Part III: Structural (White Box) Testing.

Introduction to structural testing

Structural testing is the type of testing carried out to test the structure of code. It is also known as White Box testing or Glass Box testing. This type of testing requires knowledge of the code, so, it is mostly done by the developers. It is more concerned with how the system does it rather than the functionality of the system. It provides more coverage for the testing. For example, to test certain error messages in an application, we need to test the trigger condition for it, but there must be many triggers for it. It is possible to miss out on one while testing the requirements drafted in SRS. But using this testing, the trigger is most likely to be covered since structural testing aims to cover all the nodes and paths in the structure of code.

It is complementary to Functional Testing. Using this technique the test cases drafted according to system requirements can be first analyzed and then more test cases can be added to increase the coverage. It can be used on different levels such as unit testing, component testing, integration testing, functional testing, etc. It helps in performing thorough testing on software. The structural testing is mostly automated.

Structural Testing Techniques:

- **Statement Coverage -** This technique is aimed at exercising all programming statements with minimal tests.
- **Branch Coverage -** This technique is running a series of tests to ensure that all branches are tested at least once.
- **Path Coverage -** This technique corresponds to testing all possible paths which means that each statement and branch are covered.

Calculating Structural Testing Effectiveness:

```
Statement Testing = (Number of Statements Exercised / Total Number
of Statements) x 100 %

Branch Testing = (Number of decisions outcomes tested / Total
Number of decision Outcomes) x 100 %

Path Coverage = (Number paths exercised / Total Number of paths in
the program) x 100%
```

Advantage

- Provides a more thorough testing of the software.
- Helps finding out defects at an early stage.
- Helps in eliminating dead code.
- Not time consuming as it is mostly automated.

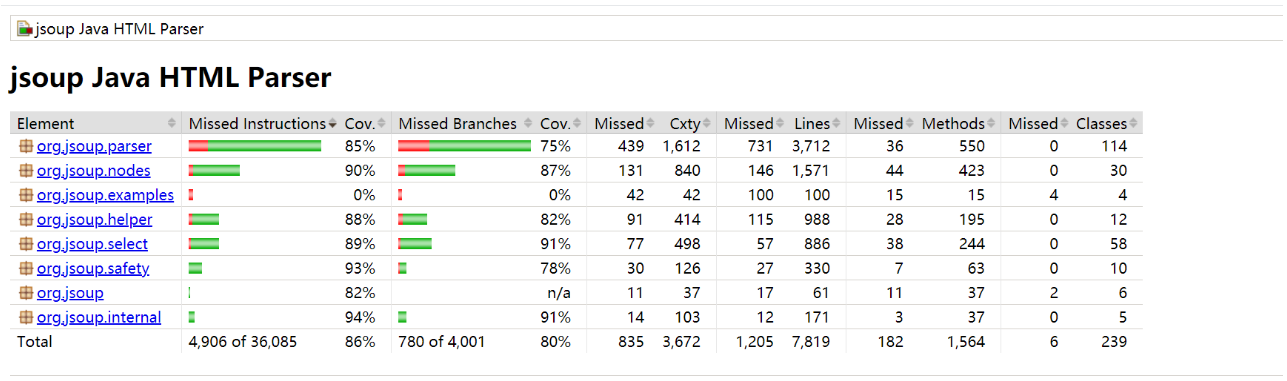
Disadvantage

- Requires knowledge of the code.
- Requires training in the tool used for testing
- It is expensive.

Coverage tool

For the test suite for your project, run a coverage tool. Document the coverage of the existing test suite (before you add any further test cases than what you have already added). Report various coverage measures, such as line, branch, and method coverage. Document some parts of the code that are currently uncovered by the existing test suite.

JaCoCo is a free code coverage library for Java, which has been created by the EclEmma team based on the lessons learned from using and integration existing libraries for many years.



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 belows.

```
<plugin>
        <groupId>org.jacoco</groupId>
        <artifactId>jacoco-maven-plugin</artifactId>
        <version>0.8.3</version>
        <configuration>
          <includes>
            <include>**/**/*</include>
          </includes>
        </configuration>
        <executions>
          <execution>
            <id>pre-test</id>
            <goals>
              <goal>prepare-agent</goal>
            </goals>
          </execution>
          <execution>
            <id>post-test</id>
            <phase>test</phase>
            <goals>
              <goal>report</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
```

