SWE 261P Project: Part 3. White Box Testing and Coverage

Wenjun Che, Yining Che, Joseph Young Lee
Group SilverLight

February 17, 2022

Abstract

In this part of the project, structural testing was performed on Jsoup. First, the existing test suite provided in Jsoup was analyzed to assess the overall structural coverage and also to determine which parts are not covered. Then, new tests cases were added to test the parts of the codes that were not thoroughly executed during testing. Performing this increased overall coverage of the structural testing by 105 lines of code. Newly added test cases can be found in the following Github link: Github

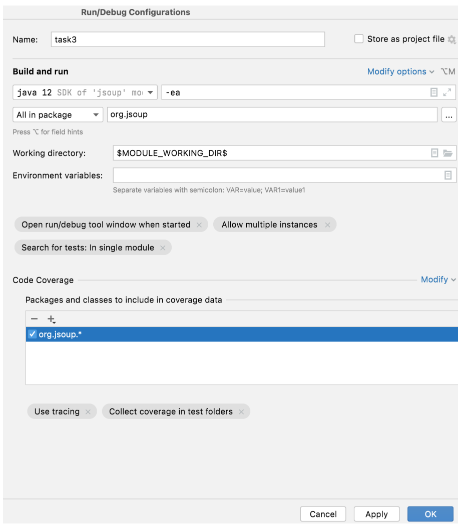
1 Introduction

Structural testing, also known as white-box testing, is a type of software testing where the tests are based on the software's internal structure, design, and code logic. Unlike functional testing, structural testing requires an in-depth knowledge of the code's internal implementation and control flows, and it complements the functional testing to provide an in-depth overall verification of the system. Structural testing, in general, is executed by the developers to verify all the decision branches, loops, methods, and statements in the code; and therefore, checks software usability, design, and security[2]. It's an important testing technique in terms of revealing errors that are "hidden" in the code, spotting unused codes, and identifying other issues with regards to best programming practices[3].

Jsoup, as an HTML parser[1], has numerous conditional statements in the code in order to handle various formats for Documents and Elements. Thus, performing structural testing on Jsoup will allow us to closely examine which part of the code is not being executed or has problems. For this reason, in this part of testing, structural testing will be applied to Jsoup. Specifically, coverage testing tools from JUnit will be used in order to assess the coverage of the lines, branches, and statements of existing test suites. Then, new test cases will be added to improve the overall coverage.

2 Coverage of the Existing Test Suite

Jsoup has an existing test package that verifies many of the core functionalities of the library. To confirm how much of the code is being covered from the existing test suite, a coverage testing was performed. We use the **Coverage Tool** in IntelliJ and Figure 1 the configuration. The results are in the Table 1.


Run/Debug Configurations

Name: task3 | Store as project file

Build and run

java 12 SDK of 'jsoup' mo | -ea

All in package | org.jsoup

Press ⌥ for field hints

Working directory: $MODULE_WORKING_DIR$

Environment variables:

Separate variables with semicolon: VAR=value; VAR1=value1

Open run/debug tool window when started | Allow multiple instances

Search for tests: In single module

Code Coverage

Packages and classes to include in coverage data

[✓ org.jsoup.*]

Use tracing | Collect coverage in test folders

Figure 1: Coverage Tool Configuration

Element	Class	Method	Line
org.jsoup	97%(240/246)	90%(1447/1603)	87%(6925/7884)

