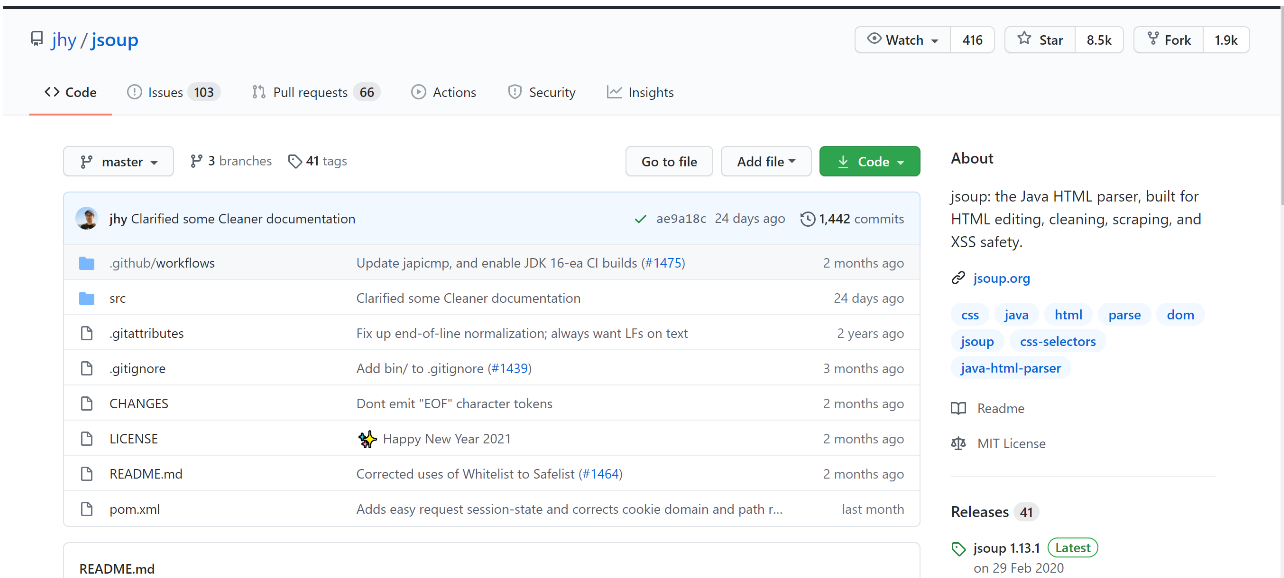
```
find . -name '*.java' | wc -l
```

The language of `JSoup` is 100% of Java.

Set Up environment for JSoup

Fork & Add

Firstly, we need to fork this project into our own repository. Using GitHub, the original github link is https://github.com/jhy/jsoup/. To fork, we press the button in the right upper corner of this page, and put it into our own account. One of our team member, Yu Sun forked this repository, and add other team members into collaborators.



To update the code in this repository, normally we discuss together, so that we push the changes directly into the master, or fork the repository into our own branch, and pull request the changes into the master.

Our link is here: https://github.com/duke326/SWE261. We have all of our team members, Lai Wang, Sun Yu and Xinyi Hu in the repo. Also we have included Prof. Jones and TA Maruf.

Build

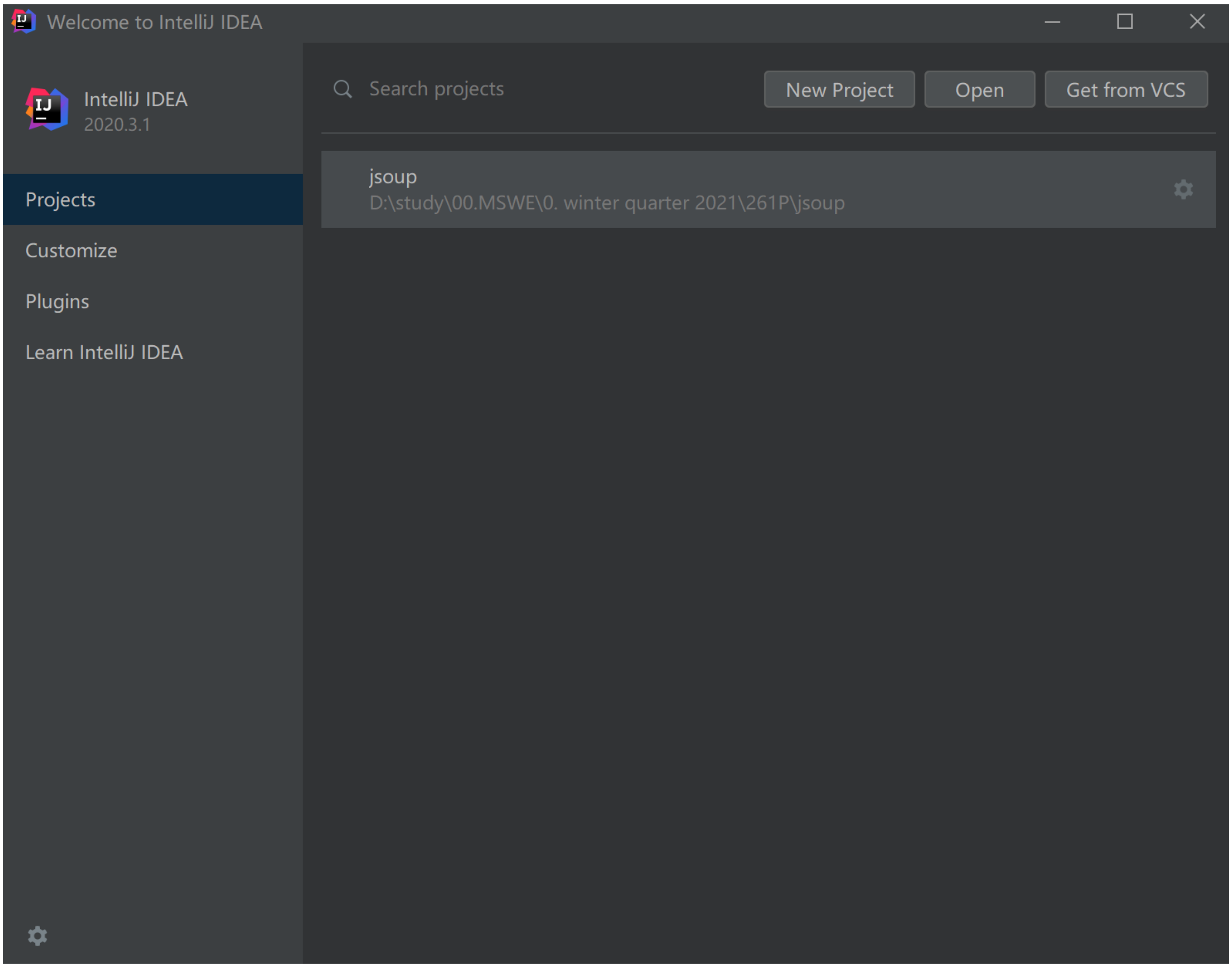
After the fork, next step, we want to know how to build our project. All of our team members use IntelliJ IDEA provided by JetBrain so that we use IntelliJ IDEA as our main development IDE.

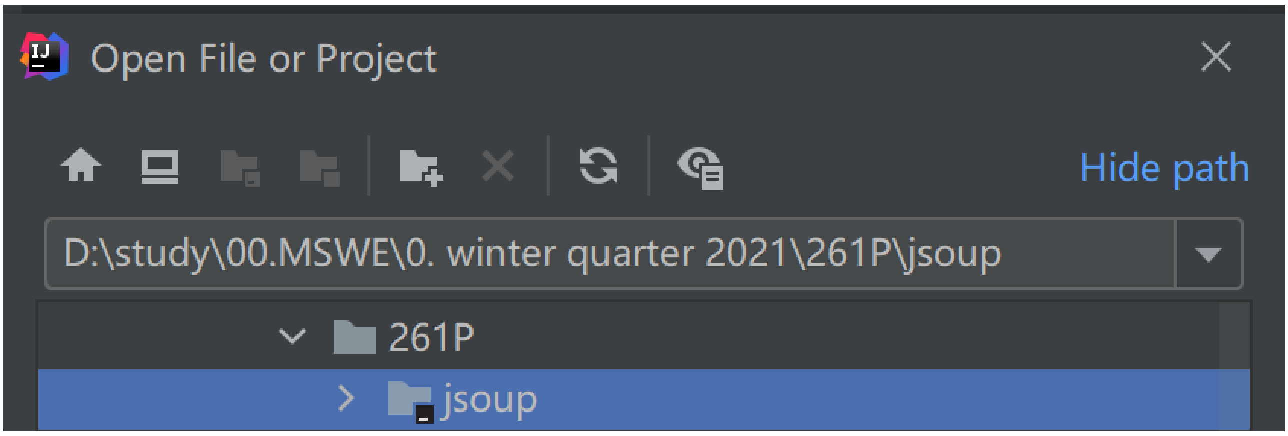
Using IntelliJ IDEA, this project uses Maven as the build infrastructure for Java projects.

Maven uses rules and patterns to provide an integrated build system. Each Maven project uses a set of shared plugins retrieved from the Maven repository, and when the project is built, Maven performs a set of predefined tasks throughout its life cycle.

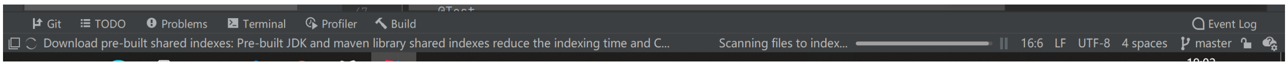
Process of using IntellJ IDEA to open Maven project

(Sample Environment: Windows 10)





Then your idea will automatically download the dependency.



How to run test cases?

When the progress bar is full, now you can run your program! Including any test case in the test folder!

Setup for using JUnit 5

To use Maven you have to use updated version for your build plug-ins and add several dependencies.

To use JUnit 5 with Maven, we need to change `pom.xml`. In the `pom.xml`, you nd to add its plugin as `maven-surefire-plugin`, and its version is `2.22.2`. To see how to write that, please refer to our code in the line 221-241.

