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.

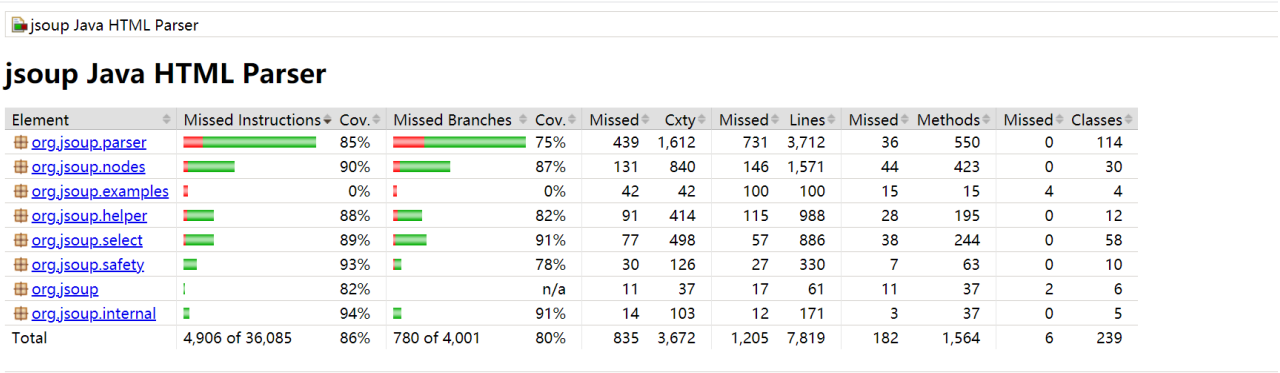
Disadvantage

- 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`.

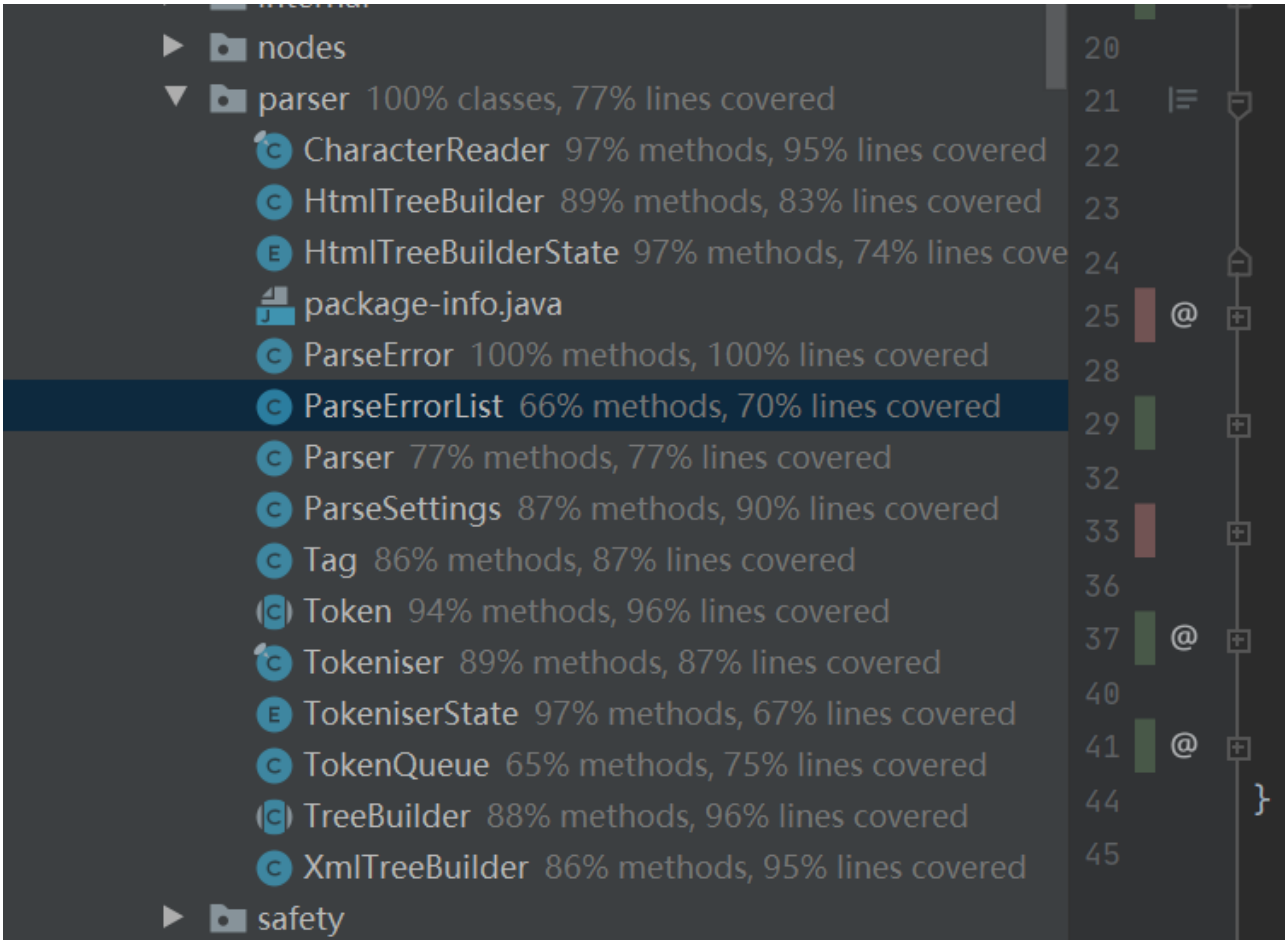
The meaning of the stars is very important, otherwise, you cannot test our project. `**` is to match the folders or directories. `*` is to match >=0 characters

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.

