SWE_261P_ Project_Part_5
Team Members:
· Yuxin Huang: yuxinh20@uci.edu
· Changhao Liu: liuc50@uci.edu
· Ruokun Xu: ruokunx@uci.edu
Team Name: OffersAreHere
Github link: https://github.com/yx-hh/commons-io

Introduction of Testable Design

Testable design is a modular design that avoids certain implementation styles which bring challenges to testers to write test cases. The goal of testable design is to make implementations easier to test(Davis, 2019).

Details of Testable Design

For testable design, we should

- Avoid complex private methods (Jones, page.17).
    - We cannot test with private methods, as we can hardly access them.
    - Private methods with complex logic may lead to some tricky bugs and are not able to be found by direct testing.

- Avoid static methods (Jones, page.18).
    - Static methods can only be called by the class instead of the object.
    - Static methods cannot be used flexibly, making functionality that has side effects or that has randomness difficult to test.

- Be careful of hardcoding in “new” (Jones, page.19).
    - We cannot stub objects if they are created using the “new” keyword.
    - Object reference can be retrieved using dependency injection instead.

- Avoid logic in constructors (Jones, page.20).
    - Constructors are hard to work around with, as a subclass's constructor always calls one of its superclass's constructors.

- Avoid Singleton Pattern (Jones, page.21).

Testable Design example

By testable design principles, we should avoid private methods with complex logic. In our project Commons-io, there is a method, named doGetFullPath (see Figure 1), which is under the path /src/main/java/org/apache/commons/io/FilenameUtils. It is a private method that cannot be tested.

Original code:

```java
private static String doGetFullPath(final String fileName, final boolean
includeSeparator) {
   if (fileName == null) {
       return null;
   }
   final int prefix = getPrefixLength(fileName);
   if (prefix < 0) {
       return null;
   }
   if (prefix >= fileName.length()) {
       if (includeSeparator) {
           return getPrefix(fileName);  // add end slash if necessary
       }
       return fileName;
   }
   final int index = indexOfLastSeparator(fileName);
   if (index < 0) {
       return fileName.substring(0, prefix);
   }
   int end = index + (includeSeparator ?  1 : 0);
   if (end == 0) {
       end++;
   }
   return fileName.substring(0, end);
}
```

Figure 1, FilenameUtils, by Ruokun Xu

To revise the method based on testable design principles, we changed the private into the public to make it testable.

Test case for Testable Design

Test case for getFullPath method by a string of file path and parameter of whether include the end separator. The following test case test different file path with its expected output.

```java
@Test
public void testDoGetFullPath(){
   assertEquals("\\a\\b", FilenameUtils.doGetFullPath("\\a\\b\\c.txt", false));
```

```
    assertEquals("\\a\\b\\", FilenameUtils.doGetFullPath("\\a\\b\\c.txt", 
true));
    assertEquals("D:/a/b/", FilenameUtils.doGetFullPath("D:/a/b/c", true));
    assertEquals("/a/b", FilenameUtils.doGetFullPath("/a/b/c", false));
}
```

Mocking and its Utilities

Mocking uses fake objects to check whether unit tests pass or fail by observing the interactions among objects(Jones, pg 23). It is a technique that helps developers to test components without relying on their actual dependencies, as mock objects substitute those dependencies and simulate behaviors (Hall, StackOverFlow, 2010).

To fully understand the benefit of mocking, we should first learn more about some challenges in integration testing. With integration testing, developers and testers check some dependent software components in a group to verify software quality and look for potential integration issues(Jones, page 10). However, integration testing may face a few challenges:

- When an issue arises, it is hard to detect which dependency causes the error.
- It is difficult to test the interactions between software modules when some parts are not implemented yet.
- Some testing may change the state of other components such as databases, leading to extra work to reset to the original state.