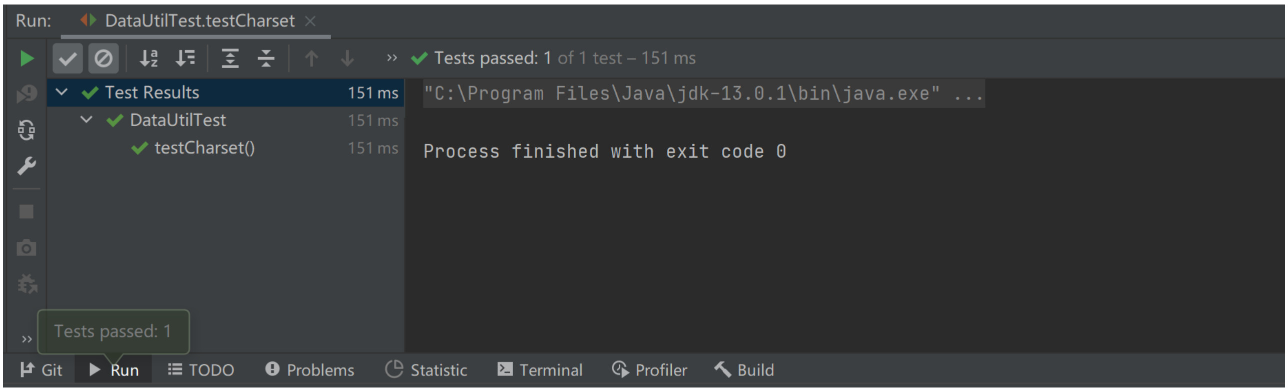
Sample to run

For example, in `test/java/ org.json/helper/DataUtilTest.java`

```java
public class DataUtilTest {
    @Test
    public void testCharset() {
        assertEquals( expected: "utf-8", DataUtil.getCharsetFromContentType("text/html;charset=utf-8 "));
        assertEquals( expected: "UTF-8", DataUtil.getCharsetFromContentType("text/html; charset=UTF-8"));
        assertEquals( expected: "ISO-8859-1", DataUtil.getCharsetFromContentType("text/html; charset=ISO-8859-1"));
        assertNull(DataUtil.getCharsetFromContentType("text/html"));
        assertNull(DataUtil.getCharsetFromContentType(null));
        assertNull(DataUtil.getCharsetFromContentType("text/html;charset=Unknown"));
    }

    @Test
```

Press the little green run button, the output would be like this:



Existing Test cases

In the first place, the existing test case we found is **black box testing**, which means that we ignore the internal functions, just check whether the output provided by function are the same as our desired output. In IEEE, black box testing means that one kind of testing that ignores the internal mechanism of a system or component and focuses solely on the outputs generated in response to selected inputs and execution conditions.

- In one word, it is a testing method that has no view of code.

Usually, using black box testing, it try to select inputs that are especially valuable so that it can test the functions equally. The reason for that is simple and clear.

- Using non-uniform method, also called **systematic testing** can deal with the sparse problem in the input space. A classic example is Java class "root" mentioned in the class.
- Using this kind of systematic testing can find bugs and remove them from hay more effectively.

As evolved in our class, the existing test case also included **continuous integration**. As we will talk about in report part IV, continuous integration means the practice of merging the working copies of all developers into the shared mainline several times a day. This kind of software testing technique is proposed by Grady Booch. It has four important components.

- Firstly, continuous integration is originated from extreme programming development process. Due to high amount of development, there would be lots of changes of code and different progress align with different colleagues. It is super important under such extreme programming environment
- Secondly, continuous integration needs to be performed, even for minor changes. Therefore, every developer are in the same pace for the project.
- Thirdly, every developer in the group and in the project needs to commit their changes every day. Therefore, every developer are in the same pace for the project.
- Fourthly, every version, especially the latest version, needs to build and pass all the tests. Otherwise, this program cannot work, and need to check its methods and functions.

In our project, it used **JUnit**. *JUnit 5* is the next generation of JUnit. The goal is to create an up-to-date foundation for developer-side testing on the JVM. This includes focusing on Java 8 and above, as well as enabling many different styles of testing.

How to run test cases? JUnit!

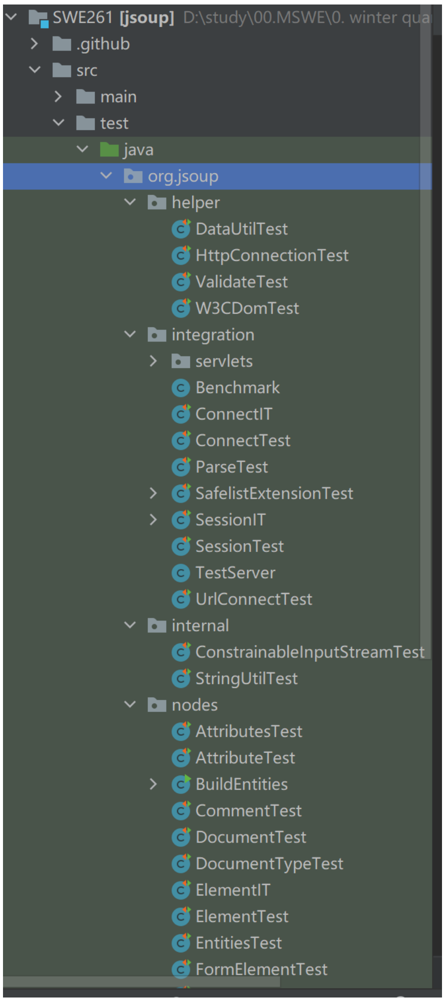
To run the test cases, we need to add JUnit into our project! As we talked about in the build section, we have already put JUnit in our pom.xml file, so that you can easily run JUnit in the IntellJ IDEA without worrying about installing it.

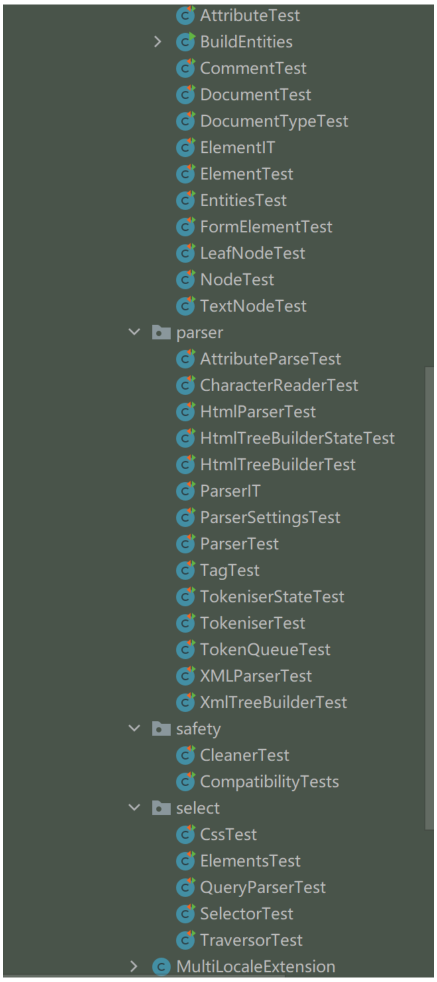
JUnit *test* is a method contained in a class that is only used for testing. This is called *test category*. To define a specific method as a test method, we usually annotate the method with `@Test`.

This method executes the code under test. Use the *assert* method provided by JUnit or other assert frameworks to compare the expected result with the actual result. These calls are usually called *assert* or *assert statement*.

The assert statement can usually define the message that will be displayed if the test fails. You should provide meaningful messages here to help users identify and resolve issues. This is especially true if the code under test or someone who did not write the test code sees the problem.

Existing test cases





Sample test case

In `org/jsoup/parser/parsesSimpleDocument.java`

```
    @Test
    public void parsesSimpleDocument() {
        String html = "<html><head><title>First!</title></head>
<body><p>First post! <img src=\"foo.png\" /></p></body></html>";
        Document doc = Jsoup.parse(html);
        // need a better way to verify these:
        Element p = doc.body().child(0);
        assertEquals("p", p.tagName());
        Element img = p.child(0);
        assertEquals("foo.png", img.attr("src"));
        assertEquals("img", img.tagName());
    }
```

This method mainly test the parser. This function parse HTML(String) to the **Document** object. **Document** object contains attributes that can be used to analysis the **Elements** in the HTML(String). This `parsesSimpleDocument.java` test method mainly test whether the result from parse method is actually match the real data using `assertEquals`. In other word, to test the validation of function parse.

In `org/jsoup/nodes/testTitles.java`

```
    @Test
    public void testTitles() {
        Document noTitle = Jsoup.parse("<p>Hello</p>");
        Document withTitle = Jsoup.parse("<title>First</title>
<title>Ignore</title><p>Hello</p>");

        assertEquals("", noTitle.title());
        noTitle.title("Hello");
        assertEquals("Hello", noTitle.title());
        assertEquals("Hello",
noTitle.select("title").first().text());

        assertEquals("First", withTitle.title());
        withTitle.title("Hello");
        assertEquals("Hello", withTitle.title());
        assertEquals("Hello",
withTitle.select("title").first().text());

        Document normaliseTitle = Jsoup.parse("<title>
Hello\nthere   \n   now   \n");
        assertEquals("Hello there now", normaliseTitle.title());
    }
```

This method mainly test the validation of **Document** object and its attribute. There are several assertion test in this test unit, including `noTitle` and `withTitle`. They tested the **title** attributes and validate the accuracy of the method parse.

In `org/jsoup/parser/TagTest.java`

```
  @Test
    public void isCaseSensitive() {
        Tag p1 = Tag.valueOf("P");
        Tag p2 = Tag.valueOf("p");
        assertNotEquals(p1, p2);
    }

  @MultiLocaleTest
    public void canBeInsensitive(Locale locale) {
```

```
        Locale.setDefault(locale);

        Tag script1 = Tag.valueOf("script",
ParseSettings.htmlDefault);
        Tag script2 = Tag.valueOf("SCRIPT",
ParseSettings.htmlDefault);
        assertSame(script1, script2);
    }
@Test
    public void equality() {
        Tag p1 = Tag.valueOf("p");
        Tag p2 = Tag.valueOf("p");
        assertEquals(p1, p2);
        assertSame(p1, p2);
    }
```

This Test class contains many unit test to validate the **Tag** and test the functionality of tag attributes in the **Document** object. The above code mainly test the case sensitive, equal and the Insensitive of different **Tag**.

Partitioning

Introduction to systematic functional testing and partition testing

Systematic functional testing is usually used in functional testing, also called black box testing method. It is a non-uniform method that try to select inputs that are especially valuable. Usually, systematic functional testing choose representatives of classes that are apt to fail often or not at all.

Partition testing is similar to the concept of partition in data structure - divide and conquer. In debug and testing field, partition testing means that the input space is divided into classes that merge into the entire space. These classes might overlap.

It is also similar to Heine-Borel theorem in mathematics in my perspective, which is to cover all of the cases using pieces of subcases. That is the second statement of this theorem: S is compact, that is, every open cover of S has a finite subcover. These subcover might overlap too. Back in my bachelors' study, my professor Pang, Xuechen gave an example - In the fall, leaves fall and they cover the ground from your dorm to campus. Leaves are finite and they do cover all the road. That's Heine-Borel theorem.

Importance of these testing methods

As mentioned in existing test cases in set up environment part, we described the importance and advantages of systematic function testing. Usually, it try to select inputs that are especially valuable so that it can test the functions equally. The reason for that is simple and clear.

- Using non-uniform method, it can deal with the sparse problem in the input space. A classic example is Java class "root" mentioned in the class.
- Using this kind of systematic testing can find bugs and remove them from hay more effectively.

