Tests

Patterns and Anti-Patterns

Yegor Bugayenko

Lecture #14 out of 16
90 minutes

Chapter #1:

Twenty Three Test Anti-Patterns

https://www.yegor256.com/2018/12/11/unit-testing-anti-patterns.html →

- Cuckoo
- Test-per-Method
- Anal Probe
- Conjoined Twins
- Happy Path
- Slow Poke
- Giant
- Mockery
- Inspector
- Generous Leftovers
- Local Hero
- Nitpicker
- Secret Catcher
- Dodger
- Loudmouth
- Greedy Catcher
- Sequencer
- Enumerator
- Free Ride
- Excessive Setup
- Line hitter
- Forty-Foot Pole Test
- The Liar

Test-per-Method

Although a one-to-one relationship between test and production classes is a reasonable starting point, a one-to-one relationship between test and production method is almost always a bad idea.

Anal Probe

A test that has to use unhealthy ways to perform its task, such as reading private fields using reflection.

Happy Path

The tests stay on happy paths (i.e. expected results, e.g. 18 years old) without testing for boundaries and exceptions (e.g. -2 years old)

Slow Poke

A unit test that runs incredibly slow. When developers kick it off, they have time to go to the bathroom, grab a smoke, or worse, kick the test off before they go home at the end of the day.

Giant

A unit test that, although it is validly testing the object under test, can span thousands of lines and contain many many test cases. This can be an indicator that the system-under-test is a God Object.

Mockery

Sometimes mocking can be good, and handy. But sometimes developers can lose themselves in their effort to mock out what isn't being tested. In this case, a unit test contains so many mocks, stubs, and/or fakes that the system under test isn't even being tested at all, instead data returned from mocks is what is being tested.

Inspector

A unit test that violates encapsulation in an effort to achieve 100% code coverage, but knows so much about what is going on in the object that any attempt to refactor will break the existing test and require any change to be reflected in the unit test.

Generous Leftovers (aka Chain Gang, Wet Floor)

An instance where one unit test creates data that is persisted somewhere, and another test reuses the data for its own devious purposes. If the "generator" is ran afterward, or not at all, the test using that data will outright fail.

Local Hero (aka Hidden Dependency)

A test case that is dependent on something specific to the development environment it was written on, in order to run. The result is that the test passes on development boxes, but fails when someone attempts to run it elsewhere.

Dodger

A unit test which has lots of tests for minor (and presumably easy to test) side effects, but never tests the core desired behavior. Sometimes you may find this in database access related tests, where a method is called, then the test selects from the database and runs assertions against the result.

Loudmouth

A unit test (or test suite) that clutters up the console with diagnostic messages, logging, and other miscellaneous chatter, even when tests are passing.

Sequencer

A unit test that depends on items in an unordered list appearing in the same order during assertions.

Enumerator (aka Test With No Name)

Unit tests where each test case method name is only an enumeration, e.g. test1, test2, test3. As a result, the intention of the test case is unclear, and the only way to be sure is to read the test case code and pray for clarity.

Free Ride (aka Piggyback)

Rather than write a new test case method to test another feature or functionality, a new assertion rides along in an existing test case.

Excessive Setup (aka Mother Hen)

A test that requires a lot of work to set up in order to even begin testing. Sometimes several hundred lines of code are used to setup the environment for one test, with several objects involved, which can make it difficult to really ascertain what is being tested due to the "noise" of all of the setup.

Line hitter

At first glance, the tests cover everything and code coverage tools confirm it with 100%, but in reality the tests merely hit the code, without doing any output analysis.

The Liar (aka Evergreen Tests)

