SWE 261P Project: Part 5. Testable Design Mocking

Wenjun Che, Ying Che, Joseph Young Lee
Group SilverLight

March 3, 2022

Abstract

This part of the project is divided into two subsections; testable design and mocking. First section discusses testable software design and assess Jsoup accordingly. Then the latter section discusses benefits of mock testing and applies the technique to Jsoup project. For mock testing, a newly added test file can be found in the following Github link: GitHub

1 Testable Design

1.1 Introduction to Testable Design

The testable design suggests software programming practices that improves the capability to test code. According to what Roy Osherove stated in his book[3], "A given piece of code needs to be easy and brief to write a unit test in opposition to." it is clear that the testable design simplifies the testing class instances, implementation substitutions, and scenarios simulations.By applying testable design into consideration, software can be better tested later on in the development cycles.

The following list are some of the suggested rules for testable design [1].

- Avoid complex private methods: private methods only can be accessed from methods inside the class and can't be tested outside of their class. If there is a bug in private methods containing complex logic, it may be difficult to narrow down.
- Avoid static Methods: They should be simple and small. If they share states and change other variables, it may be hard to test.
- Avoid hard-coded instances: Hard-coded objects can't be stubbed.
- Avoid logic in constructors: The constructor of a class triggers that of its parent class.
- Avoid singleton pattern: Singleton classes cannot be mocked nor stubbed for testing.

With the above design rules in mind, we examined through the Jsoup project, and checked for any codes that does not obey these rules, and therefore,difficult to test.

1.2 Implementation

1.2.1 Problem State

In the **Ducument** class, there is a method called **htmlEl**, which is programmed for finding the root HTML element in List 5. The code is shown in below.

Listing 1: a private method in Document Class

```
1    private Element htmlEl() {
2        for (Element el: childElementsList()) {
3            if (el.normalName().equals("html"))
4                return el;
5        }
6        return appendElement("html");
7    }
```

The method is used in other three methods from the Element:

- **public Element head()**: Get the header of the Document
- **public Element body()**: Get the main body of the Document

- **public Document normalise()**: Normalise the document by moving any text content that is not in the body element into the body.

Since this is a private method, it's difficult to test this functionality of this method through unit testing, and stubbing and mocking techniques cannot be applied neither.

1.2.2 Code Rewrite

This method is used in other places throughout the document class also. When this private function gets introduced with a bug, it could be difficult to narrow down with unit tests. Especially for **normalise** function which is complex and hard to find the bugs. Since there was not critical need for this method to be private, We should set the method to public so that this functionality of document class becomes easier to test.

Listing 2: a private method in Document Class

```
public Element htmlElNew() {
    for (Element el: childElementsList()) {
        if (el.normalName().equals("html"))
            return el;
    }
    return appendElement("html");
}
```

In order not to break the existing code, we build another method named **htmlElNew()** different from the original method called **htmlEl()**. Based on the new method, we will write test cases on the new method.

1.2.3 Code Testing

To test the functionality updated codes, two test cases were added. Code snippet of which is shown the below Listing 3. Two cases were tested: getting the root from an HTML string, and an empty HTML string given an base URL. The results match our exceptions.

Listing 3: Test Cases

```
@Test
public void testHtml() throws IOException {
    String input =
            "<html>"
                    +   "<head>"
                    +     "<meta http-equiv=\"content-type\" content=\"text/html; charset=Shift_JIS\"
                        />"
                    +   "</head>"
                    +   "<body>"
                    +     "before after"
                    +   "</body>"
                    + "</html>";
    InputStream is = new ByteArrayInputStream(input.getBytes(StandardCharsets.US_ASCII));
    Document doc = Jsoup.parse(is, null, "http://example.com");
    Element root = doc.htmlElNew();
    String expectedString = "<html>\n" +
            " <head>\n" +
            "  <meta http-equiv=\"content-type\" content=\"text/html; charset=Shift_JIS\">\n" +
            " </head>\n" +
            " <body>\n" +
            "  before after\n" +
            " </body>\n" +
            "</html>";
    Assertions.assertEquals(root.toString(),expectedString);
}
@Test
public void testEmptyHtml() throws IOException {
    String input = "";
    InputStream is = new ByteArrayInputStream(input.getBytes(StandardCharsets.US_ASCII));
    Document doc = Jsoup.parse(is, null, "http://example.com");
    Element root = doc.htmlElNew();
```

```
        String expectedString = "<html>\n" +
                " <head></head>\n" +
                " <body></body>\n" +
                "</html>";
        Assertions.assertEquals(root.toString(),expectedString);
    }
```