For partition testing, the input space is divided into classes that merge into the entire space. It also has advantages and great importance.

- It can sample each class. And in one quasi-partition, if there is bug in one function, at least one class will reveal the fault and it can clearly lead to where the bug is. The reason why there is at least one class can will reveal the fault is that classes might overlap, and more than one class tests with the same function.
- Each fault is in dense space in some class of inputs.

The step of our testing is divided into four steps.

1. Decompose the specification into equivalence partitions
2. Select representatives
3. Form test specifications
4. Produce and execute actual tests

JSoup Partitioning Test case

In `JSoup` project, we have XML parser and HTML parser, but in test cases, `JSoup` only has HTML parser testing method. Therefore, we decided to choose XML parser and test the functions in XML parser.

Our partitions and boundaries

In this project, we select XML parse as our partition feature.

In test case, we try to figure out whether `Jsoup` can process the correct xml string as well as incorrect xml string using `assertEquals` function.

For boundaries, we talked about Leap Year in class, and its boundaries are range of years and certain set of years, like 2000. For our case, we don't have ranges of xml. Then we figure out that we can use the **length** of each xml file as ranges. Also, we try boundaries case like xml with large tag name case.

Write new test cases in JUnit

Test whether `Jsoup` could get elements from the tag we assign:

```
@Test
public void parsesDocumentSize() {
    Document doc = Jsoup.parse(xml);
    Elements name = doc.getElementsByTag( "name" );
    assertEquals(2, name.size());
}
```

Test whether `Jsoup` could get element from the elements array:

```
@Test
public void parsesSimpleDocumentElement() {
    Document doc = Jsoup.parse(xml);
    Elements name = doc.getElementsByTag( "name" );
    Element element = name.get( 0 );
    String text = element.text();
    assertEquals("tom", text);
}
```

Test whether `Jsoup` could return correct answer when we input a wrong tag:

```
    @Test
    public void parsesNullExistDocumentElement() {
        Document doc = Jsoup.parse(xml);
        Elements id = doc.getElementsByTag( "id" );
        assertEquals(0, id.size());
    }
```

Test whether `Jsoup` could get xml's attribution:

```
@Test
public void parsesSimpleDocumentElementAttr() {
    Document doc = Jsoup.parse(xml);
    Elements name=doc.getElementsByTag("student");
    Element element = name.get( 0 );
    //System.out.println(element.text());
    String studentNumber=element.attr("number");
    //System.out.println(studentNumber);
    assertEquals("0001", studentNumber);
}
```

Test whether a very large tag name can be parse by `Jsoup` :

```
@Test
public void handleSuperLargeTagNames() {
    // unlikely, but valid. so who knows.

    StringBuilder sb = new StringBuilder(maxBufferLen);
    do {
        sb.append("LargeTagName");
    } while (sb.length() < maxBufferLen);
    String tag = sb.toString();
    String xml = "<" + tag + ">One</" + tag + ">";
    Document doc =
Parser.xmlParser().settings(ParseSettings.preserveCase).parseInput(
xml, "");
    Elements els = doc.select(tag);
    assertEquals(1, els.size());
    Element el = els.first();
    assertNotNull(el);
    assertEquals("One", el.text());
    assertEquals(tag, el.tagName());
```

```
}
```

Part II: Functional Models and Finite State Machines

Introduction to Finite State Machines

What is finite state machine? According to wikipedia, the finite state machine (or FSM) can be constructed before the source code or independently of the source code. A finite state machine (or FSM) can be used as a specification for allowed behavior. It has five important components.

- A finite state machine is a set of states and a set of transitions.
- A finite state machine is a directed graph.
- A finite state machine is a node that represents the state of a program.
- Edge represents the operation of transforming one program state into another program state. Usually marked with program operations, conditions or events.
- Due to countless states, FSM must be abstract.

The reason why finite models are useful for testing

Using finite models, we can draw a state transition tables. These transition tables can help us check the completeness of the program. These completeness can help us do the followings conditions.

1. Help analyze the original state of program.
2. Help analyze the complete process of program.
3. Help test potential bugs.
4. Testing might pass all the branches
5. After analyzing the branches, we can test in more detailed and more targeted way.
6. When encountered with bug, we can target at which branch has the bugs.

Choose a feature

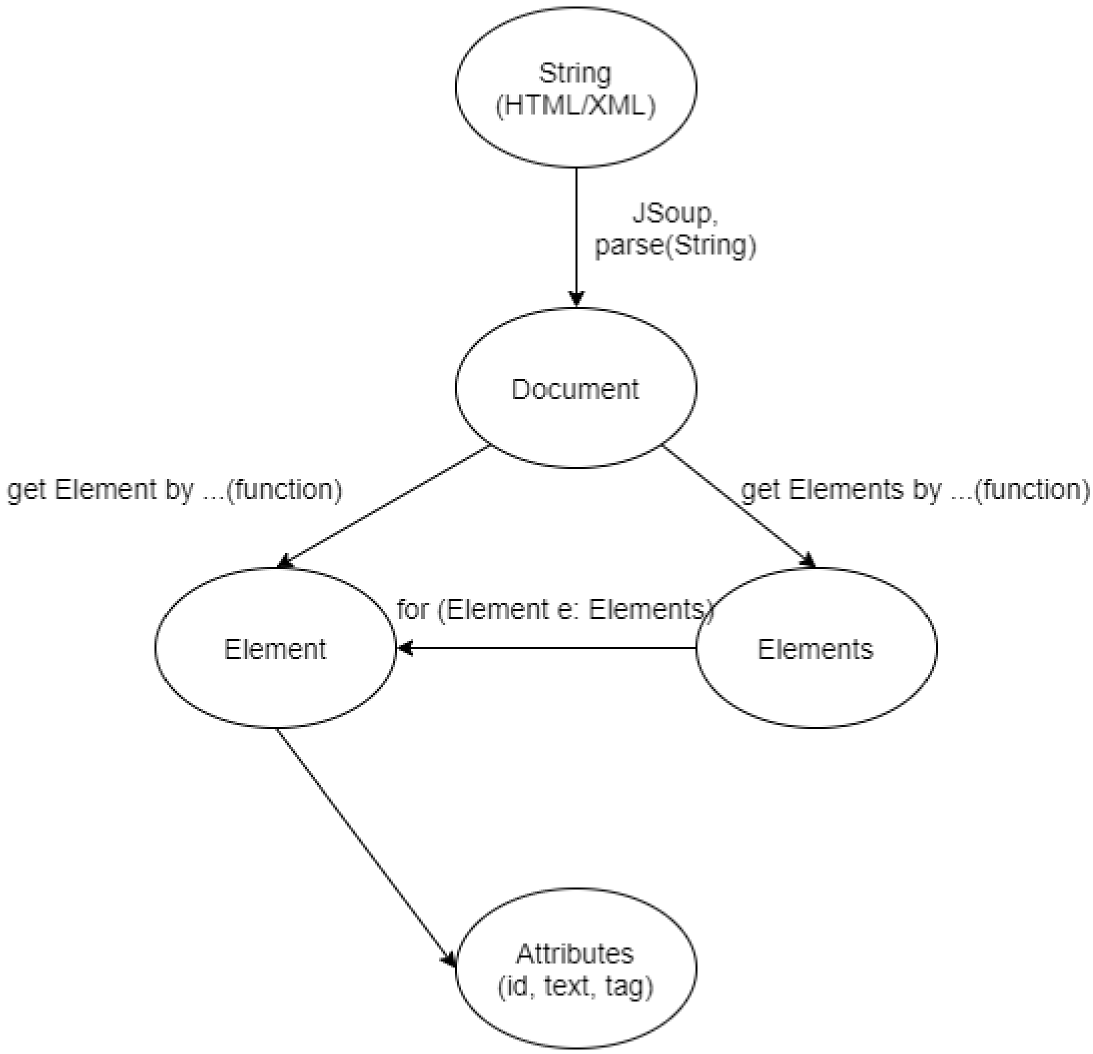
In our project `JSoup`, it is a Java library for processing actual HTML/XML. In the functional model, we extracted two features from `JSoup`.

- scrape and parse HTML/XML
- manipulate the HTML/XML elements, attributes.

Create, draw, and describe that functional model, how it works

In `JSoup` progress, you can see the process through the picture below.

1. Firstly, `parse` HTML or XML Strings to `Document`.
2. Then get `Element` or `Elements` by `JSoup` functions.
3. Using `for (Element e : Elements)`, `Elements` can transfer to `Element`.
4. Using `Element`, we can see the `Attributes`.



How to use functional models in `JSoup`

Write test cases

The test cases are stored in the directory - `/src/test/java/org.jsoup/swe261/FiniteStateMachinesTest.java`

The files within are written here.

To see the specific `input_html`, please see our github document.

```java
public class FiniteStateMachinesTest {

    @Test
    public void String2Document() {
        String html = input_html;
        String expStr = "<body>\n" +
                "  <p>First post! <img src=\"foo.png\"></p>\n" +
                "  <p>Second post! <img src=\"foo2.png\"></p>\n" +
                "</body>";
        System.out.println(doc.body());
        assertEquals(expStr, doc.body().toString());
    }

    @Test
    public void Document2Element() {
        String html = input_html;
        Document doc = Jsoup.parse(html);
        Element ele = doc.body();
        String expStr = "<p>First post! <img src=\"foo.png\"></p>";
        //System.out.println(ele.children());
        assertEquals(expStr,ele.child(0).toString());
        expStr = "<p>Second post! <img src=\"foo2.png\"></p>";
        assertEquals(expStr,ele.child(1).toString());
    }

    @Test
    public void Element2Elements() {
        String html = input_html;
        Document doc = Jsoup.parse(html);
        Element ele = doc.body();
        Elements eles = ele.children();
        int exp = 2;
        assertEquals(exp, eles.size());
    }

    @Test
    public void Document2Elements() {
        String html = input_html;
        Document doc = Jsoup.parse(html);
        Elements eles = doc.getElementsByTag("p");
        int exp = 2;
        assertEquals(exp, eles.size());
    }
```

```java
    @Test
    public void Elements2Element() {
        String html = input_html;
        Document doc = Jsoup.parse(html);
        Elements eles = doc.getElementsByTag("p");
        String expStr = "<p>First post! <img src=\"foo.png\"></p>";
        assertEquals(expStr,eles.get(0).toString());
        expStr = "<p>Second post! <img src=\"foo2.png\"></p>";
        assertEquals(expStr,eles.get(1).toString());
    }

    @Test
    public void Element2Attr() {
        String html = input_html;
        Document doc = Jsoup.parse(html);

        // need a better way to verify these:
        Element p = doc.body().child(0);
        assertEquals("p", p.tagName());
        assertEquals("foo > bar", p.attr("class"));
    }


}
```

