

QL-4) Using TDD to design production code.



We are going to examine the benefits of using TDD as a design technique and see what the benefits are if we follow a TDD approach to developing production code.

QL-4.1) Phase 1

Create a new test class called `LiveFileLoaderTest`.

Write the unit test first for a CUT called `FileLoader`. It should only have one public method on it. Once the test is written develop the CUT to pass the test.

```
public int loadFile(string fname)
```



The above function should load a text file from the disk, and return the number of characters in it.

Your test should specify the input file name including its path, and the number of characters expected to be in the file.

Assert on the actual file length (number of characters in the file) against the expected file length (number of characters read from the file)

In the production class we suggest you use something like `File.readAllLines()`

Once you have completed the above test and production code, think about this question

- Question: What's the weakness of this design

A solution can be found [here](#)

QL-4.2) Phase 2

Create a new test class called `FileLoaderTest`.

Write the unit test first for a CUT called `FileLoader`. It should only have a new public method on it. Once the test is written develop the CUT to pass the test.

In order to facilitate your design, you need to think about the structure of the test

1. In the `//Act` part of the test, we want to write something like this

```
2.      // act
      int bytesRead = cut.loadFile((fname) ->
      {
          List<String> result = null;
          try
          {
              result = Files.readAllLines(Paths.get(fname),
StandardCharsets.UTF_8);
          }
          catch (IOException e){}
          return result;
      });
```

3. Reading the API guides we see that `File.readAllLines()` returns a `List<string>`, so let's change the code to match this

```
4. package com.celestial.mockito.filetodb;
```

```

import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.ArrayList;
import java.util.List;
import static org.junit.jupiter.api.Assertions.assertEquals;
import org.junit.jupiter.api.Test;
import org.mockito.MockedStatic;
import org.mockito.Mockito;

public class FileLoaderTest
{
    // Redesign the FileLoader so that the mechanism to load files
    // up can be
    // passed in as a lambda - still tightly coupled the file
    // system
    @Test
    public void
load_all_of_file_using_inbuilt_Files_type_as_lambda()
    {
        // arrange
        String fileToLoad = "c:/tmp/KeyboardHandler.txt";
        FileLoader cut = new FileLoader(fileToLoad);
        int expectedBytesRead = 10; //1371;
        List<String> pretendFileContent = new ArrayList<>();
        pretendFileContent.add("Hello");
        pretendFileContent.add("world");
        MockedStatic<Files> ff = Mockito.mockStatic(Files.class);
        ff.when(() -> Files.readAllLines(Paths.get(fileToLoad),
StandardCharsets.UTF_8)).thenReturn(pretendFileContent);

        // act
        int bytesRead = cut.loadFile((fname) ->
        {
            List<String> result = null;
            try
            {
                result = Files.readAllLines(Paths.get(fname),
StandardCharsets.UTF_8);
            }
            catch (IOException e){}
            return result;
        });

        // assert
        assertEquals(expectedBytesRead, bytesRead);
    }
}

```

5. At line 28, we are using Mockito's `mockStatic()` to mock a Static class
6. At line 29, we setup the expectation
7. So we need to design a `loadFile()` method that takes as a parameter a block of code that can read the file from any source.
8. `public int LoadFile(string fname, ILoadFile func)`
9. Create a new interface called

```
10. package com.celestial.mockito.filetoadb;

import java.util.List;

/**
 *
 * @author selvy
 */
@FunctionalInterface
public interface ILoader
{
    List<String>    loadFile(String fname);
}
```

11. Modify `FileLoader` so it has this additional functionality

```
12.     public List<String> getLines() {
        return lines;
    }

    int loadFile(ILoader func)
    {
        lines = func.loadFile(fileToLoad);
        return calculateFileSize();
    }

    ...
```

USING MOCKITO `mockStatic()`

Before we can use Mockito for mocking static methods, we need to configure it to activate inline *MockMaker*.

We need to add a text file to the project's `src/test/resources/mockito-extensions` directory named `org.mockito.plugins.MockMaker` and add a single line of text:

```
mock-maker-inline
```



By moving the mechanics of how a file is loaded into `FileLoader` out of `FileLoader`, we've decoupled the functionality. This new design now means we can instruct `FileLoader` to load a file resource in different ways, not just using the standard file IO.