Table 1: Coverage from Existing Test Cases

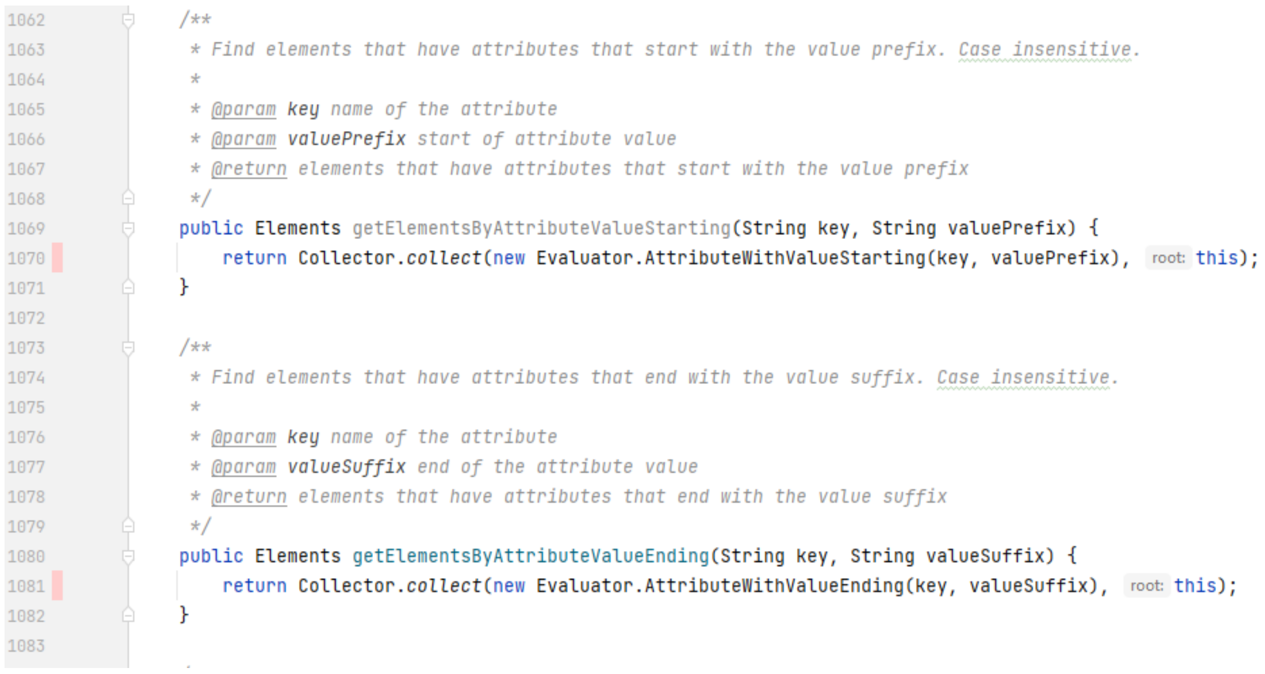
The test ran through existing 1190 test cases, which covered 97% classes and 87% lines of code. Overall, the existing test suite provided relatively high coverage throughout the program. However, as shown in the Table 2, the result also showed that the tests did not thoroughly cover some of the methods from the classes in two of the core packages that handles the nodes and the parser: **org.json.nodes** and **org.json.parser**. To improve the overall coverage of the two packages, additional structural testing will be performed. The following section highlights the testing targets.

Package	Class	Method	Line
org.jsoup	66.7%(4/6)	73%(27/37)	80.4% (41/51)
org.jsoup.examples	0%(0/4)	0%(0/15)	0%(0/91)
org.jsoup.helper	100%(12/12)	86%(178/207)	88.7%(886/999)
org.jsoup.internal	100%(5/5)	92.3%(36/39)	96.6%(171/177)
$*org.jsoup.nodes$	100%(30/30)	90.6%(395/436)	90.9%(1423/1566)
$*org.jsoup.parser$	100%(117/117)	95.4%(556/583)	86.1%(3317/3852)
org.jsoup.safety	100%(9/9)	100%(45/45)	95.1%(272/286)
org.jsoup.select	100%(63/63)	84.2%(219/260)	93.2%(815/874)