To explain these code,

1. `String2Document` is the first process. This one `parse` HTML or XML Strings to `Document`

```java
    @Test
    public void String2Document() {
        String html = input_html;
        Document doc = Jsoup.parse(html);
        String expStr = "<body>\n" +
                " <p>First post! <img src=\"foo.png\"></p>\n" +
                " <p>Second post! <img src=\"foo2.png\"></p>\n" +
                "</body>";
        System.out.println(doc.body());
        assertEquals(expStr, doc.body().toString());
    }
```

Reversely, `Document` can transfer to HTML/XML

```
@Test
    public void Document2Element() {
        String html = input_html;
        Document doc = Jsoup.parse(html);
        Element ele = doc.body();
        String expStr = "<p>First post! <img src=\"foo.png\"></p>";
        //System.out.println(ele.children());
        assertEquals(expStr,ele.child(0).toString());
        expStr = "<p>Second post! <img src=\"foo2.png\"></p>";
        assertEquals(expStr,ele.child(1).toString());
    }
```

2. Using `for (Element e : Elements)`, `Elements` can transfer to `Element`.

```
@Test
    public void Element2Elements() {
        String html = input_html;
        Document doc = Jsoup.parse(html);
        Element ele = doc.body();
        Elements eles = ele.children();
        int exp = 2;
        assertEquals(exp, eles.size());
    }
```

Reversely, `Elements` can transfer to `Element`.

```
@Test
    public void Elements2Element() {
        String html = input_html;
        Document doc = Jsoup.parse(html);
        Elements eles = doc.getElementsByTag("p");
        String expStr = "<p>First post! <img src=\"foo.png\"></p>";
        assertEquals(expStr,eles.get(0).toString());
        expStr = "<p>Second post! <img src=\"foo2.png\"></p>";
        assertEquals(expStr,eles.get(1).toString());
    }
```

3. Using `Element`, we can see the `Attributes`.

```java
@Test
    public void Element2Attr() {
        String html = input_html;
        Document doc = Jsoup.parse(html);

        // need a better way to verify these:
        Element p = doc.body().child(0);
        assertEquals("p", p.tagName());
        assertEquals("foo > bar", p.attr("class"));
    }
```

Part III: Structural (White Box) Testing.

Introduction to structural testing

Structural testing is the type of testing performed to test the structure of the code. Also called white box test or glass box test. This type of testing requires knowledge of the code, so in most cases it is done by the developer. It is more concerned with how the system works, rather than the function of the system. It provides more coverage for testing.

It is a supplement to functional testing. Using this technology, you can first analyze test cases drafted according to system requirements, and then you can add more test cases to increase coverage. It helps to test the software comprehensively. Most structural testing is automated.

Advantage

- Gives a more exhausted testing for the software.
- Helps you find defects as early as possible.
- Helps eliminate invalid codes.
- No time wasted, because it is mostly automated.

Disadvantage

- Need to understand the code.
- Need to use test tools for training
- It is expensive.

Coverage tool we use

JaCoCo is a code coverage library for Java, which was created by the EclEmma team based on years of experience in using and integrating existing libraries.

Compared with Eclemma used in the class, JaCoCo is also produced by the same company and have mostly the same functions. We tried other tools as well mentioned in the reference, but they do not work as good as JaCoCo, for example, cuberuto. It can produce a html website page to tell you the coverage for classes, methods and etc. Example as belows. It seems very neat and clear.

jsoup Java HTML Parser

jsoup Java HTML Parser

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
org.jsoup.parser		85%		75%	439	1,612	731	3,712	36	550	0	114
org.jsoup.nodes		90%		87%	131	840	146	1,571	44	423	0	30
org.jsoup.examples		0%		0%	42	42	100	100	15	15	4	4
org.jsoup.helper		88%		82%	91	414	115	988	28	195	0	12
org.jsoup.select		89%		91%	77	498	57	886	38	244	0	58
org.jsoup.safety		93%		78%	30	126	27	330	7	63	0	10
org.jsoup		82%		n/a	11	37	17	61	11	37	2	6
org.jsoup.internal		94%		91%	14	103	12	171	3	37	0	5
Total	4,906 of 36,085	86%	780 of 4,001	80%	835	3,672	1,205	7,819	182	1,564	6	239

How to add Jacoco?

Go to our maven project, find our `pom.xml` file, and add as in the file. In the file, group id is `org.jacoco`, artifactid is `jacoco-maven-plugin`, version is `0.8.3`.

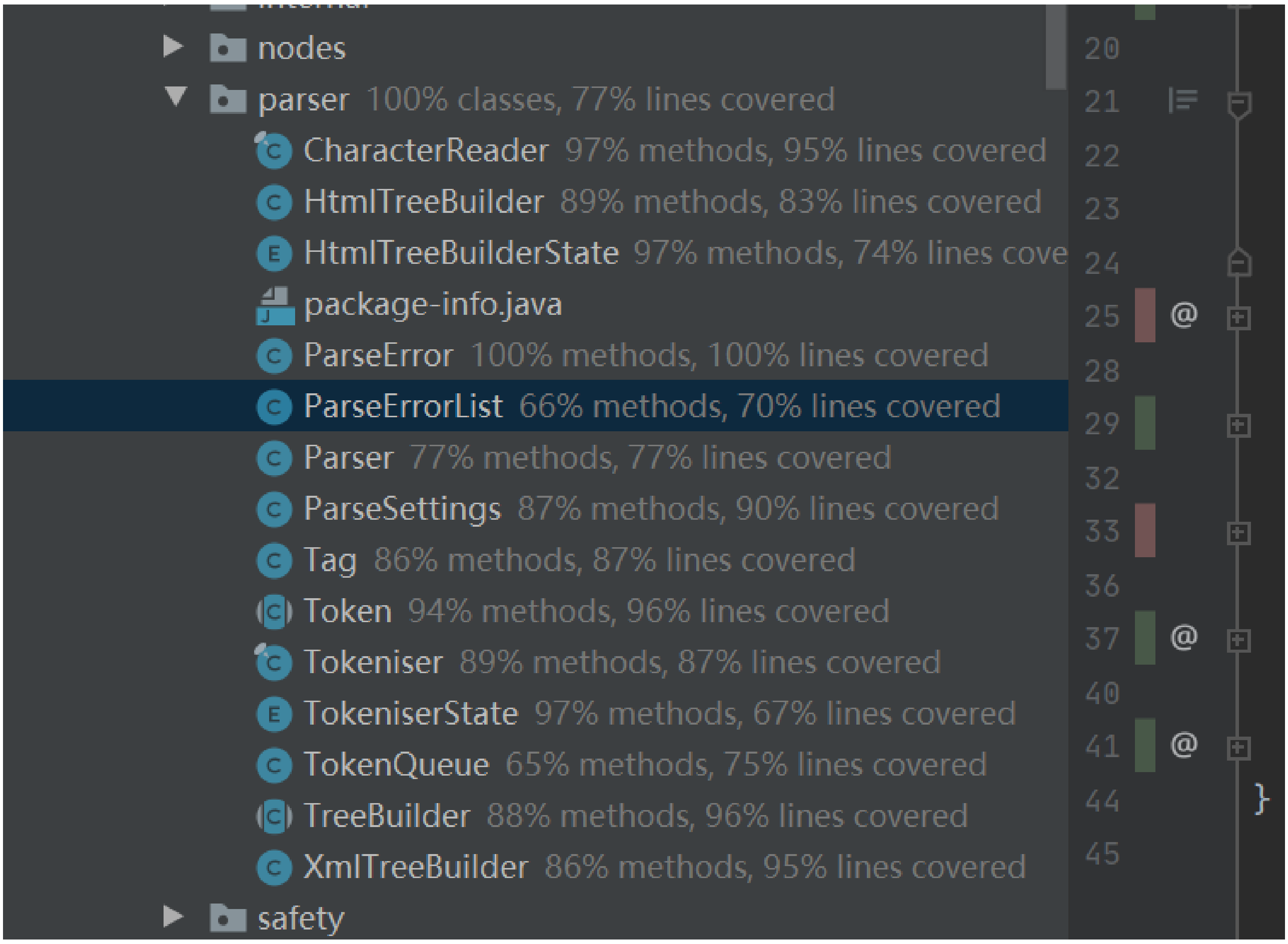
The meaning of the stars is very important, otherwise, you cannot test our project. `**` is to match the folders or directories. `*` is to match >=0 characters

Coverage for JSoup

We want to focus on `org.jsoup.parser`, and its coverage is below, such as line, branch, and method coverage. We put it into a table. Unfortunately, JaCoCo didn't provide the exact number for missed or total for branch, but only its coverage.

MEASURES	MISSED	TOTAL	COVERAGE
line	731	3712	80%
branch	N/A	N/A	75%
method	36	550	93%

One interesting function for **JaCoCo** is that it can mention the percentage of coverage for methods and lines in the JetBrain Idea file. So, as you can see, this is the example for parser file and its coverage for methods and lines.