So the test has forced us to think carefully about what we think is the purpose of `FileLoader`. Our initial intent may have been to load a file from the disk, which it can do, but we have broadened its design so it is more flexible and less brittle.

Once you have completed the above test and production code, think about this question

- Question: What's the weakness of this design

A solution can be found [here](#)

QL-4.3) Phase 3 - working stubs

The phase 2 test was weakened by the fact that it broke one of the tenants of a good test

- A test should not make any IO calls

Because we have designed the FileLoader so that a lambda can be passed in, we can rethink what the test should look like.

Create a new test that specifically targets using canned data.

In the //Act part of the test redesign the lambda expression so that it returns a list of strings.


You should not have to change the CUT.

Once you have completed the above test and production code, think about this question

- Question: Can we further improve the test

A solution can be found [here](#)

QL-4.4) Phase 4 - working with mocks

 Mocks are not stubs

A Mock can be used to create Stubs, and Dummies

A mocking library will create a dummy object that must be populated with methods that have predefined results and parameters

In this implementation of the test we want to use `java.nio.file.Files` object. But wait, `java.nio.file.Files.ReadLines()` is a class-level operation and not an object-level operation. Most mocking frameworks struggle with mocking class-level methods.

We are going to use a mocking library called Mockito (you saw it being used earlier). It's one of the more popular mocking libraries for Java code

Begin by writing your test in the normal way

```
// arrange
string fileToLoad = "c:/tmp/KeyboardHandler.java.txt";
FileLoader cut = new FileLoader();
// setup ur canned data, these will represent the lines in the
file
List<string> pretendFileContent = new();
pretendFileContent.Add("Hello");
pretendFileContent.Add("world");
int expectedBytesRead = 10;
```

Because `java.io.File` is a static class, this time we will create our own dummy interface with the required methods for our tests and CUT

Create this file in tests folder and not the production folder

```

package com.celestial.mockito.filetodb;

import java.nio.charset.Charset;
import java.nio.file.Path;
import java.util.List;

public interface MyFile
{
    public List<String> readAllLines(Path path, Charset cs);
}

```

In our test, we can now mock this interface and set the expectations for the required method

```

// Mock the interface
MyFile file = mock(MyFile.class);

// Setup the expectation
when(file.readAllLines(Paths.get(fileToLoad), StandardCharsets.
UTF_8)).thenReturn(pretendFileContent);

```

Line 2 creates a skeletal object of the MyFile interface.

Line 5 tells the mocked object that when the method **readAllLines()** is invoked the parameter value given in **fileToRead**, it should return the object **pretendFileContent**. We will look at this more closely in a short while.

Now in the //Act part of the test add the following code

```

// act
int bytesRead = cut.loadFile((fname) ->
{
    List<String> result = file.readAllLines(Paths.get(fname),
StandardCharsets.UTF_8);

    return result;
});

```

In the lambda's body, we simply invoke `file.readAllLines(...)`. This is using the mocked object and mocked method we previously set up.

The Assert statement should be no different from the previous Assert statements in the other tests.

Run the test.

A solution can be found [here](#)

Do expectation parameters really matter?

Let's look at the code

```

// arrange
String fileToLoad = "c:/tmp/KeyboardHandler.java.txt";
FileLoader cut = new FileLoader( fileToLoad );
...
...
// Mock the interface
MyFile file = mock(MyFile.class);

// Setup the expectation
when(file.readAllLines(Paths.get(fileToLoad), StandardCharsets.
UTF_8)).thenReturn(pretendFileContent);

// act
int bytesRead = cut.loadFile((fname) ->
{
    List<String> result = file.readAllLines(Paths.get(fname),
StandardCharsets.UTF_8);

    return result;
});

```

Change line 15 to (we've passed in **XYZ** as the file name)

```

        List<String> result = file.readAllLines(Paths.get
("XYZ"), StandardCharsets.UTF_8);

```

Rerun the test and see what happens. It should fail.

When you set up an expectation on a mock object, you are not just specifying what method will be called, you are stating what values will be passed in as parameters to that method, and what value should be returned if those values match what you have said. In other words, if when the test is run the test values match the expected values, then the value you have stated will be returned. This is why it is called setting up the expectations.

 [Git repo](#)