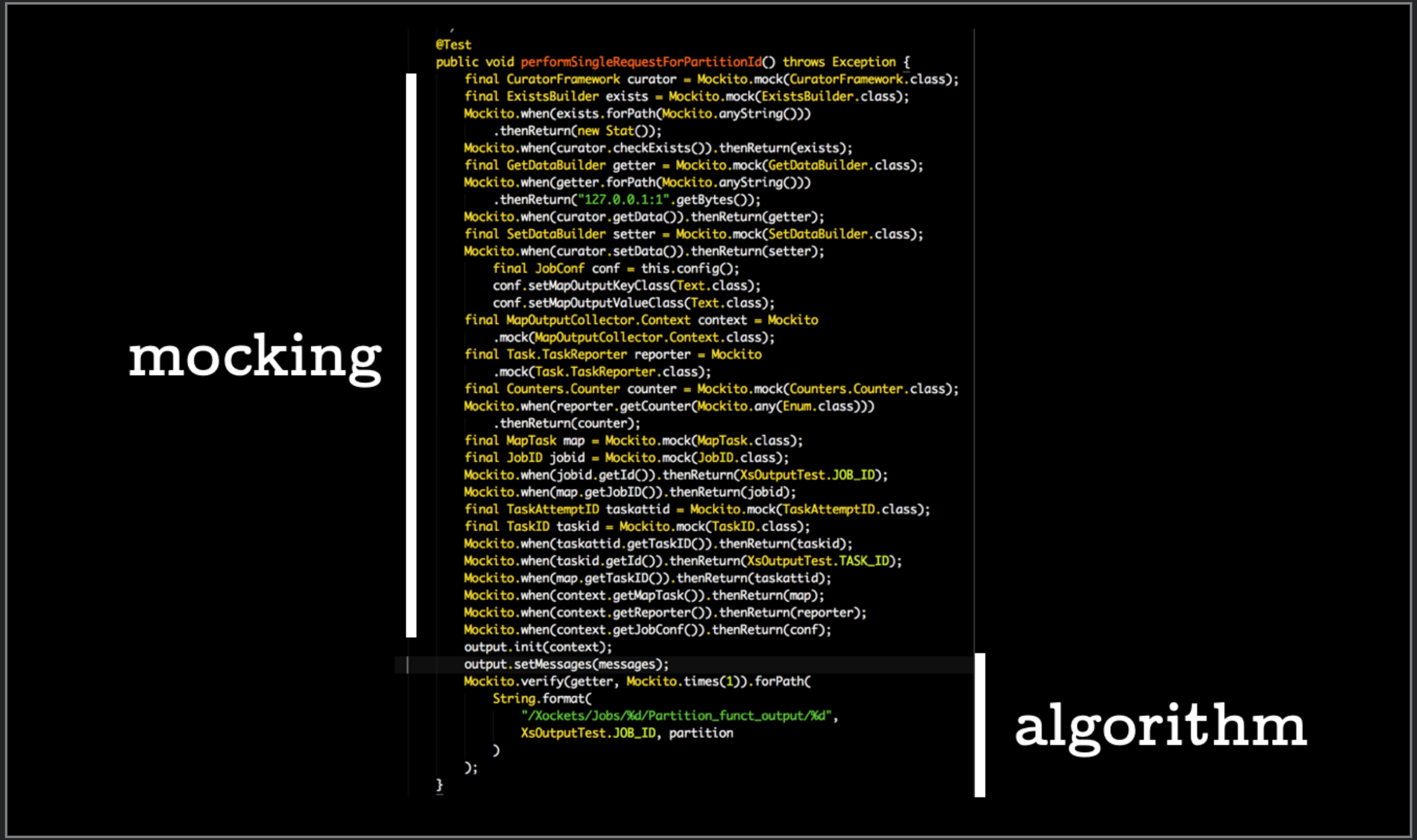
A test doesn't validate any behaviour and passes in every scenario. Any new bug introduced in the code will never be discovered by this test. It was probably created after the implementation was finished, so the author of this test couldn't know whether the test actually tests something.

Chapter #2:

Mocking Frameworks

Mocking Frameworks are Evil

https://www.youtube.com/watch?v=1bAixLa0CSA →



```java
@Test
public void performSingleRequestForPartitionId() throws Exception {
    final CuratorFramework curator = Mockito.mock(CuratorFramework.class);
    final ExistsBuilder exists = Mockito.mock(ExistsBuilder.class);
    Mockito.when(exists.forPath(Mockito.anyString()))
        .thenReturn(new Stat());
    Mockito.when(curator.checkExists()).thenReturn(exists);
    final GetDataBuilder getter = Mockito.mock(GetDataBuilder.class);
    Mockito.when(getter.forPath(Mockito.anyString()))
        .thenReturn("127.0.0.1:1".getBytes());
    Mockito.when(curator.getData()).thenReturn(getter);
    final SetDataBuilder setter = Mockito.mock(SetDataBuilder.class);
    Mockito.when(curator.setData()).thenReturn(setter);
        final JobConf conf = this.config();
        conf.setMapOutputKeyClass(Text.class);
        conf.setMapOutputValueClass(Text.class);
    final MapOutputCollector.Context context = Mockito
        .mock(MapOutputCollector.Context.class);
    final Task.TaskReporter reporter = Mockito
        .mock(Task.TaskReporter.class);
    final Counters.Counter counter = Mockito.mock(Counters.Counter.class);
    Mockito.when(reporter.getCounter(Mockito.any(Enum.class)))
        .thenReturn(counter);
    final MapTask map = Mockito.mock(MapTask.class);
    final JobID jobid = Mockito.mock(JobID.class);
    Mockito.when(jobid.getId()).thenReturn(XsOutputTest.JOB_ID);
    Mockito.when(map.getJobID()).thenReturn(jobid);
    final TaskAttemptID taskattid = Mockito.mock(TaskAttemptID.class);
    final TaskID taskid = Mockito.mock(TaskID.class);
    Mockito.when(taskattid.getTaskID()).thenReturn(taskid);
    Mockito.when(taskid.getId()).thenReturn(XsOutputTest.TASK_ID);
    Mockito.when(map.getTaskID()).thenReturn(taskattid);
    Mockito.when(context.getMapTask()).thenReturn(map);
    Mockito.when(context.getReporter()).thenReturn(reporter);
    Mockito.when(context.getJobConf()).thenReturn(conf);
    output.init(context);
    output.setMessages(messages);
    Mockito.verify(getter, Mockito.times(1)).forPath(
        String.format(
            "/Xockets/Jobs/%d/Partition_funct_output/%d",
            XsOutputTest.JOB_ID, partition
        )
    );
}
```