With the help of mocking, developers tackle those problems as mocking has a few benefits:

- Mock objects can simulate the behaviors of some real objects
- Mocking allows developers to develop and write integration tests at the same time
- Mock objects won't affect the real object state, thus avoiding side-effects

Tested a Feature Using Mockito

As you may recall, our project Apache Commons IO is a library of utility features helping developers deal with IO. The feature, in the PathFileComparator class, that we picked is to compare two files' paths depending on the case sensitivity. It should be noted that it has dependency on File class, which makes it challenging to test.

Without mocking, it is difficult to test this feature for the following reasons:

- We had to create new files in the current directory and to call File class's constructor for the setup
- We had to find out which folder authorizes us to write those testing files
- We had to spend a lot of time writing logic in creating and cleaning testing files

- We had to allocate some disk space for those testing files, which is a waste of space

Thanks to Mockito, a mocking framework that allows us to write clean and lightweight test cases using its APIs, we were able to test our features and resolve all the issues that we mentioned before. With Mockito, we can easily mock File class using `Mockito.mock(File.class)` syntax. And assign expected behaviors to the mock objects using `.when` and `.thenReturn`. For example, Mockito.when(file1.getPath()).thenReturn("test/file1.txt") allows file1 mock object to return a string with value of "test/file1.txt".

Therefore, we successfully tested our feature using Mockito and enjoyed a few benefits:
- There was no need to create actual files, as we mocked the File class.
- We didn't need to clean any actual files after running our test cases
- Increased our productivity and saved development time, as we didn't add any testing files in any folders
- Specified the behavior of mocking object to test our feature without worrying about the internal implementation of dependent components.

Test cases that we wrote in Mockito:

Here is a code snippet that we utilized Mockito to test this feature:
- Used Mockito to mock different files and to compare the path of two files.
- Tested case sensitive comparator and case insensitive comparator and expected compare function's output.

```
@Test
public void testCompare(){
   final File file1 = Mockito.mock(File.class);
   Mockito.when(file1.getPath()).thenReturn("test/file1.txt");
   final File file2 = Mockito.mock(File.class);
   Mockito.when(file2.getPath()).thenReturn("test/file1.txt");
   final File file3 = Mockito.mock(File.class);
   Mockito.when(file3.getPath()).thenReturn("TEST/file1.txt");
   final File file4 = Mockito.mock(File.class);
   Mockito.when(file4.getPath()).thenReturn("tes/file1.txt");

   final Comparator<File> sensitiveComparator = new PathFileComparator();
   assertEquals(0, sensitiveComparator.compare(file1, file2), "sensitive file1
& file2 = 0");
   assertTrue(sensitiveComparator.compare(file1, file3) > 0, "sensitive file1 &
file3 > 0");
   assertTrue(sensitiveComparator.compare(file1, file4) > 0, "sensitive file1 &
file4  > 0");
```

```
   final Comparator<File> insensitiveComparator =
PathFileComparator.PATH_INSENSITIVE_COMPARATOR;
   assertEquals(0, insensitiveComparator.compare(file1, file2), "insensitive
file1 & file2 = 0");
   assertEquals(0, insensitiveComparator.compare(file1, file3), "insensitive
file1 & file3 = 0");
   assertTrue(insensitiveComparator.compare(file1, file4) > 0, "insensitive
file1 & file4 > 0");
   assertTrue(insensitiveComparator.compare(file3, file4) > 0, "insensitive
file3 & file4  > 0");
}
```

Reference

Davis, H. (2019, October 5). *What is a testable design? – QuickAdviser*. Harry Davis. https://quick-adviser.com/what-is-a-testable-design/

Jones, J. (2022b). Integration_Testing_Mocking_Testable_Design [Slides]. Canvas. https://canvas.eee.uci.edu/courses/43617/files/folder/Lecture%20Slides?preview=17765481

What is Mocking? (2010, April 19). Stack Overflow. https://stackoverflow.com/questions/2665812/what-is-mocking