Table 2: Coverage Breakdown

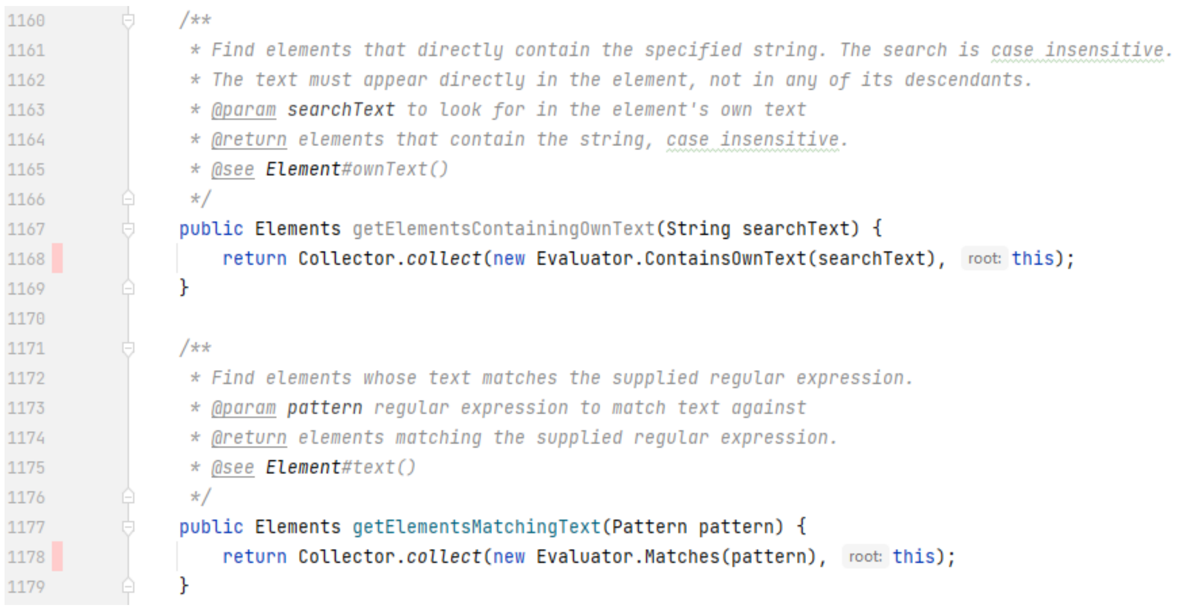
2.1 (json.org.node) Element

The Element class, in the org.json.node package, has some parts of the class methods that were not covered from the existing test suite. More specifically, methods that are used for selecting elements by attribute name, attribute value, and text content, did not get thoroughly tested. Refer to the example methods in the Figure 2 and Figure 3 below. These methods can be important especially when targeting attributes, attribute values, or text contents when using the Jsoup parser. Thus, additional test cases will be added to cover them.



```
/**
 * Find elements that have attributes that start with the value prefix. Case insensitive.
 *
 * @param key name of the attribute
 * @param valuePrefix start of attribute value
 * @return elements that have attributes that start with the value prefix
 */
public Elements getElementsByAttributeValueStarting(String key, String valuePrefix) {
    return Collector.collect(new Evaluator.AttributeWithValueStarting(key, valuePrefix), root: this);
}

/**
 * Find elements that have attributes that end with the value suffix. Case insensitive.
 *
 * @param key name of the attribute
 * @param valueSuffix end of the attribute value
 * @return elements that have attributes that end with the value suffix
 */
public Elements getElementsByAttributeValueEnding(String key, String valueSuffix) {
    return Collector.collect(new Evaluator.AttributeWithValueEnding(key, valueSuffix), root: this);
}
```

Figure 2: Element class methods for selecting elements by attribute values, which are not covered from existing test suite.

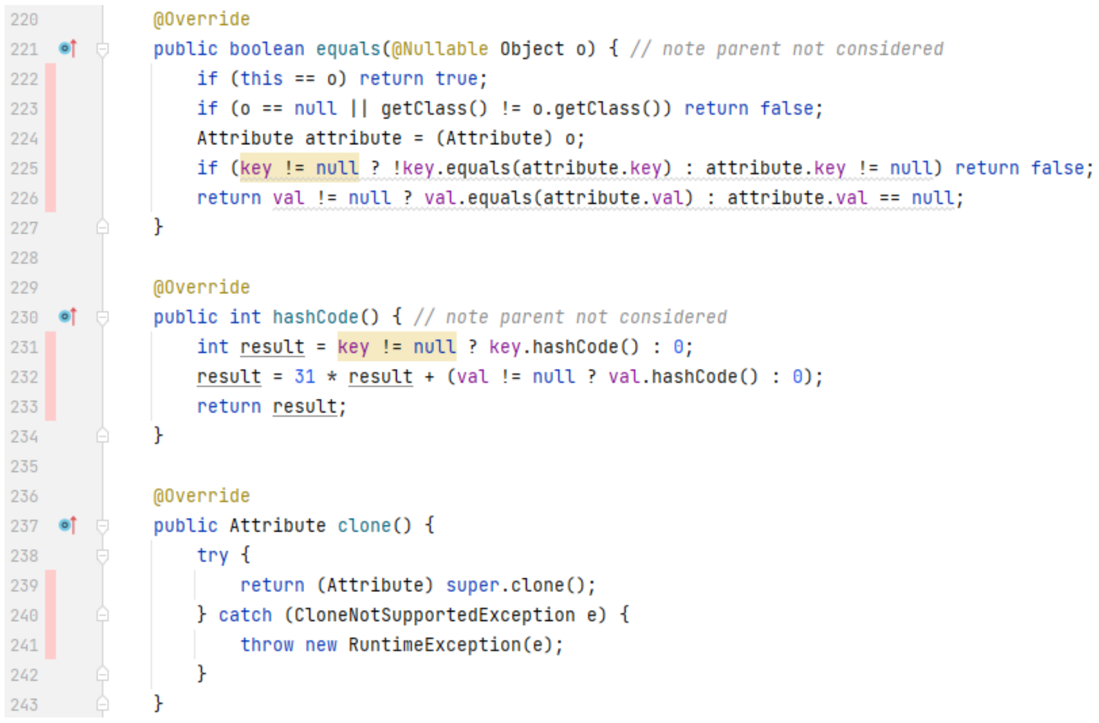


```java
/**
 * Find elements that directly contain the specified string. The search is case insensitive.
 * The text must appear directly in the element, not in any of its descendants.
 * @param searchText to look for in the element's own text
 * @return elements that contain the string, case insensitive.
 * @see Element#ownText()
 */
public Elements getElementsContainingOwnText(String searchText) {
    return Collector.collect(new Evaluator.ContainsOwnText(searchText),  root: this);
}

/**
 * Find elements whose text matches the supplied regular expression.
 * @param pattern regular expression to match text against
 * @return elements matching the supplied regular expression.
 * @see Element#text()
 */
public Elements getElementsMatchingText(Pattern pattern) {
    return Collector.collect(new Evaluator.Matches(pattern),  root: this);
}
```