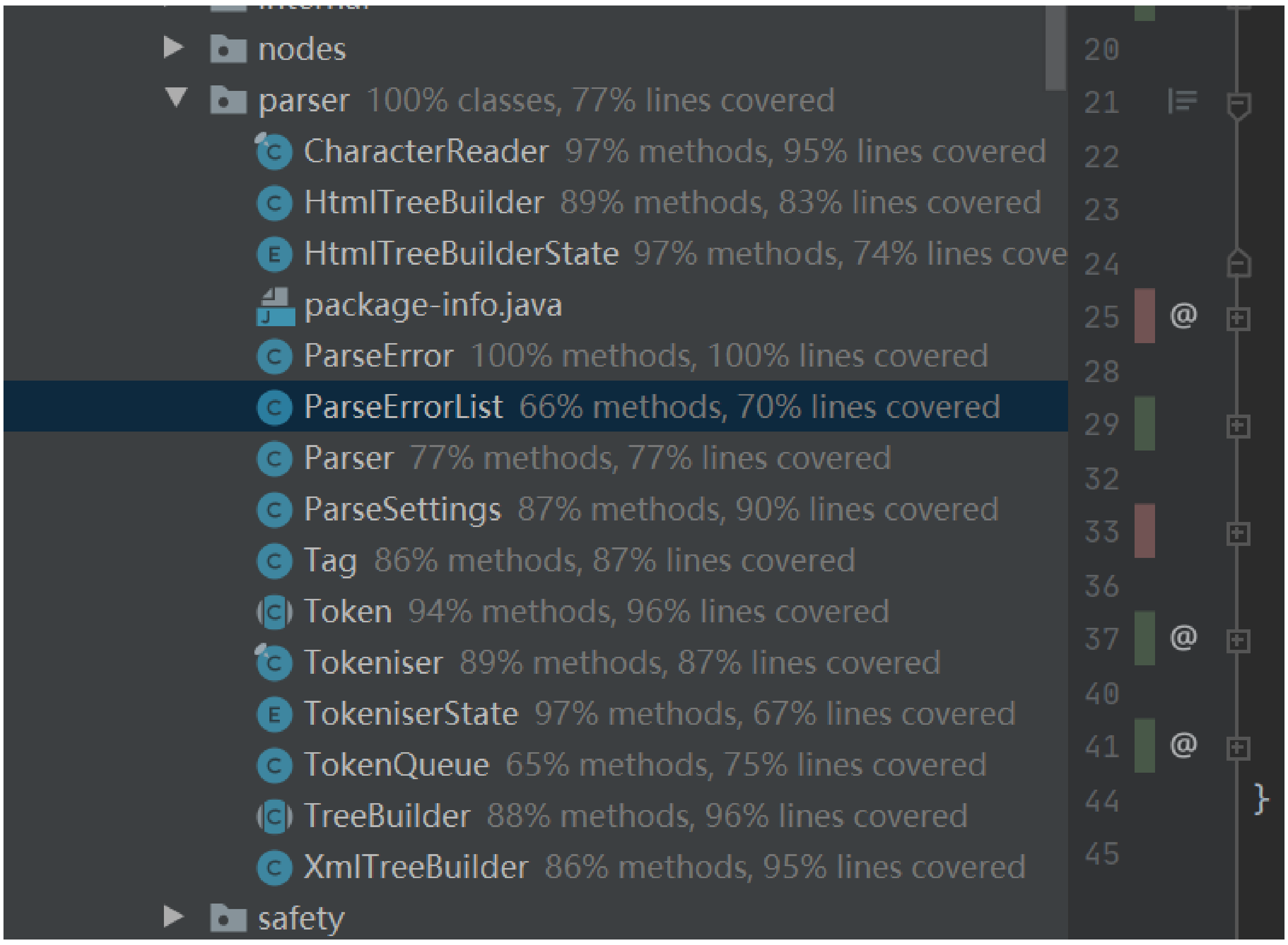
The meaning of the stars is very important, otherwise, you cannot test our project.

```
*   Match zero or more characters
**  Match zero or more directories
?   Match a single character
```

Coverage for JSoup

We want to focus on `org.jsoup.parser`, and its coverage is below, such as line, branch, and method coverage.

