SWE 261P Project: Part 2. Functional Testing and Finite State Machines

Wenjun Che, Yining Che, Joseph Young Lee
Group SilverLight

February 10, 2022

Abstract

In this project, a well-known black-box software testing technique, functional modeling, was applied to verify a feature in Jsoup. More specifically, the functionality to parse, search, and modify CSS class attributes in a targeted element has been analyzed and modeled into a finite state machine model. The model provides an abstraction of the functionality for efficient testing. A set of parameterized JUnit testing was programmed and performed to cover the model, but no errors were discovered. These test cases can continue to be used in the future to provide additional verification of the system's functionality. Newly added test cases can be found in the following Github link: Github

1 Introduction

In black-box testing, **functional models** are often defined to simplify the complex functionalities of software into abstract models. Such abstraction represents software in a more compact form, making it easier to understand and test. Using the functional models, testing can be simplified to testing just the important characteristics for particular features, which allows us to easily predict the expected output, and check whether the system is working properly.

One of the most common functional modeling approaches is the **finite-state machine** model. The finite state machine is an abstract representation of the system which can be defined by a set of finite states and a set of transitions that alters the state. A finite-state machine would prove especially useful for testing Jsoup; because a finite-state machine is useful for testing any processing where the output is based on the occurrence of one or more sequences of events, where the output is based on the occurrence of one or more sequences of events, such as detection of specified input sequences, sequential format validation, and parsing[1]. This is also the case for Jsoup because it is commonly used to parse through HTML contents, performing a sequence of actions to allow users to retrieve or modify contents and/or data from a webpage.

By defining a finite-state machine, a sequence of events can be effectively be narrowed down, which allows tests to focus on specific features and components. Thus, in this project, we will be applying the above techniques to test Jsoup.

2 Test Plan

Jsoup's main function is to parse the HTML files. Using Jsoup, the inputted HTML source gets parsed into a Document object, and then, into a set of Elements. Then, it's up to the developer to decide how to handle the parsed webpage contents.

2.1 Target Component/Feature

A HTML webpage is basically a structured hierarchical tree of elements, which eventually gets rendered into a webpage. Figure 1 shows the Document Object Model (DOM) that HTML follows:

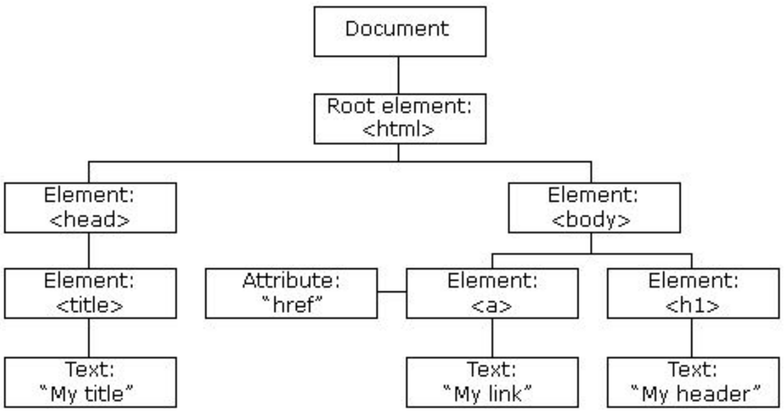

Figure 1: Document Object Model (DOM) in HTML[4]

Element nodes, in essence, make up the contents of a webpage, and one of the common uses of Jsoup is to retrieve and modify these elements. To verify this functionality, the tests will be performed around the Element class, which is inherited from the base Node class, within Jsoup. They are defined in the following files:

- src/main/java/org/jsoup/nodes/Element.java
- src/main/java/org/jsoup/nodes/Node.java

And we will focus on testing the functionality to parse, search, and modify the CSS class attribute of elements from a given document.

3 Finite State Machine

To model a finite state machine, a system state must be defined. In this model, we will define the states according to the CSS class attribute of an element. In an element, we can either add a specific class, remove a specific class, or remove the class attribute entirely. Testing with two classes, following four states have been defined:

- <div></div>
- <div class="class1"><div>
- <div class="class2"><div>
- <div class="class1 class2"><div>

Transition between each state is triggered by calling the methods that modify the CSS attributes and classes within a given element. The following methods will be called on a parsed element to trigger the transitions between the states:

- Add class1: **Element.addClass("class1"))**
- Add class2: **Element.addClass("class2"))**
- Remove class1: **Element.removeClass("class1")**
- Remove all classes: **Node.removeAttr("class");**

3.1 Model

The Figure 2 finite-state machine model illustrates all of the five state transitions that could occur on each of the four state nodes. Now that the finite-state machine has been defined, test cases to cover all of the transition edges have been programmed and added to the testing package.