Figure 3: Element class methods for selecting elements by containing texts, which are not covered from existing test suite.

2.2 (json.org.node) Attribute

Figure 4 shows methods from Attribute class, in the json.org.node package, which did not get covered from the testing. This class implements Cloneable and Map classes from the Java package. It also overrides methods from the base Object class in order to allow the Attribute class to check, produce hash codes for comparison, and clone the object as needed. While this should work as expected, additional testing coverage could give more guarantee that this executes properly.

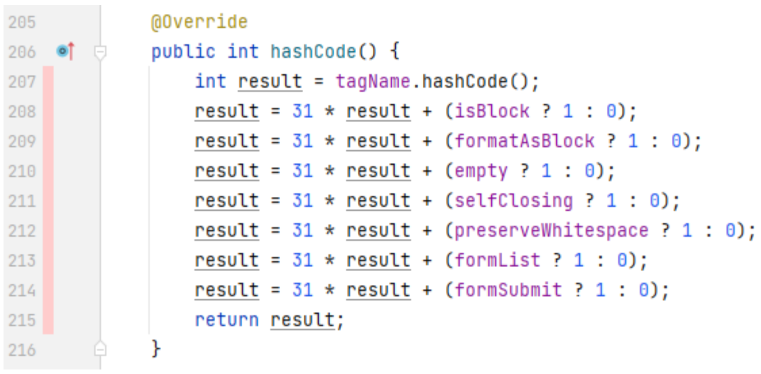


```java
@Override
public boolean equals(@Nullable Object o) { // note parent not considered
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    Attribute attribute = (Attribute) o;
    if (key != null ? !key.equals(attribute.key) : attribute.key != null) return false;
    return val != null ? val.equals(attribute.val) : attribute.val == null;
}

@Override
public int hashCode() { // note parent not considered
    int result = key != null ? key.hashCode() : 0;
    result = 31 * result + (val != null ? val.hashCode() : 0);
    return result;
}

@Override
public Attribute clone() {
    try {
        return (Attribute) super.clone();
    } catch (CloneNotSupportedException e) {
        throw new RuntimeException(e);
    }
}
```

Figure 4: Missed Functions in Attribute Class

2.3 (json.org.parser) Tag

The Tag class also has parts of the codes that were not covered by the testing, specifically the **hascode()** method shown in Figure 5. Similar to the Attribute class, this method is also inherited from base Java object class but it also adjusts the hash code depending on the fields value of an Tag object. It need a specific method to calculate the hash value of a Tag object and promise no hash collision.