Instead: Doubles or Fake Objects

https://www.youtube.com/watch?v=CCr8hRE_TUs →

```
class Book {
  private int id;
  private Database database;
  Book(Database db, int i) {
    this.database = db;
    this.id = i;
  }
  String title() {
    return this.database.fetch(
      “SELECT title FROM book WHERE id=?”,
      this.id
    );
  }
}
```

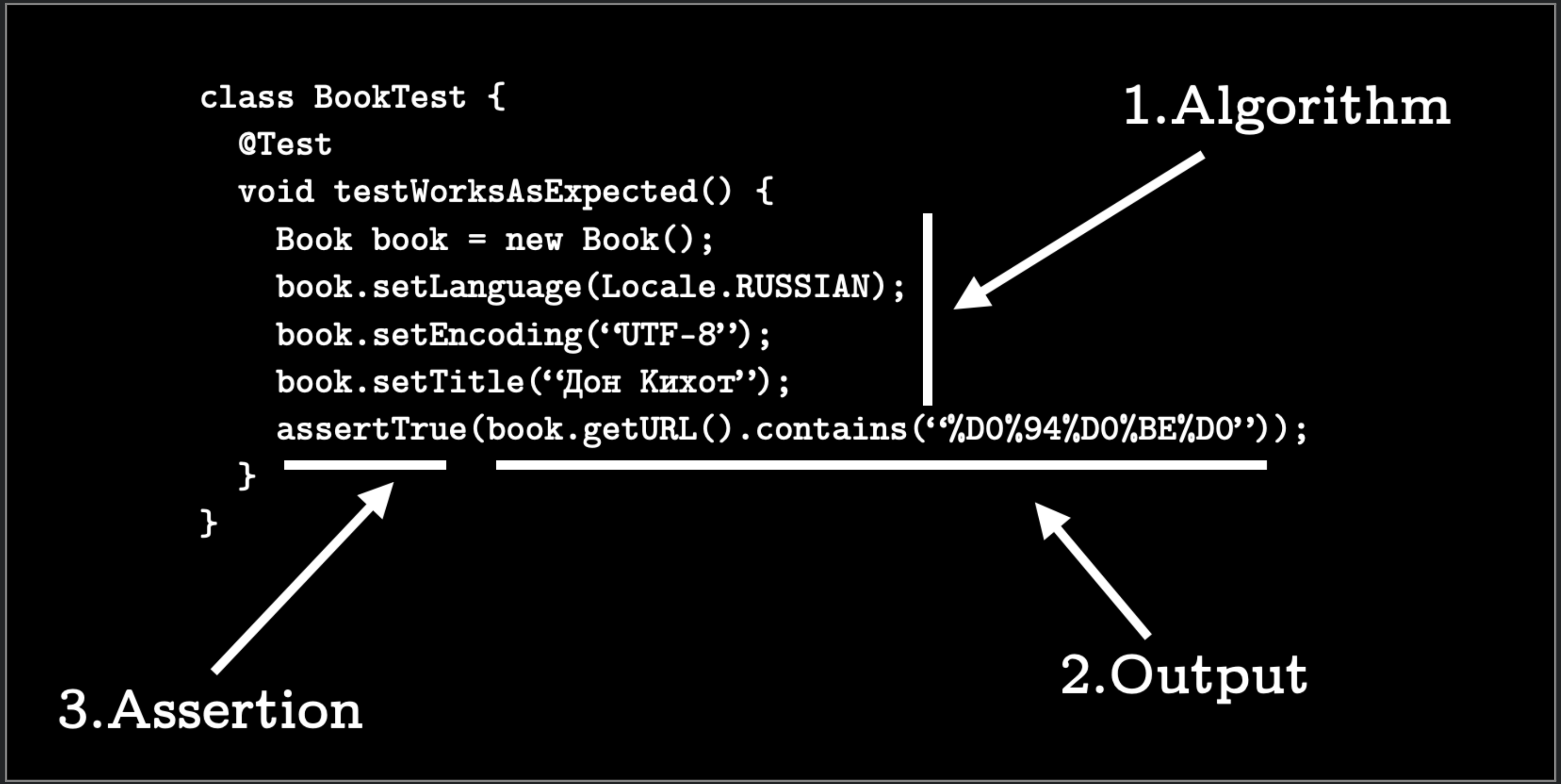
```
class BookTest {
  @Test
  public canFetchTitleFromDatabase() {
    Book book = new Book(
      new FakeDatabase(), 123
    );
    assert book.title().equals(‘UML Distilled’);
  }
}
```