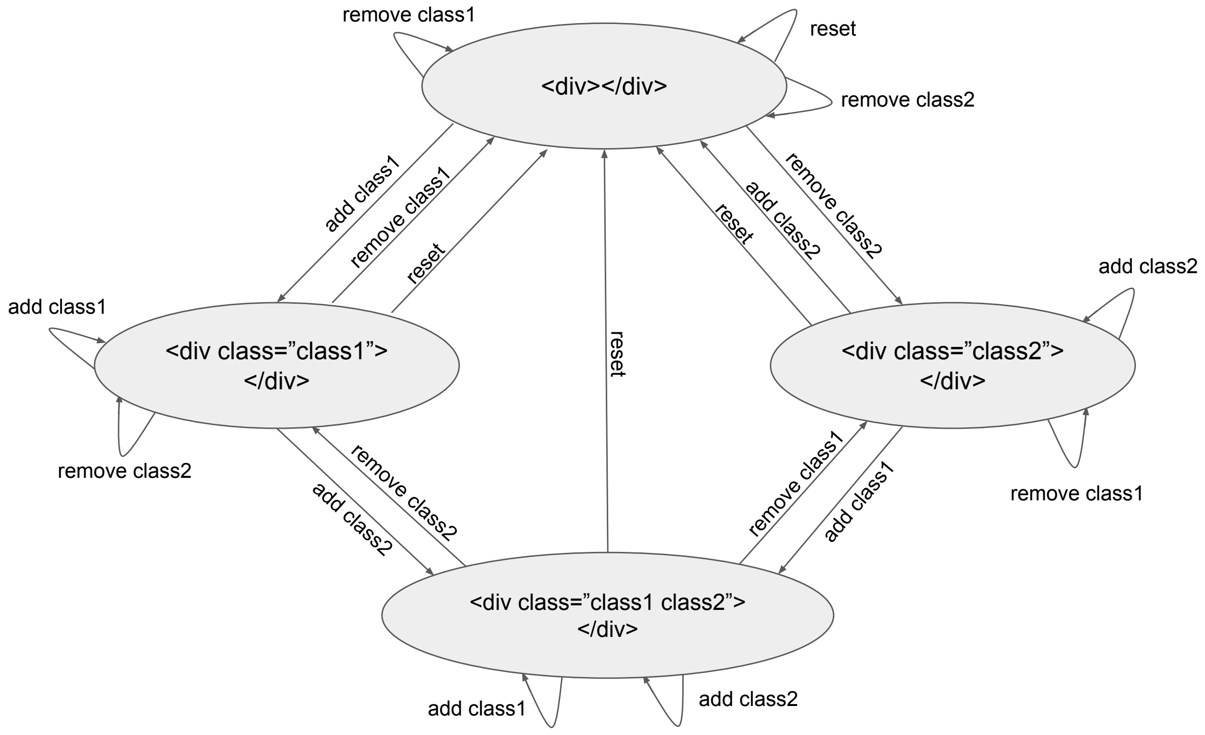

Figure 2: Finite-State Machine Graph

4 Test Cases

The JUnit testing cases, covering all of the transition edges, have been programmed into the following file. This file can also be found in the following Github repository: https://github.com/Christina0114/261SoftwareTesing/blob/master/src/test/java/org/jsoup/nodes/FSMTest.java:

- src/test/java/org/jsoup/nodes/FSMTest.java

The FSMTest file includes the following JUnit tests:

- testAddClass(String html, String tag)
- testAddClass2(String html, String tag)
- testRemoveClass(String html, String tag)
- testRemoveClass2(String html, String tag)
- testResetClass(String html, String tag)

In each method, parameterized JUnit testing was performed using the following sequence of String inputs which represent 5 common elements, and their tags, that often use the class attribute for loading a batch of CSS styles.

- <div></div>,div
- <p></p>,p
- <h></h>,h
- <title></title>,title
- <footer></footer>,footer