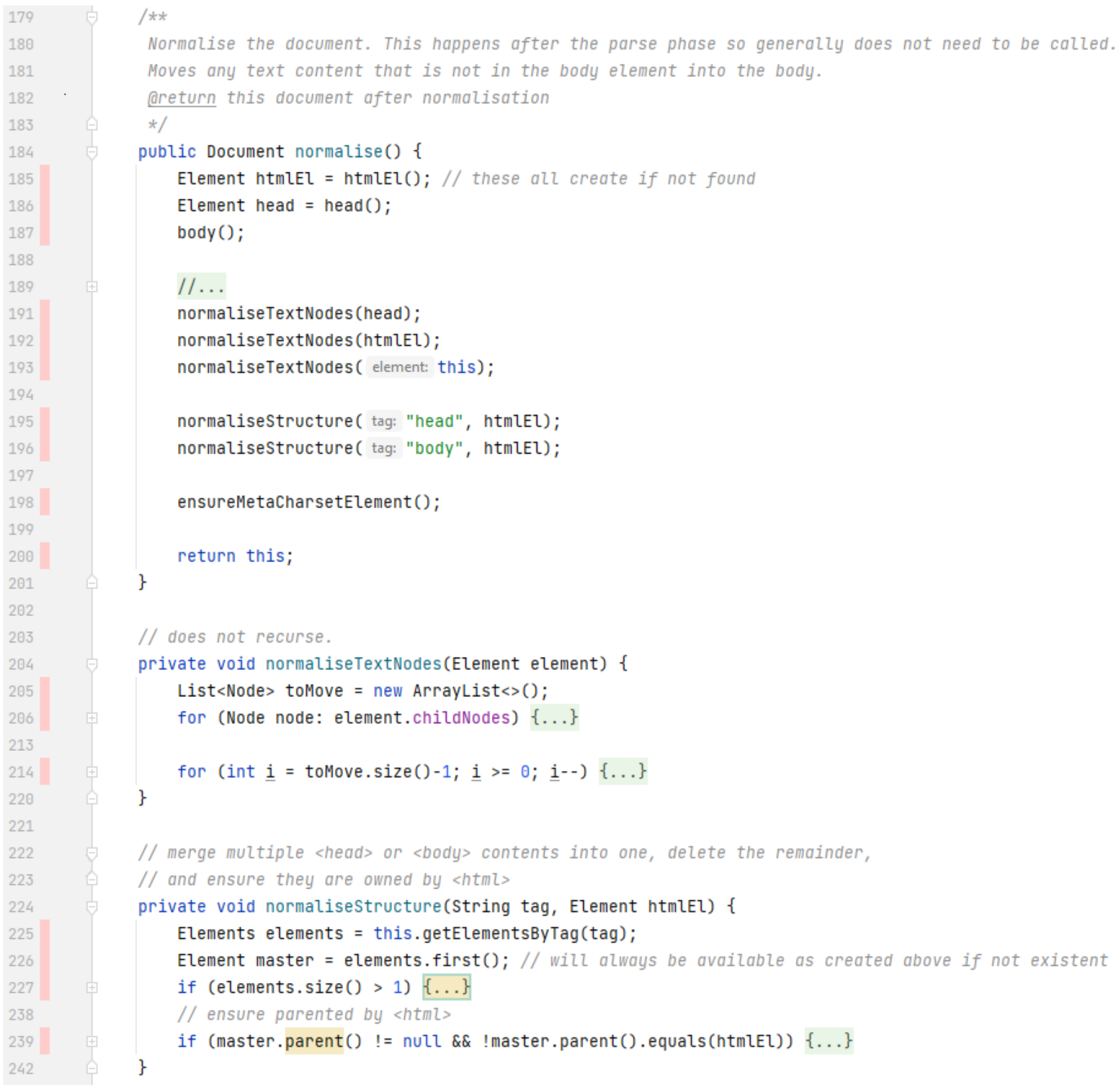


```java
@Override
public int hashCode() {
    int result = tagName.hashCode();
    result = 31 * result + (isBlock ? 1 : 0);
    result = 31 * result + (formatAsBlock ? 1 : 0);
    result = 31 * result + (empty ? 1 : 0);
    result = 31 * result + (selfClosing ? 1 : 0);
    result = 31 * result + (preserveWhitespace ? 1 : 0);
    result = 31 * result + (formList ? 1 : 0);
    result = 31 * result + (formSubmit ? 1 : 0);
    return result;
}
```

Figure 5: Missed Functions in Tag Class

2.4 (json.org.node) Document

The Document in json.org.node package also had methods that were not covered from testing, shown in Figure 6. The **Normalisation** of Document Object is to move any text content that is not in the body element of the HTML body. It calls two methods to normalise text nodes and structure.

