Part V: Testable Design

Introduction to Testable Design

What is testable design? The basic value proposition of testable design is to be able to test code better. As Roy Osherove said[1], testable design is “a given piece of code should be easy and quick to write a unit test against.”

Guidelines for Testable Design

After talking about the concept of testable design, you might still have question on how to make a testable design. There are some rules to follow as below.

- In testable design, we need to avoid complex `private` methods. The reason for this rule is quite clear. In Java, we cannot test with `private` methods. Under this circumstance, when there are some complex `private` methods, we cannot test with them.
- In testable design, we need to avoid `static` methods. The reason for this rule is also Java's definitions. In Java, `static` method operate on the class instead of the object.
- In testable design, we need to be careful to hardcode in `new`. If so, object cannot be stubbed.
- In testable design, we need to avoid logic in constructors. It's difficult to bypass a constructor because subclass constructors always trigger at least one superclass constructor.
- In testable design, we need to avoid singleton pattern.

JSoup testable design

In the `JSoup` functions, in the folder `src/main/java/org.jsoup/nodes/Element`, we have this function called `childElements`. As we talked about in the Guidelines for Testable Design section, we try to avoid complex `private` method because we cannot test with them. In this function, we also have this function is private so it cannot be tested.

Original Code is as following.

```
    List<Element> childElements() {
        return children;
    }
```

To change it into a testable design, we set it into `public`, and the new function name is set to ``.

To write new test case for our testable design, we put that into `src/test/java/org.jsoup/swe261`. The test function is called `TestableDesign.java`

```
@Test
    public void childListTest(){
        String html = ""; // To see the original html string,
please refer to our code
        Document doc = Jsoup.parse(html);
        Element ele = doc.body();
        System.out.println(ele.childElementsV2().get(0));
        System.out.println(ele.childElementsV2().get(1));
        String exp = "<p>First post! <img src=\"foo.png\"></p>";
        String exp2 = "<p>Second post! <img src=\"foo2.png\"></p>";
        assertEquals(2,ele.childElementsV2().size());
        assertEquals(exp,ele.childElementsV2().get(0).toString());
        assertEquals(exp2,ele.childElementsV2().get(1).toString());
    }
```

Introduction to Mocking

What is mocking? To understand mocking, we firstly need to understand the word "**mock**" - A fake object that decides whether a unit test has passed or failed by watching interactions between objects.

Importance of Mocking

After talking about the basic concepts of mocking, we come to the question why we need mocking. Combined with its concept, we concluded four reasons of necessity of mocking.

- Mocking process can simulate external dependencies. Without mocking, if a test case fails, we don't know whether the failure is because of our code unit or because of our code dependencies.
- Mocking process can promote the interaction between objects.

- During development, mocking can help developers start testing early because mocking also support demos and evaluations. All units of the project can be carried out in parallel without having to wait for everyone to be ready.
- Mocking can help us avoid repeating test code in similar tests.

Mock JSoup Now!

Mock with function normaliseDocumentNodes

The feature we chose to mock JSoup is `normaliseDocumentNodes`. This function's original code is as follows.

```
public Element normaliseDocumentNodes() {
        //Element htmlEl = htmlEl(); // these all create if not found htmlNode

        Element head = head2();
        body2();
        // pull text nodes out of root, html, and head els, and push into body. non-text nodes are already taken care
        // of. do in inverse order to maintain text order.
        normaliseTextNodes2(head);
        normaliseTextNodes2(ele.htmlel());
        normaliseTextNodes2(this);

        ensureMetaCharsetElement();

        return this.ele.htmlel();
    }
```

But, wait. Why we need mocking to test with this function, not other methods?

As we mentioned in the "Importance of Mocking" section, the third reason is that mocking can help developers start testing early. To test with this function, we need `element`, which is external dependency. What are we testing? The interaction between `element` and `document`. Theoretically, if we want to test with this function, we need to test with element first. If element is invalid, we cannot test with this function. But, using mocking, we can test with this function easily because we already know the output of element!

Test with function normaliseDocumentNodes now

The feature we chose to mock JSoup is `normaliseDocumentNodes`.

Firstly, we set up this test and use `mock` function here.

```
@Before
    public void setup() {
        ele = mock(Element.class);
        doc = new Document("<html></html",ele);
        MockitoAnnotations.initMocks(this);
    }
```

Then we get to test this function by mockito. We used the function

- `when` is to set the return value
- `assert` is to judge if the result is equal
- `verify` is to verify how many times the function runed

The code is as following

```
@Test
    public void mockitoTest1(){
        String exp = "";//To see the original html string, please
refer to our code
        when(ele.childElementsList()).thenReturn(getEleList());
        when(ele.htmlel()).thenReturn(getEle());
        System.out.println(doc.normaliseDocumentNodes());

 assertThat(doc.normaliseDocumentNodes().toString()).isEqualTo(exp)
;
        verify(ele, times(4)).childElementsList();
        verify(ele, times(4)).htmlel();
    }
```

Reference

[1] https://livebook.manning.com/book/effective-unit-testing/chapter-7/8

[2] https://devopedia.org/mock-testing