hw4_testable_design_mocking.md

Testable Design and Mocking

Shrijith Saraswathi Venkatramana (shrijits@uci.edu)

Testable Design

Aspects/Goals of what makes a testable design

In simple terms, testable design is a way of structuring a code-base to enable maximum code coverage.

This means, minimizing certain kinds of design patterns. Following are some aspects for achieving testable design:

1. Avoid complex private methods (because these cannot be tested)
2. Avoid static methods (because side-effects cannot be harnessed)
3. Avoid hardcoding in "new" ("new" objects cannot be stubbed)
4. Avoid business logic in constructors
5. Avoid singleton pattern

Example in OpenNLP

The `ArgumentParser` class has an **inner class**, which is private and static. This hosts a utility function which takes in a method (obtained through reflection), a string parameter name and value.

The class checks whether the value is of integer type and if it is not, an exception is thrown with a specific error message.

Since this inner-class is private, it cannot be invoked by testing tools.

```
public class ArgumentParser {
  ...

  private static class IntegerArgumentFactory  implements ArgumentFactory {

    public Object parseArgument(Method method, String argName, String argValue) {

      Object value;

      try {
        value = Integer.parseInt(argValue);
      }
      catch (NumberFormatException e) {
        throw new TerminateToolException(1, String.format(INVALID_ARG, argName, argValue) +
            "Value must be an integer!", e);
      }

      return value;
    }

    ....
}
```

Making `IntergerArgumentFactory` testable

Since the method of generic nature, it makes sense to make it public class. So I rewrote this inner-class as public:

```
  public class IntegerArgumentFactoryRedefined  implements ArgumentFactory {

    public Object parseArgument(Method method, String argName, String argValue) {

      Object value;
```

```
      try {
        value = Integer.parseInt(argValue);
      }
      catch (NumberFormatException e) {
        throw new TerminateToolException(1, String.format(INVALID_ARG, argName, argValue) +
                "Value must be an integer!", e);
      }

      return value;
    }
  }
```

New Test Case

The following code uses Java's reflection capability to create a mock `Method` object, and then calls the new `IntegerArgumentFactory`

```
  @Test
  public void testIntegerArgumentFactory() {
    Class aParser = ArgumentParser.class;
    Method[] mlist = aParser.getMethods();
    ArgumentParser ap = new ArgumentParser();
    ArgumentParser.IntegerArgumentFactoryRedefined iaf = ap.new IntegerArgumentFactoryRedefined();
    Integer i = (Integer) iaf.parseArgument(mlist[0], "hello", "3");
    Assert.assertTrue(i==3);
  }
```

Mocking

Describe mocking

The key idea behind mocking is to isolate a unit from its dependencies through the use of fake/mock objects. So if a class A depends on an object of class B, then a fake object of class B is created through a framework such as `Mockito`.

The fake object by Mockito can mimic the original object for method calls and it can also keep track of the number of calls to different methods of the mock object to determine the expected sort of interaction.

Mockable feature: `ParagraphStream`

A `ParagraphStream` is an object that's used to build up documents.

The ParagraphStream takes in Java streams and then using certain heuristics, converts them into an iterable of strings.

Utility of mocking in this context is that we can ensure how many times certain methods were called. For example, if an input stream has 10 "blanks", indicating new lines, by the heuristic, one expects 5 "paragraph" entities. We can check for this number through Mockito `verify`.

Testing `ParagraphStream`

```
public class ParagraphStreamMockTest {
    @Test
    public void testSimpleReading() throws IOException {
        ParagraphStream paraStream = Mockito.mock(ParagraphStream.class);
        // following corresponds to: 1 2 " " " " 4 5 " "
        when(paraStream.read()).thenReturn("1\n2\n", "4\n5\n", null);

        String the_para;
        List<String> para_list = new ArrayList<String>();
        while ((the_para = paraStream.read()) != null)
            para_list.add(the_para);

        Assert.assertEquals("1\n2\n", para_list.get(0));
        Assert.assertEquals("4\n5\n", para_list.get(1));

        verify(paraStream, times(3)).read();
        verify(paraStream, never()).reset();
```

```
            verifyNoMoreInteractions(paraStream);
        }
    }
```

Note that `thenReturn` mocks paraStream behavior by giving a different return value on every subsequent call.

This behavior is used to build up a list of strings `para_list`, which is like a document.

We ensure the number of times paraStream is caled through Mockito's `verify`.