Chapter #3:

OOP + Tests

xUnit Tests are Procedural

https://www.youtube.com/watch?v=CCr8hRE_TUs →



```
class BookTest {
  @Test
  void testWorksAsExpected() {
    Book book = new Book();
    book.setLanguage(Locale.RUSSIAN);
    book.setEncoding(‘‘UTF-8’’);
    book.setTitle(‘‘Дон Кихот’’);
    assertTrue(book.getURL().contains(‘‘%D0%94%D0%BE%D0’’));
  }
}
```

Instead: Single-Method Unit Tests

https://www.youtube.com/watch?v=CCr8hRE_TUs →

```
class BookTest {
  @Test
  void testWorksAsExpected() {
    // nothing goes here
    assertThat(
      new Book("Дон Кихот", "UTF-8", Locale.RUSSIAN),
      new HasURL(new StringContains("%D0%94%D0%BE%D0"))
    );
  }
}
```

Chapter #4:

Testable Code

All Dependencies Are Injectable

Wrong

```
import java.nio.file.Files;
class Book {
  String title() {
    return Files.readAllLines(
      Paths.get("/my-files/book.txt")
    )[0];
  }
}
```

Right

```
import java.nio.file.Files;
class Book {
  private Path file;
  Book(Path f) {
    this.file = f;
  }
  String title() {
    return Files.readAllLines(
      this.file
    )[0];
  }
}
```

Each Interface Has a Fake Object

Interface

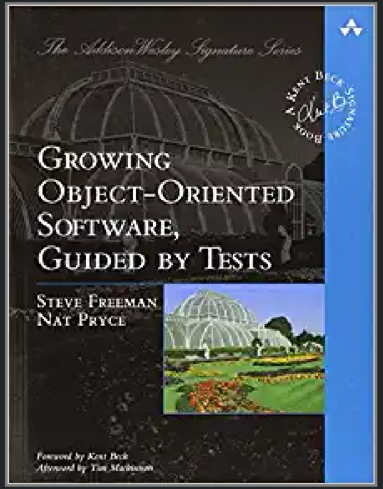
```java
interface Book {
  String title();
  String author();
}
```

Fake Book

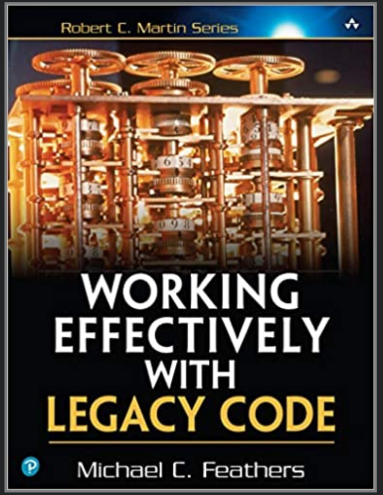
```java
class FakeBook implements Book {
  @Override
  String title() {
    return "Fake Title";
  }
  @Override
  String author() {
    return "Fake Author";
  }
}
```

Chapter #5:

Books, Venues, Call-to-Action



"Growing Object-Oriented Software, Guided by Tests" by Steve Freeman et al.



"Working Effectively with Legacy Code" by Michael Feathers

Where to go:

International Symposium on Software Testing and Analysis (ISSTA)

Call to Action:

Get rid of mocking framework in your code and only use fake objects.

Still unresolved issues:

- How to detect test anti-patterns automatically?
- How to refactor tests automatically?
- How to write fake objects faster?
- How to educate programmers to write better tests?