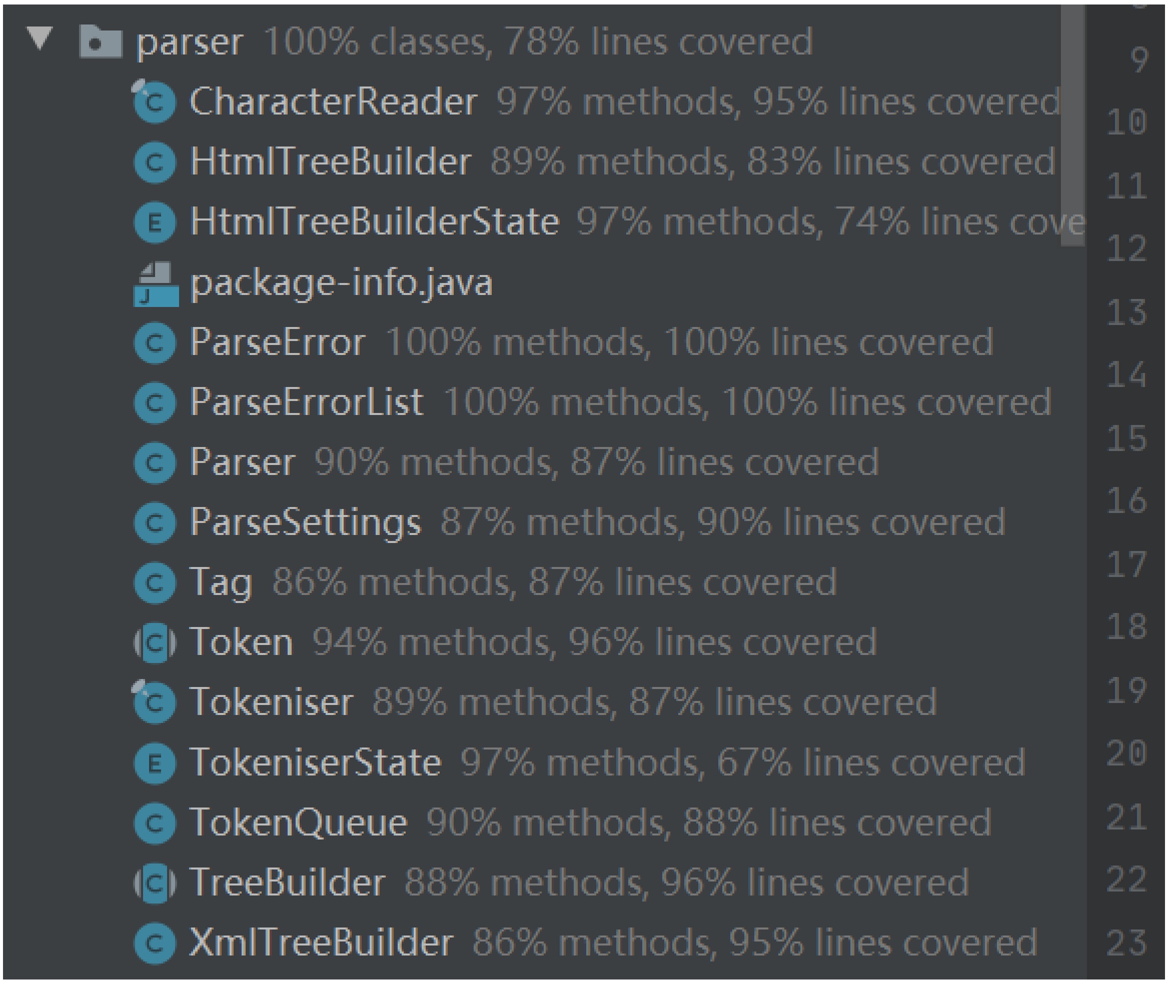


For `parser` folder, we focus on `ParseErrorList`, which only has 66% methods, 70% lines covered. `Parser` only has 77% methods, 77% lines covered. `TokenQueue` only has 65% methods, 75% lines covered. To improve this, we write new test cases afterwards. Parser is the core function for JSoup, because JSoup uses it to parse the HTML or XML file into its own classes. Using parser, you can get its elements, and attributes, which is very important and vital.

New test case

We put our improvement code in the folder `/src/test/java/org.jsoup/parser/ParseImprove.java`. Compared with the former method and coverage, this time, the result is much more improved.

Also, using the interesting function mentioned above, it can show the percentage of coverage for methods and lines for the parser folder.



The coverage before and after are documented in the table below.

FUNCTION	METHOD BEFORE	METHOD AFTER	LINE BEFORE	LINE AFTER
ParseErrorList	66%	100%	70%	100%
Parser	77%	90%	77%	87%
TokenQueue	65%	90%	75%	88%

To explain the code, we wrote 6 methods to improve these three java files.

First, function `parseErrorListTest` improve `getMaxSize()`, `ParseErrorList()` in `ParseErrorList`.

```
@Test
    public void parseErrorListTest() {
        ParseErrorList testList = new ParseErrorList(16,3);
        ParseErrorList copyList = new ParseErrorList(testList);
        //Assert
        assertEquals(3,copyList.getMaxSize());
    }
```

Second, function `parserTest` improve `setTreeBuilder()`, `isTrackErrors()`, `isContentForTagData()` in `Parser`.

```
@Test
    public void parserTest() {
        TreeBuilder treeBuilder = new HtmlTreeBuilder();
        Parser testParser = new Parser(treeBuilder);
        TreeBuilder testTreeBuilder = new HtmlTreeBuilder();
        //Parser copyParser = new Parser(testParser);
        testParser.setTreeBuilder(testTreeBuilder);
        //Assert
        assertEquals(false,testParser.isTrackErrors());
        assertEquals(false,testParser.isContentForTagData("123"));
    }
```

Third, function `parserTest2` improve `setTreeBuilder()`, `isTrackErrors()`, `isContentForTagData()` in `Parser`.

```
@Test
    public void parserTest2() {
        TreeBuilder treeBuilder = new HtmlTreeBuilder();
        Parser testParser = new Parser(treeBuilder);
        //Assert
        assertEquals(false,testParser.isContentForTagData("123"));
    }
```

Four, Five and Six. function `TokenQueueTest` improve `peek()`, `addFirst()`, `matchesCS()`, `matchesAny()`, `advance()`, `consumeTagName()`.

```
@Test
    public void TokenQueueTest() {
        TokenQueue testTokenQueue =  new TokenQueue("abcdefg");
        //Assert
        assertEquals('a',testTokenQueue.peek());
        testTokenQueue.addFirst('z');
        //Assert
        assertEquals('z',testTokenQueue.peek());
    }

    @Test
    public void TokenQueueTest2() {
```

```
        TokenQueue testTokenQueue =  new TokenQueue("abcdefg");
        assertEquals(false, testTokenQueue.matchesCS("asc"));
        assertEquals(true, testTokenQueue.matchesAny('a'));
        assertEquals(false, testTokenQueue.matchesStartTag());

    }

    @Test
    public void TokenQueueTest3() {
        TokenQueue testTokenQueue =  new TokenQueue("abcdefg");
        testTokenQueue.advance();
        assertEquals("bcdefg",testTokenQueue.chompTo("qwe"));
        testTokenQueue.consumeTagName();
    }
```

Part IV: Continuous Integration

Introduction to continuous integration

What is continuous integration? It means the practice of merging the working copies of all developers into the shared mainline several times a day. This kind of software testing technique is proposed by Grady Booch. It has four important components.

- Firstly, continuous integration is originated from extreme programming development process. Due to high amount of development, there would be lots of changes of code and different progress align with different colleagues. It is super important under such extreme programming environment
- Secondly, continuous integration needs to be performed, even for minor changes. Therefore, every developer are in the same pace for the project.
- Thirdly, every developer in the group and in the project needs to commit their changes every day. Therefore, every developer are in the same pace for the project.
- Forthly, every version, especially the latest version, needs to build and pass all the tests. Otherwise, this program cannot work, and need to check its methods and functions.

Importance of continuous integration

After talking about the basic concepts of continuous integration, we come to the question why we need continuous integration. Combined with its four important components, we concluded five reasons of necessity of continuous integration.

- Firstly, combined with all of the four basic concepts, it will be quicker to predict the total development time for this project.
- Secondly, due to the second, third, forth basic concepts, it's easier to detect bugs.
    - Because developer need to commit their changes even for minor changes, it is way easier to detect bugs. We can *strangled the baby bugs in the cradle*
    - Also, bugs can be detected separately because each change is separated. It's easy to trace each commit and each bug.
    - Here, we can use differential debugging. It can help by comparing known good codes with faulty codes.

Using continuous integration

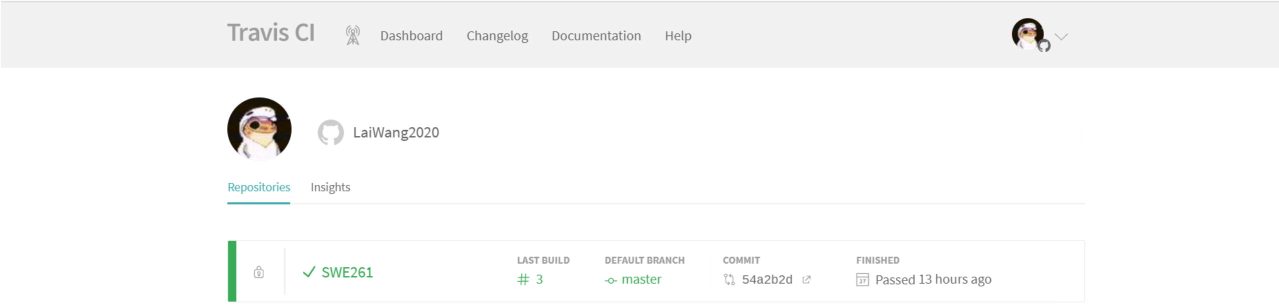
Talking about so many advantages of continuous integration, next step, we want to combine continuous integration with our `JSoup` project.

With the help of TravisCI

First of all, we need to sign up for TravisCI. TravisCI is a very famous tool for continuous integration. We can easily sync our projects with TravisCI and test it. According to the official document, we can use TravisCI in the follwing steps.

- Push our code to github
- Github triggers TravisCI file to build our project and see how it works.
- Appears whether our build passes or fails. (Hope it passes!)
- TravisCI deploys to Heroku
- TravisCI tells our account how it works.

We signed up for an account for TravisCI named LaiWang2020, and sync up with our github account. After signing up, it can choose one / more of your repository to build. We are testing with our `JSoup` project, which is named `SWE261`.



Create a file to build

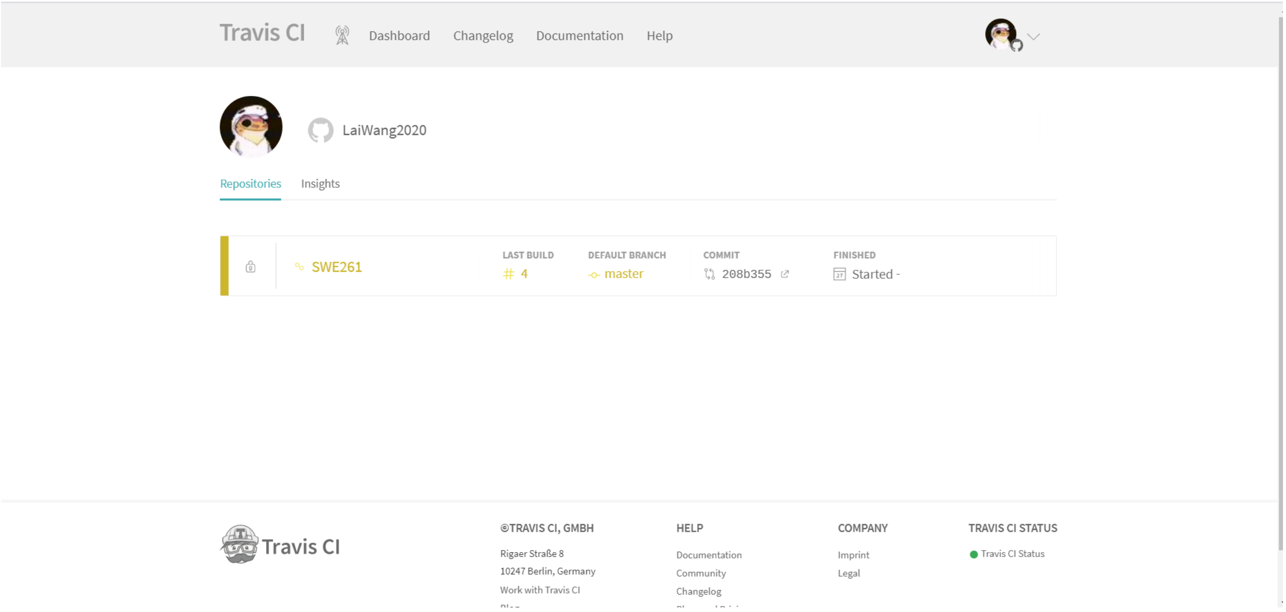
To use TravisCI, we need to create a .travis.yml file for our project. Because TravisCI cannot choose a repository that doesn't belong to you, even for joint collaboration, Lai Wang signed up for an account and synced his GitHub. He forked our `JSoup` project again, and TravisCI project is firstly done in here: https://github.com/LaiWang2020/SWE261/blob/master/.travis.yml. .travis.yml file contains information below.