2 Mocking

2.1 Introduction

In unit testing, mocking is a process to reduce test flakiness and improve test coverage by replacing the actual dependencies with easy implementations[4]. It provide developers tools to program unit tests that tests interactions with other modules and components. In this process, the tester will replace the dependencies with a simple 'mocked' objects which can simulate the behaviors. Then the tester can have a clear view of the interactions among different objects instead of the state transformation.

Mock testing framworks, including Mockito, provide tools to perform mock tests quickly and effectively. These tools allow to mock class instances and also provide useful tools for monitoring the interations to simulate the dependencies In this way, the scope of the unit test could be limited. The fake object, "mock object," has the same methods and fields as the object class they mimic to simulate dependencies.

Mocking frameworks usually support "Callback" and offer many APIs for generating new mock objects, checking interactions, and defining return values. Mock tests can provide the following[1]:

- Auto-generation of mock objects that implement a given interface
- Logging of what calls are performed on the mock objects
- Methods/primitives for declaring and asserting your expectations

2.2 Implementation

For Jsoup, we will be using mock testing technique to test and verify the interactions between Element class and its content. The Tag and Attribute classes were mocked and used to interact with the Element class. This allows the test cases to focus on the interactions instead of the status of the attributes of tag class instances. This is especially helpful for Jsoup project as there can be various forms of attributes and tags in an HTML document. Mockito package version 4.3.1 was added to the project as a dependency using Maven, and the test cases were added to **src/test/java/org/jsoup/SWE261/MockTest.java**.

2.2.1 Mock class Setup

Listing 4: Setting up mock instances of Tag and Attributes class

```
    @Before
    public void setup() {
        tag = mock(Tag.class);
        attributes = mock(Attributes.class);
        ele = new Element(tag, "", atts);
        MockitoAnnotations.openMocks(this);
    }
```

2.2.2 Test Cases

The following test cases verify the interaction of an Element with mocked Tag and Attribute instances.Specifically, it tests getting the tag and attributes.

Listing 5: a private method in Document Class

```
    @Test
    public void checkTagObject() {
        Assertions.assertTrue(tag instanceof Tag);
        Assertions.assertTrue(tag instanceof Cloneable);
    }

```

```
    @Test
    public void getElementTest() {
        when(tag.getName()).thenReturn("p");
        System.out.println(ele.tagName());
        verify(tag, times(1)).getName();
        System.out.println(ele.isBlock());
        verify(tag, times(1)).isBlock();
        System.out.println(ele.cssSelector());
        verify(tag, times(2)).getName();
    }

    @Test
    public void normalTagTest() {
        System.out.println(ele.normalName());
        verify(tag, times(1)).normalName();
    }

    @Test
    public void textTest() {
        System.out.println(ele.text());
        verify(tag, times(1)).isBlock();
    }
```

3 Results and Conclusion

In this project, we discussed testable design and analyzed Jsoup accordingly. Software systems that follow such design is well divided and easy to be tested. A method inside Document class was updated and two test cases were added. Then, mock testing techniques were applied in order to test the interactions between Elements, Attributes, and Tags classes. The tests added can be used in the future to provide addition confidence to the system verification.

References

[1] James A. Jones. Integration testing, mocking, testable design. https://canvas.eee.uci.edu/courses/43617/files/folder/Lecture%20Slides?preview=17765481.

[2] Lasse Koskela. *Effective unit testing: A guide for Java developers*. Simon and Schuster, 2013.

[3] Roy Osherove. *The Art of Unit Testing: with examples in C*. Simon and Schuster, 2013.

[4] Hengcheng Zhu, Lili Wei, Ming Wen, Yepang Liu, Shing-Chi Cheung, Qin Sheng, and Cui Zhou. Mocksniffer: Characterizing and recommending mocking decisions for unit tests. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 436–447, 2020.