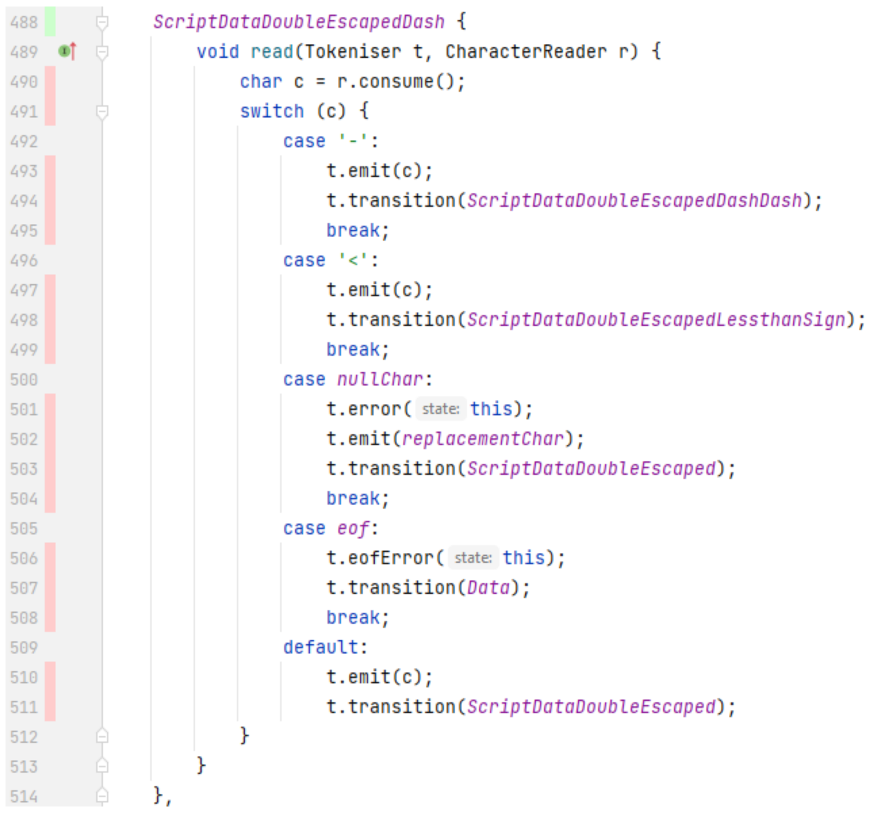


```java
/**
 Normalise the document. This happens after the parse phase so generally does not need to be called.
 Moves any text content that is not in the body element into the body.
 @return this document after normalisation
 */
public Document normalise() {
    Element htmlEl = htmlEl(); // these all create if not found
    Element head = head();
    body();

    //...
    normaliseTextNodes(head);
    normaliseTextNodes(htmlEl);
    normaliseTextNodes( element: this);

    normaliseStructure( tag: "head", htmlEl);
    normaliseStructure( tag: "body", htmlEl);

    ensureMetaCharsetElement();

    return this;
}

// does not recurse.
private void normaliseTextNodes(Element element) {
    List<Node> toMove = new ArrayList<>();
    for (Node node: element.childNodes) {...}

    for (int i = toMove.size()-1; i >= 0; i--) {...}
}

// merge multiple <head> or <body> contents into one, delete the remainder,
// and ensure they are owned by <html>
private void normaliseStructure(String tag, Element htmlEl) {
    Elements elements = this.getElementsByTag(tag);
    Element master = elements.first(); // will always be available as created above if not existent
    if (elements.size() > 1) {...}
    // ensure parented by <html>
    if (master.parent() != null && !master.parent().equals(htmlEl)) {...}
}
```

Figure 6: Missed Functions in Document Class

2.5 (json.org.parser) TokenizerState

In the org.json.parser package, the coverage from existing testing case for the TokenizerState class was relatively low. This class is an enum class where it defines state of the the tokenizer in order for the system to identify whether if a token is misshaped or whether if it has problems, as the system parses through a document. Because tokens could be misshaped in various ways when parsing through different HTML sources, there are several switch statements involved in this class, but not all cases were covered thoroughly from testing. For example, in Figure 7, states with unexpected escapes were not tested:



```
ScriptDataDoubleEscapedDash {
    void read(Tokeniser t, CharacterReader r) {
        char c = r.consume();
        switch (c) {
            case '-':
                t.emit(c);
                t.transition(ScriptDataDoubleEscapedDashDash);
                break;
            case '<':
                t.emit(c);
                t.transition(ScriptDataDoubleEscapedLessthanSign);
                break;
            case nullChar:
                t.error( state: this);
                t.emit(replacementChar);
                t.transition(ScriptDataDoubleEscaped);
                break;
            case eof:
                t.eofError( state: this);
                t.transition(Data);
                break;
            default:
                t.emit(c);
                t.transition(ScriptDataDoubleEscaped);
        }
    }
},
```

Figure 7: Parts of tokenizer states not covered from existing test suite.