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



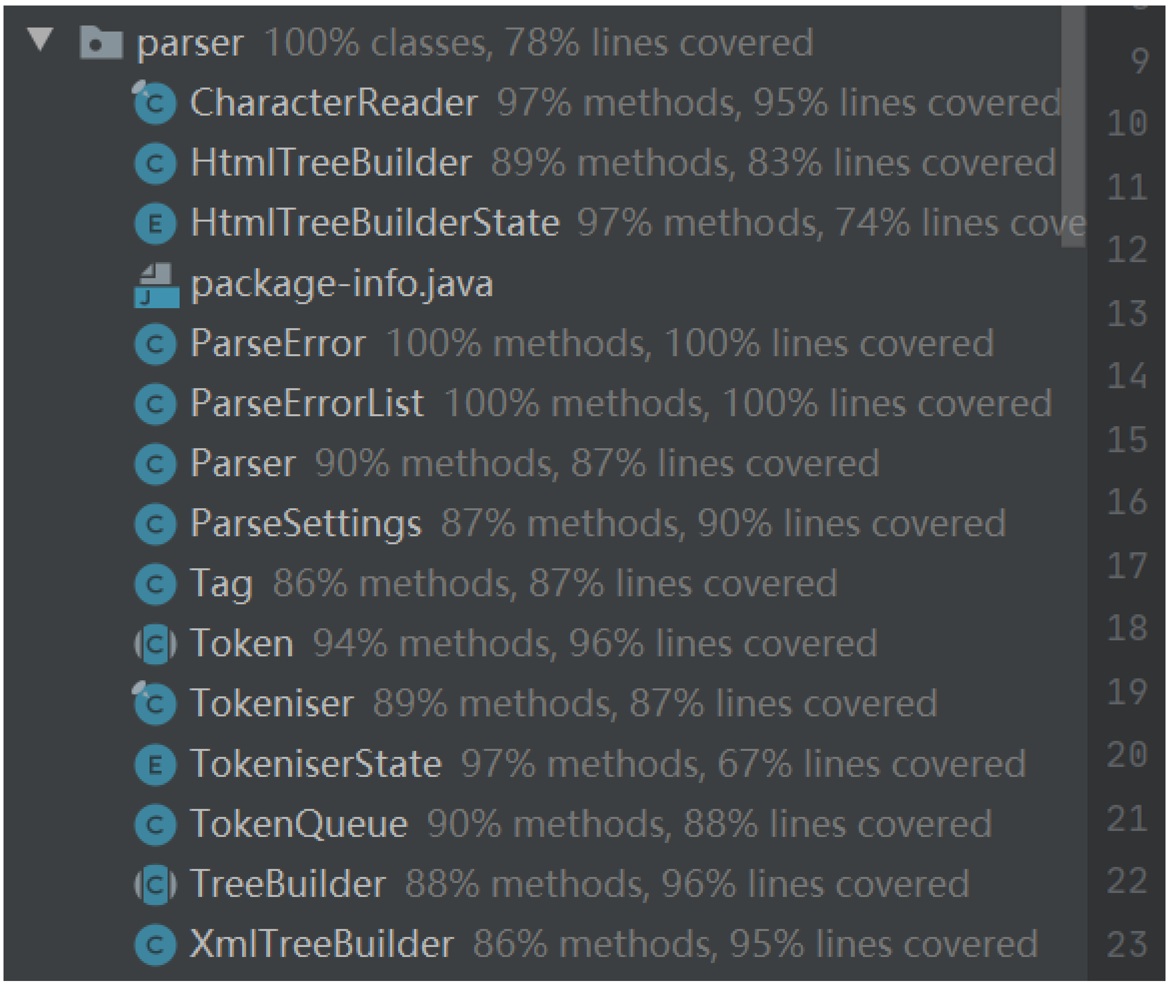
```
▶ nodes
▼ parser 100% classes, 77% lines covered
    CharacterReader 97% methods, 95% lines covered
    HtmlTreeBuilder 89% methods, 83% lines covered
    HtmlTreeBuilderState 97% methods, 74% lines cove
    package-info.java
    ParseError 100% methods, 100% lines covered
    ParseErrorList 66% methods, 70% lines covered
    Parser 77% methods, 77% lines covered
    ParseSettings 87% methods, 90% lines covered
    Tag 86% methods, 87% lines covered
    Token 94% methods, 96% lines covered
    Tokeniser 89% methods, 87% lines covered
    TokeniserState 97% methods, 67% lines covered
    TokenQueue 65% methods, 75% lines covered
    TreeBuilder 88% methods, 96% lines covered
    XmlTreeBuilder 86% methods, 95% lines covered
▶ safety
```

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.

New test case

Write new test cases to improve the coverage of the existing test suite in a meaningful way. Ideally, increase the coverage by at least 50 lines of code, or more. Document the coverage before and after, describe the code that you covered with your new test cases, and describe what functionality they test.

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.



```
parser 100% classes, 78% lines covered
    CharacterReader 97% methods, 95% lines covered
    HtmlTreeBuilder 89% methods, 83% lines covered
    HtmlTreeBuilderState 97% methods, 74% lines cove
    package-info.java
    ParseError 100% methods, 100% lines covered
    ParseErrorList 100% methods, 100% lines covered
    Parser 90% methods, 87% lines covered
    ParseSettings 87% methods, 90% lines covered
    Tag 86% methods, 87% lines covered
    Token 94% methods, 96% lines covered
    Tokeniser 89% methods, 87% lines covered
    TokeniserState 97% methods, 67% lines covered
    TokenQueue 90% methods, 88% lines covered
    TreeBuilder 88% methods, 96% lines covered
    XmlTreeBuilder 86% methods, 95% lines covered
```

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/

https://www.cnblogs.com/fnlingnzb-learner/p/10637802.html

Team Members

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

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

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