4.1 testAddClass / testAddClass2

testAddClass methods test the state changes when new classes gets added to an element. These two methods cover all of the add class transitions, starting from the to <div><div> down to the bottom node <div class="class1 class2"><div>, including the self edges while checking the expected system states.

```
@ParameterizedTest
   @CsvSource({"<div></div>, div", "<p></p>, p", "<h></h>, h", "<title></title>, title",
       "<footer></footer>, footer"})
   void testAddClass(String html, String tag){
       Document doc = Jsoup.parse(html);
       Element div = doc.select(tag).first();

       div.addClass("class1");
       assertEquals(div.toString(),"<" + tag + " class=\"class1\"></" + tag + ">");

       div.addClass("class1"); // self-edge
       assertEquals(div.toString(),"<" + tag + " class=\"class1\"></" + tag + ">");

       div.addClass("class2");
       assertEquals(div.toString(),"<" + tag + " class=\"class1 class2\"></" + tag + ">");

       div.addClass("class1"); // self-edge
       assertEquals(div.toString(),"<" + tag + " class=\"class1 class2\"></" + tag + ">");

       div.addClass("class2"); // self-edge
       assertEquals(div.toString(),"<" + tag + " class=\"class1 class2\"></" + tag + ">");

       div.removeAttr("class");
       assertEquals(div.toString(),"<" + tag + "></" + tag + ">");
   }
```

4.2 testRemoveClass / testRemoveClass2

testRemoveClass methods test the state changes when a class gets removed from an element. These two methods also thoroughly cover all of the remove class transitions, starting from the to <div class="class1 class2"><div> down to the bottom node $< div >< div >$, including the self edges while checking the expected system states.

```
   @ParameterizedTest
   @CsvSource({"<div></div>, div", "<p></p>, p", "<h></h>, h", "<title></title>, title",
       "<footer></footer>, footer"})
   void testRemoveClass2(String html, String tag){
       Document doc = Jsoup.parse(html);
       Element div = doc.select(tag).first();

       // start with 2 original classes
       div.addClass("class1");
       div.addClass("class2");
       assertEquals(div.toString(),"<" + tag + " class=\"class1 class2\"></" + tag + ">");

       div.removeClass("class1");
       assertEquals(div.toString(),"<" + tag + " class=\"class2\"></" + tag + ">");

       div.removeClass("class1"); // self-edge
       assertEquals(div.toString(),"<" + tag + " class=\"class2\"></" + tag + ">");

       div.removeClass("class2");
       assertEquals(div.toString(),"<" + tag + "></" + tag + ">");

       div.removeClass("class1"); // self-edge
       assertEquals(div.toString(),"<" + tag + "></" + tag + ">");

       div.removeClass("class2"); // self-edge
       assertEquals(div.toString(),"<" + tag + "></" + tag + ">");
   }
```

4.3 testResetClass

testRemoveClass methods tests the state changes back to $< div >< div >$ node by calling the Node.removeAttr("class") from each different states. These two methods thoroughly cover all of the reset class transitions in the model. The reset transition edges from each node, including the self edges on the $< div >< div >$, were covered; while checking the expected system states.

```java
@ParameterizedTest
@CsvSource({"<div></div>, div", "<p></p>, p", "<h></h>, h", "<title></title>, title",
    "<footer></footer>, footer"})
void testResetClass(String html, String tag){
    Document doc = Jsoup.parse(html);
    Element div = doc.select(tag).first();

    // reset the no class to empty
    div.removeAttr("class"); // self-edge
    assertEquals(div.toString(),"<" + tag + "></" + tag + ">");

    // reset the class1 to empty
    div.addClass("class1");
    assertEquals(div.toString(),"<" + tag + " class=\"class1\"></" + tag + ">");
    div.removeAttr("class");
    assertEquals(div.toString(),"<" + tag + "></" + tag + ">");

    // reset the class2 to empty
    div.addClass("class2");
    assertEquals(div.toString(),"<" + tag + " class=\"class2\"></" + tag + ">");
    div.removeAttr("class");
    assertEquals(div.toString(),"<" + tag + "></" + tag + ">");

    // reset the class1 and class2 to empty
    div.addClass("class1");
    div.addClass("class2");
    assertEquals(div.toString(),"<" + tag + " class=\"class1 class2\"></" + tag + ">");
    div.removeAttr("class");
    assertEquals(div.toString(),"<" + tag + "></" + tag + ">");
}
```

4.4 Running the tests

To run all the tests included in the package, we can also the Maven test command to run all JUnit tests in the project:

```
mvn test
```

To run just the FSMTest:

```
mvn test -D test=org.jsoup.nodes.FSMTest
```

Or the above test command can also be triggered through Maven tools supported in IDEs such as IntelliJ or Eclipse.

5 Conclusion

No errors were discovered from the testing. These tests can continuously be used in future developments to provide additional verifications to ensure Jsoup's functionality to parse, search, and modify element's class attributes according to the defined finite state machine model.

References

[1] Boris Beizer. Software testing techniques (second edition).

[2] James A. Jones. Swe261p-finite functional models. https://canvas.eee.uci.edu/courses/43617/files/folder/Lecture%20Slides?preview=17436045.

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

[4] JavaScript DOM Navigation. Javascript dom navigation. https://www.w3schools.com/js/js_htmldom_navigation.asp.