To improve the coverage for the above parts of the code, additional test cases will be added:

3 Improvement of Test Coverage

Based on the finding in Section 2, new test cases are add in the Github link: Github. The overall coverage results after adding new test cases are shown in Table 3. We increase the coverage by **105** lines of code. Details are shown in the following sections.

3.1 (json.org.node) Element

For the Element class, three test methods have been added to provide addiditional testing coverage. These tests cover methods for selecting elements by attribute name, attribute values, and also text contents. For example, **elementsByAttributeValue** test covers Element class methods for selecting elements by attribute values.

```
/**
 *  Test for Element
 */
@Test void elementsByAttributeValue(){
    Document doc = Jsoup.parse(
            "<span class='class1'>Hello</span>" +
            "<span class='myclass2'>test</span>" +
            "<span style='padding:1px'></span>");

    Element span = doc.getElementsByAttributeValueContaining("class","myclass2").last();
    assertTrue(span.hasClass("myclass2"));

    span = doc.getElementsByAttributeValueMatching("class","class1").get(0);
    assertTrue(span.hasClass("class1"));

    span = doc.getElementsByAttributeValueEnding("style","px").get(0);
    assertTrue(span.hasAttr("style"));

    span = doc.getElementsByAttributeStarting("class").get(0);;
    assertTrue(span.hasClass("class1"));

    span = doc.getElementsByAttributeValueNot("class","class1").last();
    assertTrue(span.hasAttr("style"));
}
```

3.2 (json.org.node) Attribute

We add new test cases to **Attribute** which focus on the following methods:

- createFromEncoded():test the Attribute could be built from unencoded string like "/w"
- shouldCollapseAttribute(): test collapsible if it's a boolean attribute and value is empty or same as name
- equals():test if two Attribute Objects are equal
- clone():test if it could be cloned to another object
- hashCode():test how to get an hash value from an Attribute Object

```
/**
 * test for Attribute
 */
@Test
public void testCreateFromEncoded() {
    Attribute a = new Attribute("w","hello"); Attribute b =
        Attribute.createFromEncoded("/w","hello");
    Assertions.assertEquals(a.toString(),b.toString());

}
@Test
public void testCollapseAttribute() {
    Attribute a = new Attribute("w","hello");
    boolean iscoll = a.shouldCollapseAttribute(out.outputSettings());
    Assertions.assertFalse(iscoll);
}
@Test
public void testCollable() {
    Attribute a = new Attribute("w","hello");
    Attribute sameAtt = new Attribute(k,v);
    Attribute diffAtt = new Attribute("new",v);
    String t = "fs";
    Document out = new Document("www.baidu.com");
    Assertions.assertFalse(a.equals(t));
    Assertions.assertFalse(a.equals(null));
    Assertions.assertTrue(a.equals(sameAtt));
    Assertions.assertFalse(a.equals(diffAtt));
}
@Test
public void testHashCode() {
   Attribute a = new Attribute("w", "hello");
    Assertions.assertEquals(a.hashCode(),99166011);
}
@Test
public void testClone() {
    Attribute a = new Attribute("class","hello");
    Attribute same = new Attribute(k,v);
    Attribute cloneAtt = a.clone();
```

```
        Assertions.assertTrue(a.equals(cloneAtt));
    }
```

After adding the test cases, the improvement is shown in Table 5.

3.3 (json.org.node) Document

We add new test cases to Document class which are focus on following methods:

- normalise(): test moving any text content that is not in the body element into the body
- normaliseTextNodes(): test normalising the nodes
- normaliseStructure(): test normalising the structure

```
    @Test public void testDocumentEasyNor() {
    Document d = new Document("www.example.com");
    Document newD = d.normalise();
    Assertions.assertEquals("<html>\n" +
            " <head></head>\n" +
            " <body></body>\n" +
            "</html>", d.html());
    }
```

3.4 (json.org.parser) Tag

For Tag class, we mainly test the following methods which handle hashing and checking of the tag values:

- hashCode(): test how to hash an Tag Object based on its field values
- equals(): test if two Tag objects are same

```
    @Test public void testTagHash() {
        Tag p1 = Tag.valueOf("P");
        int tagHashValue = p1.hashCode();
        Assertions.assertEquals(tagHashValue,-1421590287);
    }
    @Test public void testTagEqual() {
        Tag p1 = Tag.valueOf("p");
        Tag p2 = p1;
        Boolean result = p1.equals(p2);
        assertEquals(p1, p2);
        Tag p3 = Tag.valueOf("a");
        Tag p4 = Tag.valueOf("a");
        assertTrue(p3.equals(p4));
    }
```