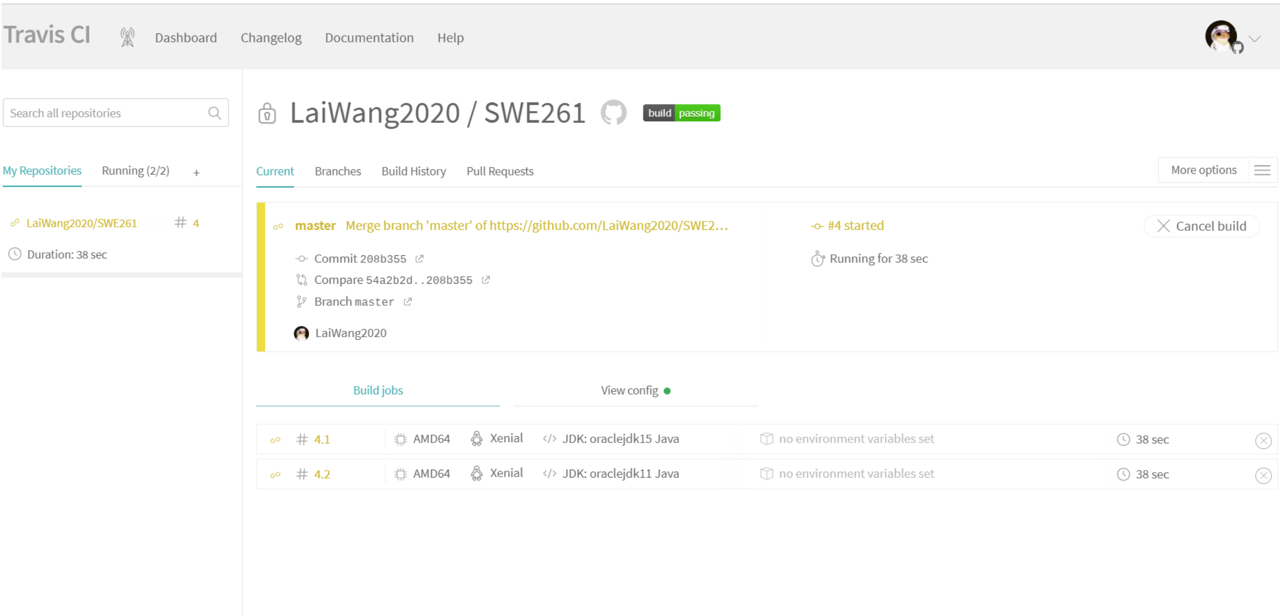
```
language: java

jdk:
  - oraclejdk15
  - oraclejdk11
```

Commit the changes and verify

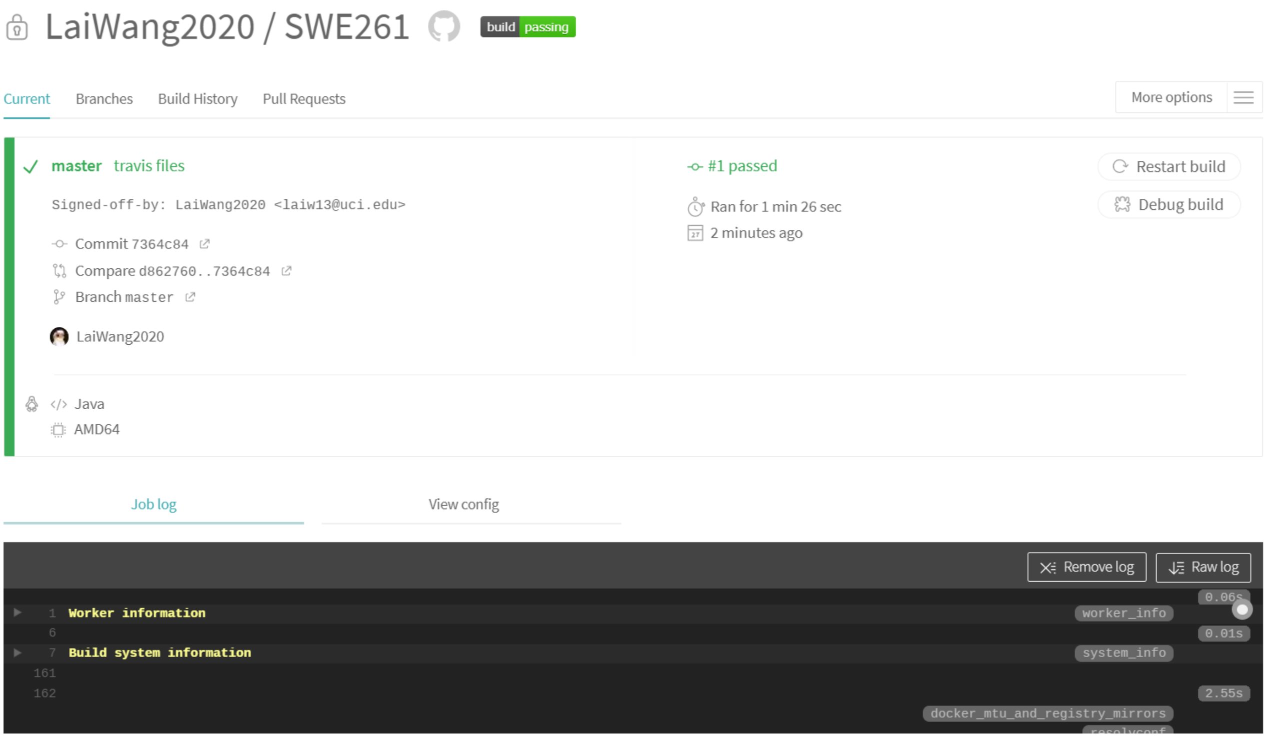
After we add this .travis.yml file, we committed our repository from local to GitHub. Our TravisCI will appear like this.



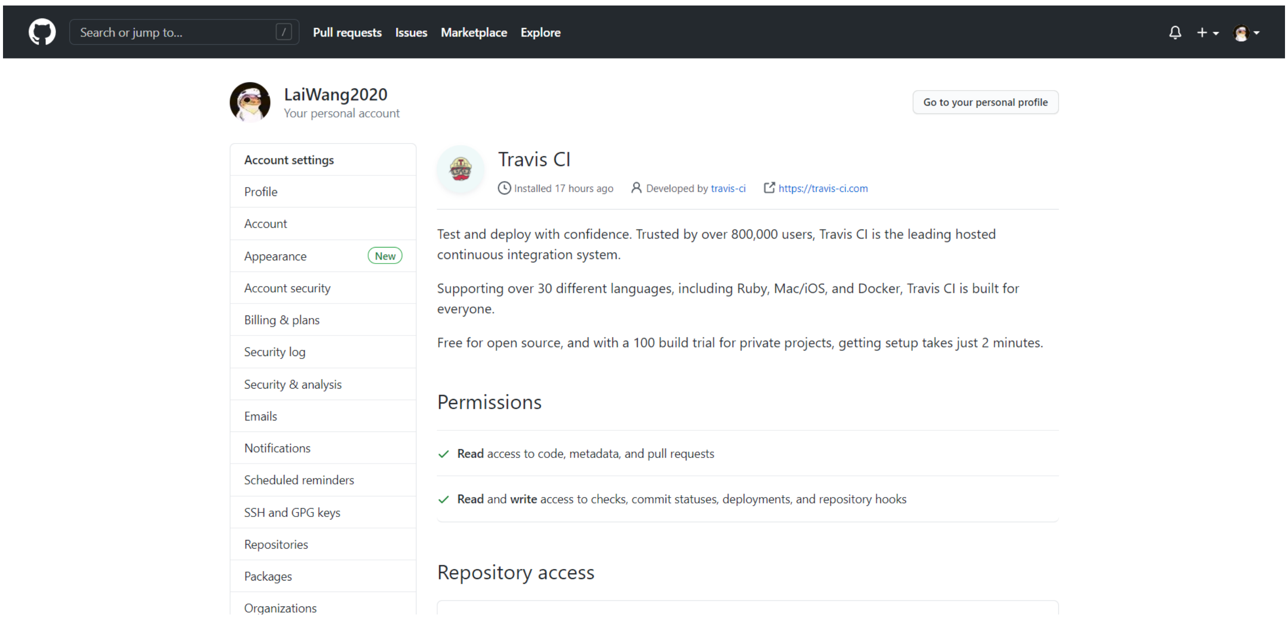


After approximately three minutes, the build process will be done, and it appears like this.

It ran for 1 min 26 sec, and build passes.



Adding extension to GitHub, provided by TravisCI, GitHub can also appear the result of our project `JSoup`. The extension is appeared in the account settings



And the `build passing` is appeared in our repository.

Awesome! It;s more like a professional code repository.

Part V: Testable Design

Introduction to Testable Design

What is testable design? The basic value proposition of testable design is to be able to test code better. As Roy Osherove said[1], testable design is “a given piece of code should be easy and quick to write a unit test against.”

Guidelines for Testable Design

After talking about the concept of testable design, you might still have question on how to make a testable design. There are some rules to follow as below.

- In testable design, we need to avoid complex `private` methods. The reason for this rule is quite clear. In Java, we cannot test with `private` methods. Under this circumstance, when there are some complex `private` methods, we cannot test with them.
- In testable design, we need to avoid `static` methods. The reason for this rule is also Java's definitions. In Java, `static` method operate on the class instead of the object.
- In testable design, we need to be careful to hardcode in `new`. If so, object cannot be stubbed.
- In testable design, we need to avoid logic in constructors. It's difficult to bypass a constructor because subclass constructors always trigger at least one superclass constructor.
- In testable design, we need to avoid singleton pattern.

JSoup testable design

In the `JSoup` functions, in the folder `src/main/java/org.jsoup/nodes/Element`, we have this function called `childElements`. As we talked about in the Guidelines for Testable Design section, we try to avoid complex `private` method because we cannot test with them. In this function, we also have this function is private so it cannot be tested.

Original Code is as following.

```
List<Element> childElements() {
    return children;
}
```

To change it into a testable design, we set it into `public`, and the new function name is set to ``.

