http://www.yegor256.com/2015/05/25/unit-test-scaffolding.html

A Few Thoughts on Unit Test Scaffolding

25 May 2015 modified on 25 May 2015 Yegor Bugayenko

When I start to repeat myself in unit test methods by creating the same objects and preparing the data to run the test, I feel disapointed in my design. Long test methods with a lot of code duplication just don't look right. To simplify and shorten them, there are basically two options, at least in Java: 1) private properties initialized through `@Before` and `@BeforeClass`, and 2) private static methods. They both look anti-OOP to me, and I think there is an alternative. Let me explain.

© Léon: The Professional by Luc Besson

JUnit officially suggests a test fixture:

```
public final class MetricsTest {
  private File temp;
  private Folder folder;
  @Before
```

```
  public void prepare() {
    this.temp = Files.createTempDirectory("test");
    this.folder = new DiscFolder(this.temp);
    this.folder.save("first.txt", "Hello, world!");
    this.folder.save("second.txt", "Goodbye!");
  }
  @After
  public void clean() {
    FileUtils.deleteDirectory(this.temp);
  }
  @Test
  public void calculatesTotalSize() {
    assertEquals(22, new Metrics(this.folder).size());
  }
  @Test
  public void countsWordsInFiles() {
    assertEquals(4, new Metrics(this.folder).wc());
  }
}
```

I think it's obvious what this test is doing. First, in `prepare()`, it creates a "test fixture" of type `Folder`. That is used in all three tests as an argument for the `Metrics` constructor. The real class being tested here is `Metrics` while `this.folder` is something we need in order to test it.

What's wrong with this test? There is one serious issue: **coupling** between test methods. Test methods (and all tests in general) must be perfectly isolated from each other. This means that changing one test must not affect any others. In this example, that is not the case. When I want to change the `countsWords()` test, I have to change the internals of `before()`, which will affect the other method in the test "class".

With all due respect to JUnit, the idea of creating test fixtures in `@Before` and `@After` is wrong, mostly because it encourages developers to couple test methods.

Here is how we can improve our test and isolate test methods:

```
public final class MetricsTest {
  @Test
  public void calculatesTotalSize() {
    final File dir = Files.createTempDirectory("test-1");
    final Folder folder = MetricsTest.folder(
      dir,
      "first.txt:Hello, world!",
      "second.txt:Goodbye!"
    );
    try {
      assertEquals(22, new Metrics(folder).size());
    } finally {
      FileUtils.deleteDirectory(dir);
    }
  }
  @Test
  public void countsWordsInFiles() {
    final File dir = Files.createTempDirectory("test-2");
    final Folder folder = MetricsTest.folder(
      dir,
      "alpha.txt:Three words here",
      "beta.txt:two words"
      "gamma.txt:one!"
    );
    try {
      assertEquals(6, new Metrics(folder).wc());
    } finally {
      FileUtils.deleteDirectory(dir);
    }
  }
  private static Folder folder(File dir, String... parts) {
    Folder folder = new DiscFolder(dir);
    for (final String part : parts) {
      final String[] pair = part.split(":", 2);
      this.folder.save(pair[0], pair[1]);
    }
    return folder;
  }
}
```

Does it look better now? We're not there yet, but now our test methods are perfectly isolated. If I want to change one of them, I'm not going to affect

the others because I pass all configuration parameters to a private static utility (!) method `folder()`.

A utility method, huh? Yes, it smells.

The main issue with this design, even though it is way better than the previous one, is that it doesn't prevent code duplication between test "classes". If I need a similar test fixture of type `Folder` in another test case, I will have to move this static method there. Or even worse, I will have to create a utility class. Yes, there is nothing worse in object-oriented programming than utility classes.

A much better design would be to use "fake" objects instead of private static utilities. Here is how. First, we create a fake class and place it into `src/main/java`. This class can be used in tests and also in production code, if necessary (`Fk` for "fake"):

```
public final class FkFolder implements Folder, Closeable {
  private final File dir;
  private final String[] parts;
  public FkFolder(String... prts) {
    this(Files.createTempDirectory("test-1"), parts);
  }
  public FkFolder(File file, String... prts) {
    this.dir = file;
    this.parts = parts;
  }
  @Override
  public Iterable<File> files() {
    final Folder folder = new DiscFolder(this.dir);
    for (final String part : this.parts) {
      final String[] pair = part.split(":", 2);
      folder.save(pair[0], pair[1]);
    }
    return folder.files();
  }
  @Override
  public void close() {
    FileUtils.deleteDirectory(this.dir);
```

```
  }
}
```

Here is how our test will look now:

```
public final class MetricsTest {
  @Test
  public void calculatesTotalSize() {
    final String[] parts = {
      "first.txt:Hello, world!",
      "second.txt:Goodbye!"
    };
    try (final Folder folder = new FkFolder(parts)) {
      assertEquals(22, new Metrics(folder).size());
    }
  }
  @Test
  public void countsWordsInFiles() {
    final String[] parts = {
      "alpha.txt:Three words here",
      "beta.txt:two words"
      "gamma.txt:one!"
    };
    try (final Folder folder = new FkFolder(parts)) {
      assertEquals(6, new Metrics(folder).wc());
    }
  }
}
```

What do you think? Isn't it better than what JUnit offers? Isn't it more reusable and extendable than utility methods?

To summarize, I believe scaffolding in unit testing must be done through fake objects that are shipped together with production code.