3.5 (json.org.parser) TokenizerState

For the Tokenizer State, two methods have been added to cover different tokenizer states when unexpected escape of script occurs in the HTML doc. For example, **handlesEscapeScript** test covers cases for tokenizer states for different situations that happen after escape of a script element.

```
    /***
     *  TokenizerState: Tokenizer states handling script element and escape
     */
    @Test public void handlesEscapedScript() {
        // Escape inside script
        Document doc = Jsoup.parse("<script><!-- one <script>-lah</script> --></script>");
        assertEquals("<!-- one <script>-lah</script> -->", doc.select("script").first().data());

        // EOF
        doc = Jsoup.parse("<script><!-- one <script>");
        assertEquals("<!-- one <script>", doc.select("script").first().data());
        // Double-escaped
        doc = Jsoup.parse("<script><!-- one <script>--lah</script> --></script>");
        assertEquals("<!-- one <script>--lah</script> -->", doc.select("script").first().data());

        // "<" after escape
        doc = Jsoup.parse("<script><!-- one <script>-<lah</script> --></script>");
        assertEquals("<!-- one <script>-<lah</script> -->", doc.select("script").first().data());

        // nullChar after escape
        doc = Jsoup.parse("<script><!-- one <script>-\u0000lah</script> --></script>");
        assertEquals("", doc.select("script").first().text());

        //Escaped and EOF
        doc = Jsoup.parse("<script><!-- one <script>-");
        assertEquals("", doc.select("script").first().text());
    }
```

4 Results and Conclusion

After the set of new tests described above have been added, the JUnit coverage testing was performed again using the same configuration. Total 16 new tests have been added and the coverage testing, which executed through 1,206 tests in total. Table 3 shows comparison of the overall structural coverage before and after the new tests have been added. The overall coverage of the testing has increased by 105 lines and 11 methods.

	Element	**Class, %**	**Method, %**	**Line, %**
After	org.jsoup	97.6% (240/246)	91.1% (1477/1622)	89% (7030/7896)
Before	org.jsoup	97.6% (240/246)	90% (1456/1622)	87.7% (6925/7896)

Table 3: Overall coverage Results After Adding New Cases

The following tables show detailed breakdown of improvements on the new coverage testing results. Only the packages that were discussed in the above sections have been improved.

	Class, %	**Method, %**	**Line, %**
Before	100% (3/3)	89.4% (127/142)	92.4% (439/475)
After	100% (3/3)	95.% (136/142)	95.6% (454/475)

Table 4: Coverage Comparison of Element Class

	Class, %	**Method, %**	**Line, %**
Before	100% (1/1)	78.3% (18/23)	78.7% (59/75)
After	100% (1/1)	100% (23/23)	94.7% (71/75)

Table 5: Coverage Comparison of Attribute Class

	Class, %	**Method, %**	**Line, %**
Before	100% (4/4)	94.4% (51/54)	78.9% (150/190)
After	100% (4/4)	100% (54/54)	88.9% (169/190)

Table 6: Coverage Comparison of Document Class

	Class, %	**Method, %**	**Line, %**
Before	100% (1/1)	90.9% (20/22)	88% (88/100)
After	100% (1/1)	100% (22/22)	98% (98/100)

Table 7: Coverage Comparison of Tag Class

	Class, %	**Method, %**	**Line, %**
Before	100% (68/68)	98.6% (140/142)	75.3% (816/1083)
After	100% (68/68)	100% (142/142))	79.9% (865/1083)

Table 8: Coverage Comparison of TokenizerState Class

In a nutshell, structural testing is an effective and essential technique to improve test suite coverage and, therefore, the software system's overall performance. Even though Jsoup is a mature software application with a robust testing suite, we still improved test coverage. Especially for the Attribute, Document, Tag, and TokenizerState classes, method coverage significantly increased to 100%. As a parser library, Jsoup involves many branches within its control flow, allowing it to handle unexpected and unique cases effectively. Many improvements were from covering through conditional branches that were not thoroughly executed. Even though structure testing itself cannot guarantee the absence of errors and bugs in the program, it can provide higher confidence when verifying the system when used in tandem with functional testing and other testing techniques. The newly added testing results can continue to be used in the future as automated testing to allow additional verification.

References

[1] jsoup: Java html parser, built for html editing, cleaning, scraping, and xss safety. `https://jsoup.org/`.

[2] Structural testing. `https://www.tutorialspoint.com/software_testing_dictionary/structural_testing.htm`.

[3] White-box testing - wikipedia. `https://en.wikipedia.org/wiki/White-box_testing`.