To write new test case for our testable design, we put that into `src/test/java/org.jsoup/swe261`. The test function is called `TestableDesign.java`

```
@Test
    public void childListTest(){
        String html = ""; // To see the original html string, 
please refer to our code
        Document doc = Jsoup.parse(html);
        Element ele = doc.body();
        System.out.println(ele.childElementsV2().get(0));
        System.out.println(ele.childElementsV2().get(1));
        String exp = "<p>First post! <img src=\"foo.png\"></p>";
        String exp2 = "<p>Second post! <img src=\"foo2.png\"></p>";
        assertEquals(2,ele.childElementsV2().size());
        assertEquals(exp,ele.childElementsV2().get(0).toString());
        assertEquals(exp2,ele.childElementsV2().get(1).toString());
    }
```

Introduction to Mocking

What is mocking? To understand mocking, we firstly need to understand the word "**mock**" - A fake object that decides whether a unit test has passed or failed by watching interactions between objects.

Importance of Mocking

After talking about the basic concepts of mocking, we come to the question why we need mocking. Combined with its concept, we concluded four reasons of necessity of mocking.

- Mocking process can simulate external dependencies. Without mocking, if a test case fails, we don't know whether the failure is because of our code unit or because of our code dependencies.
- Mocking process can promote the interaction between objects.
- During development, mocking can help developers start testing early because mocking also support demos and evaluations. All units of the project can be carried out in parallel without having to wait for everyone to be ready.
- Mocking can help us avoid repeating test code in similar tests.

Mock JSoup Now!

Mock with function normaliseDocumentNodes

The feature we chose to mock JSoup is `normaliseDocumentNodes`. This function's original code is as follows.

```
public Element normaliseDocumentNodes() {
        //Element htmlEl = htmlEl(); // these all create if not
found htmlNode

        Element head = head2();
        body2();
        // pull text nodes out of root, html, and head els, and
push into body. non-text nodes are already taken care
        // of. do in inverse order to maintain text order.
        normaliseTextNodes2(head);
        normaliseTextNodes2(ele.htmlel());
        normaliseTextNodes2(this);

        ensureMetaCharsetElement();

        return this.ele.htmlel();
    }
```

But, wait. Why we need mocking to test with this function, not other methods?

As we mentioned in the "Importance of Mocking" section, the third reason is that mocking can help developers start testing early. To test with this function, we need `element`, which is external dependency. What are we testing? The interaction between `element` and `document`. Theoretically, if we want to test with this function, we need to test with element first. If element is invalid, we cannot test with this function. But, using mocking, we can test with this function easily because we already know the output of element!

Test with function normaliseDocumentNodes now

The feature we chose to mock JSoup is `normaliseDocumentNodes`.

Firstly, we set up this test and use `mock` function here.

```
@Before
    public void setup() {
        ele = mock(Element.class);
        doc = new Document("<html></html>",ele);
        MockitoAnnotations.initMocks(this);
    }
```

Then we get to test this function by mockito. We used the function

- `when` is to set the return value
- `assert` is to judge if the result is equal
- `verify` is to verify how many times the function runed

The code is as following

```
@Test
    public void mockitoTest1(){
        String exp = "";//To see the original html string, please refer to our code
        when(ele.childElementsList()).thenReturn(getEleList());
        when(ele.htmlel()).thenReturn(getEle());
        System.out.println(doc.normaliseDocumentNodes());

 assertThat(doc.normaliseDocumentNodes().toString()).isEqualTo(exp);
        verify(ele, times(4)).childElementsList();
        verify(ele, times(4)).htmlel();
    }
```

Part VI: Static Analyzers

Introduction to Static Analysis

What is a static analysis? Usually, static analysis means code review. In companies, especially in one project, colleagues will review your code manually.

- In one word, static analysis is mostly informal manual-human reviews of code.
- Also called "code reviews" or "compile-time analysis".

During our experience in `JSoup` project, we fork our project into our own repository and pull request into the master. One or two of our collaborators will peer view the PR code, give it some advice or directly merge it into the master.

There are best practices for the process of static analyzers.

1. Review small portions of code at a time
2. Record all feedback
3. Review code independently before gathering to discuss
4. Use checklists

Importance of Static Analysis

After talking about the basic concepts of static analysis, we come to the question why we need static analysis. Combined with its basic concepts, we concluded reasons of necessity of static analysis.

- Static analysis can help you find potential bugs early. Using static analysis, your code reviewers can help you find these bugs manually.
- Static analysis can help you stick to the same coding style or coding standard. Since more than one people are involved in one function or one line of code, the clarity of code will increase. In my opinion, to raise the possibility of merging code, you are not the only one who sees the code so that you need to make more comments and try to make the code neat and clear.
- Static analysis can help team collaboration. Your team need to involve in the same project and be responsible for every line of code. To achieve that, your team members must discuss more often and share your ideas on code.

Tools for Static Analysis

We used two different static analyzers on our project `JSoup`.

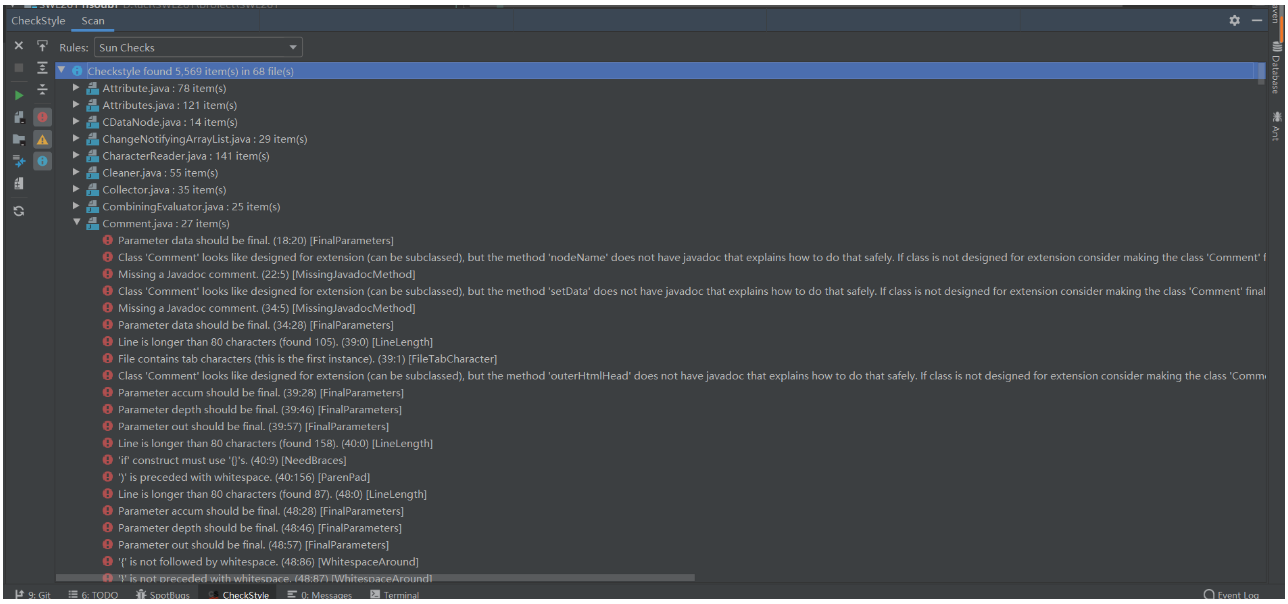
The first one is **SpotBugs**. SpotBugs is a program that uses static analysis to find errors in Java code.

The second one is **Checkstyle**. Checkstyle is a development tool that helps programmers write Java code that meets coding standards. Sample coding standard: google coding style or sun check.

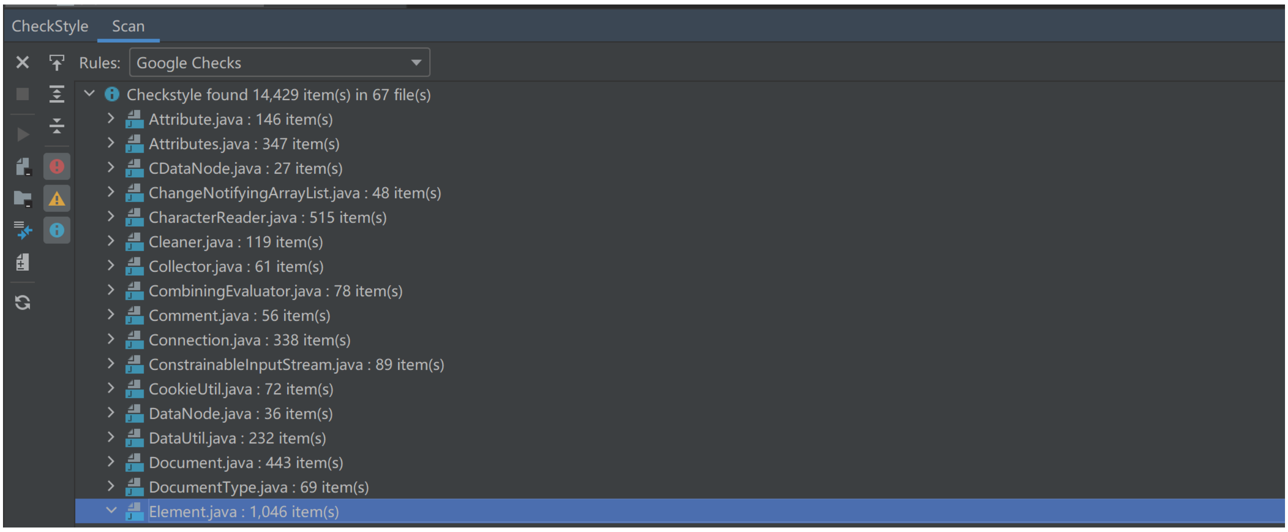
Results of Static Analysis by Spotbugs and Checkstyle

CheckStyle

Firstly, we used the checkstyle tool. I pressed the "Check Module" button and the result is as below. This used the sun checks method.



The following picture used the google checks method.