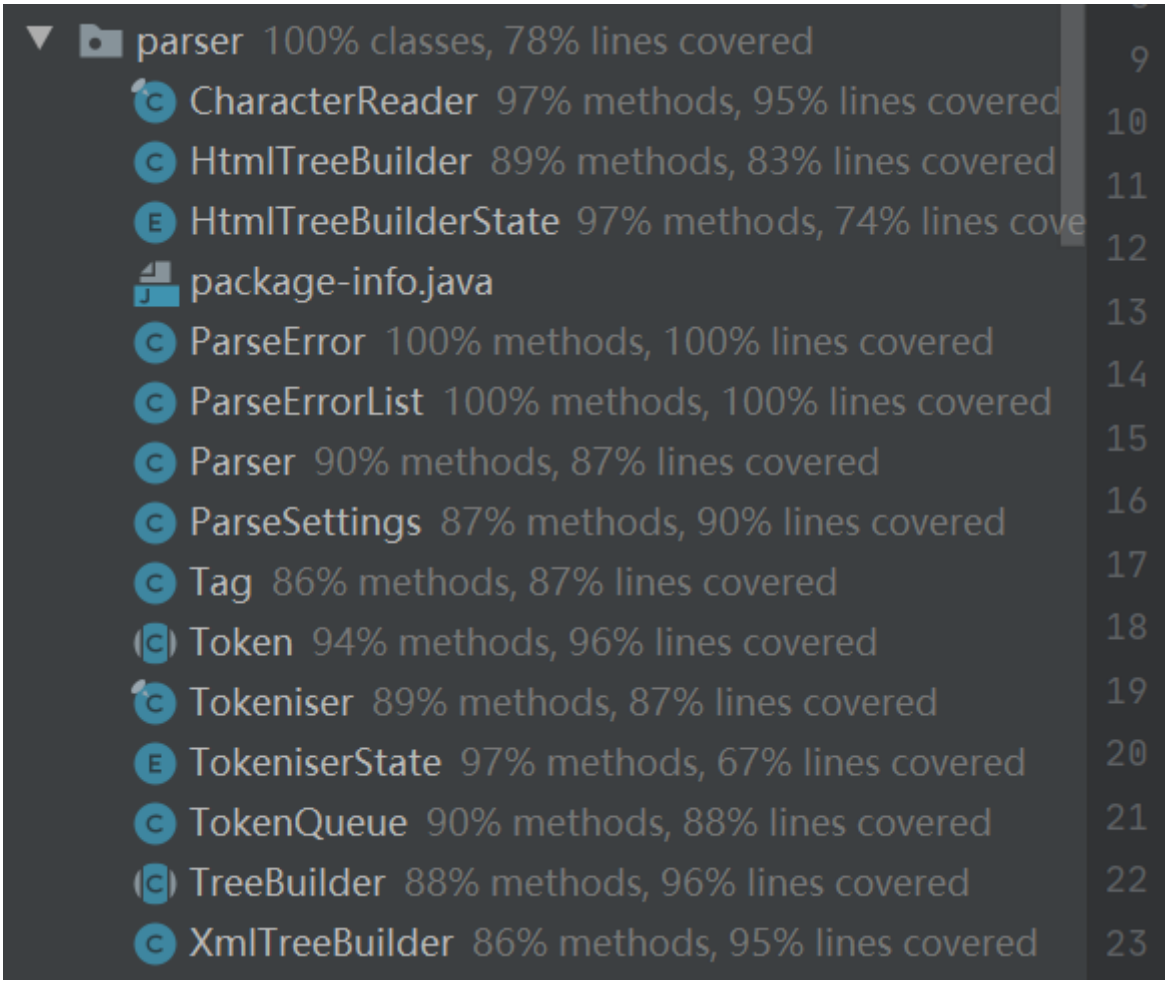


For `parser` folder, we focus on `ParseErrorList`, which only has 66% methods, 70% lines covered. `Parser` only has 77% methods, 77% lines covered. `TokenQueue` only has 65% methods, 75% lines covered. To improve this, we write new test cases afterwards. Parser is the core function for JSoup, because 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();
    }
```

Reference

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

https://www.tutorialspoint.com/software_testing_dictionary/structural_testing.htm

https://stackify.com/code-coverage-tools/

https://www.eclemma.org/jacoco/