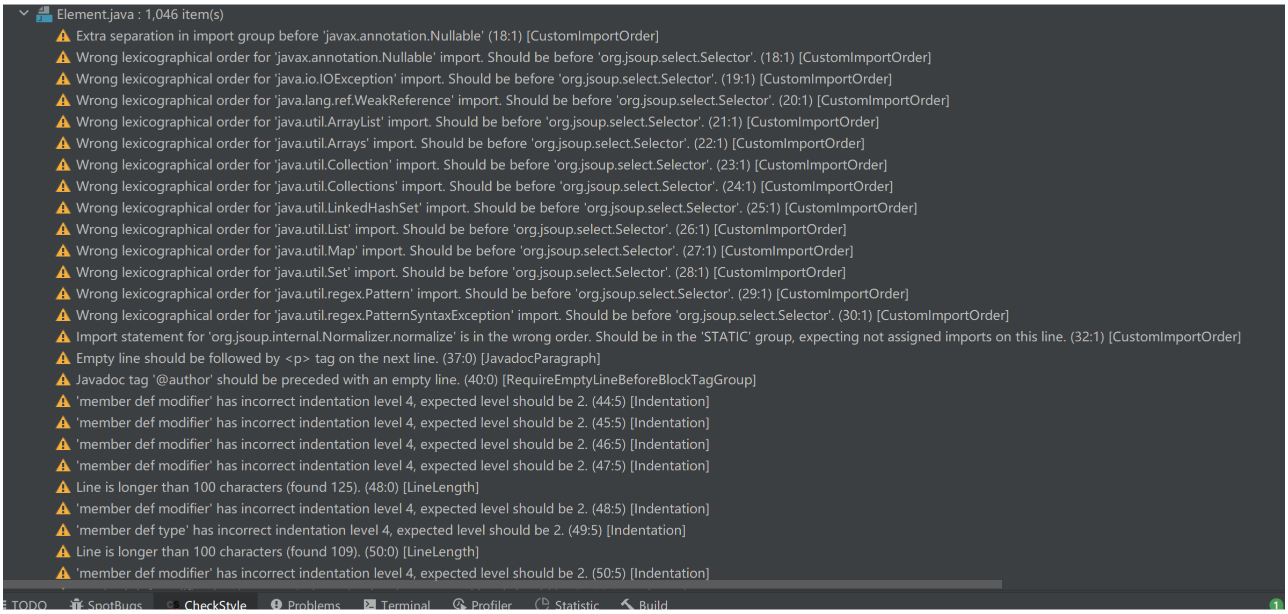
Deep dive into some errors

The first error I met with is `CustomImportError`. Actually it emphasised on the order of import, not an actual bug that you will see in the command line.

The second error I met with is `Indentation`. It used 4 tabs instead of 2 tabs.

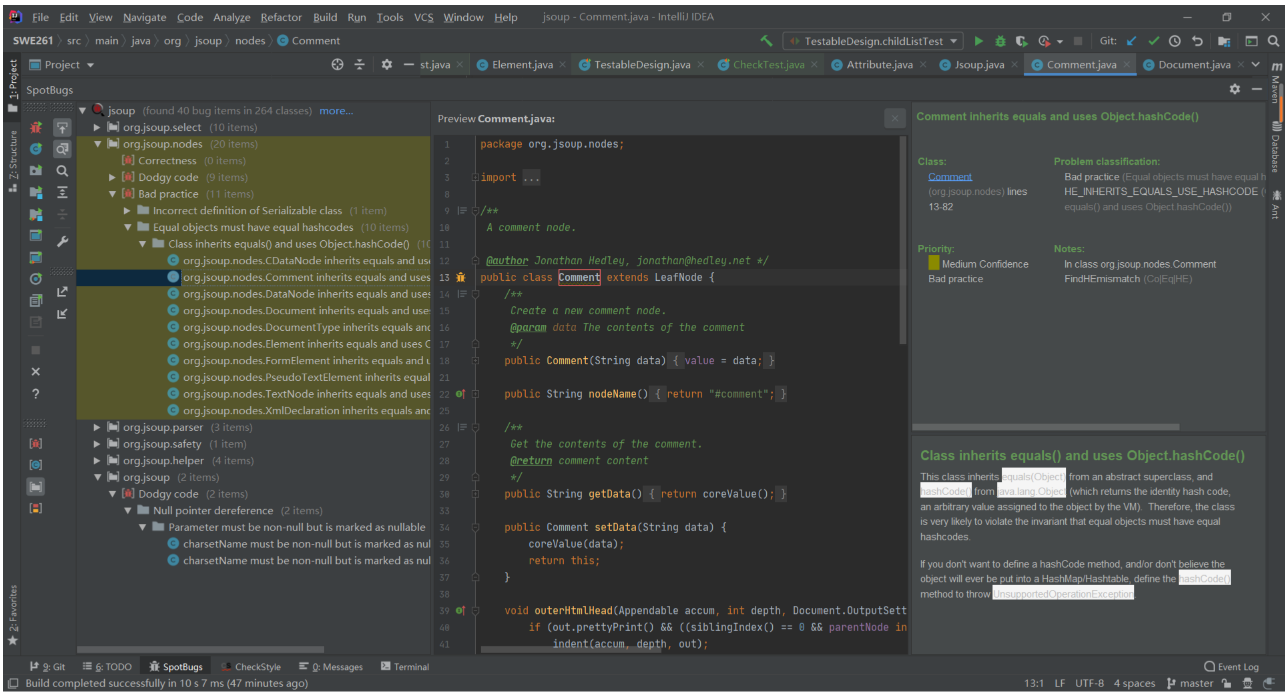
The third error I met with is `RequiredEmptyLineBeforeBlockTagGroup`. It requires us to have an empty line before tag `@Return`.

From my perspective, these errors existed, but they aren't actual problems in the code. They are more likely to be the coding style error instead of bugs.



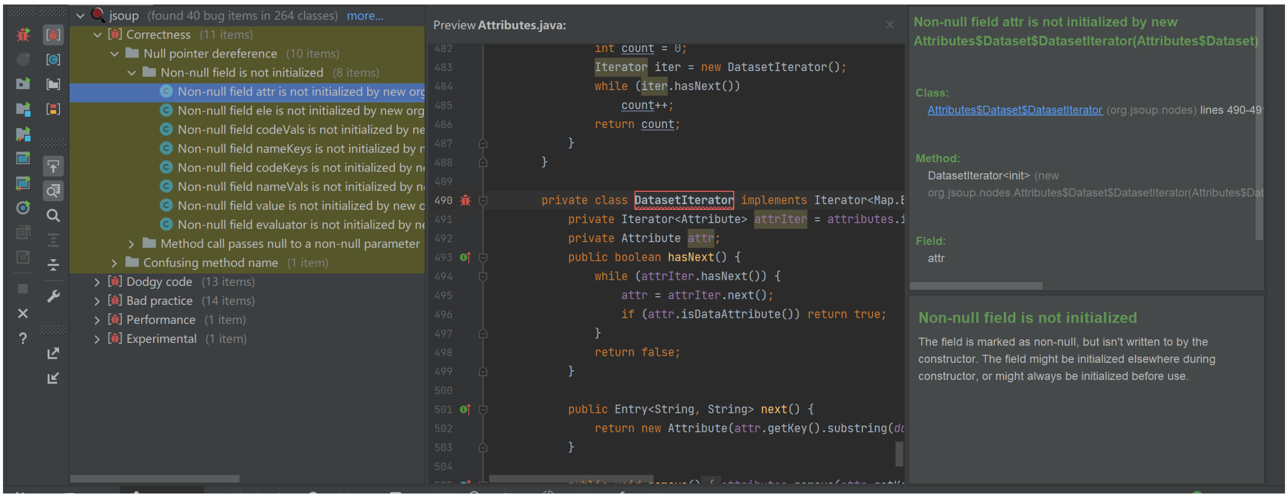
Spotbugs

Next, we used the spotbugs tool. Different from the checkstyle tool, it is divided into folders and each java file, and it will spotlights the lines and functions or classes that has some type error or bugs.

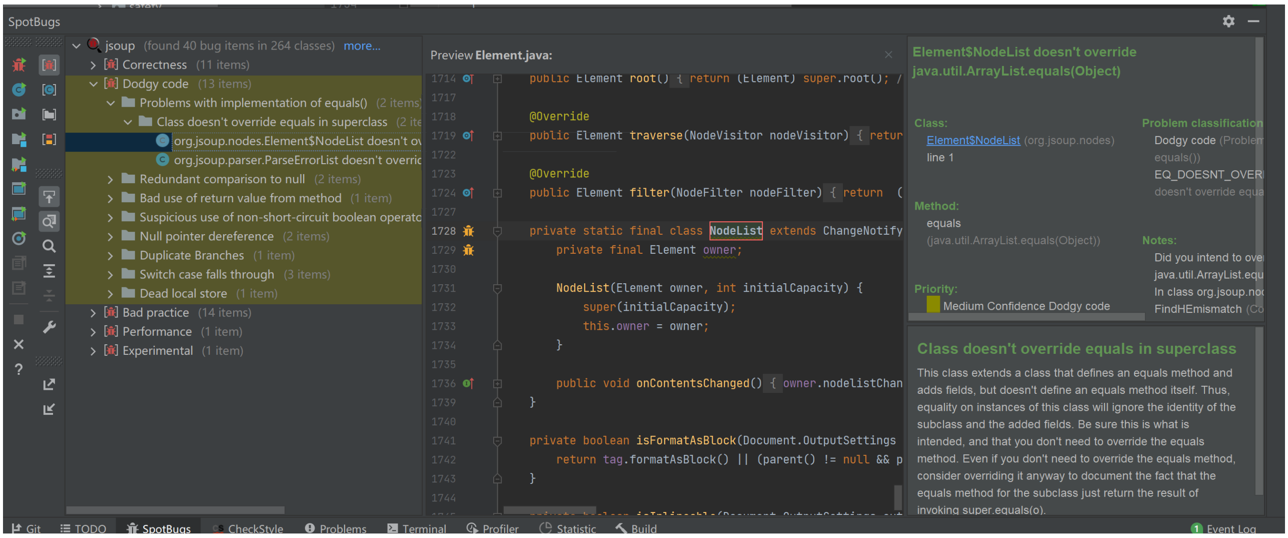


Deep dive into some errors

The first error in **correctness** is that non-null field is not initialized. From my perspective, it is mentioning that `JSoup` didn't initialize the parameter for the class `DatasetIterator`



The second error in **dodgy code** is that classes doesn't override equals in superclass. Because it used `extend`, and it should override `equals` function. But as long as it didn't use it, it should be fine that this class doesn't override equals in superclass.



Contrast between Spotbugs and Checkstyle

The information for each tool is quite different.

For spotbugs, this one is more neat and clear with description and code. Every error is categorized into five folders.

- Correctness
- Dodgy code

- Bad practice
- Performance
- Experimental

For checkstyle, this one is more messy. Every error message is put under its original file name. Once you press the error information, you will jump into the error line. The error most frequently appeared are

- CustomImportError
- LineLength(too long)
- Indentation
- RequiredEmptyLineBeforeBlockTagGroup

They are more likely to be the coding style error instead of bugs.

Team Members

Sun Yu(http://github.com/duke326/)

Lai Wang(https://github.com/laiwang2020/)

Xinyi Hu(https://github.com/samaritanhu)

Reference

---

[1] https://livebook.manning.com/book/effective-unit-testing/chapter-7/8

[2] https://www.vogella.com/tutorials/JUnit/article.html

[3] https://www.softwaretestingclass.com/what-is-structural-testing/

[4] https://www.tutorialspoint.com/software_testing_dictionary/structural_testing.htm

[5] https://stackify.com/code-coverage-tools/

[6] https://www.eclemma.org/jacoco/

[7] https://devopedia.org/mock-testing