TDD: Test Doubles

CS 3250
Software Testing

[Lasse Koskela, “Test Driven”, Chapter 4]
[Tilo Linz, “Testing in Scrum”, Chapter 4]
Overview of TDD Process

1. From user story to requirements to tests
2. Choosing the first test
3. Breadth-first, depth-first
4. Let’s not forget to refactor
5. Adding a bit of error handling
6. Loose ends on the test list
7. Repeat

Test first – make it run – make it better
3. Breadth-First, Depth-First

- What to do with a “hard” red phase?
  - Issue is “What to fake” vs. “What to build”

- “Faking” is an accepted part of TDD
  - That is, “deferring a design decision”
Breadth-First

- Implement the higher-level functionality first by faking the required lower-level functionality
Depth-First

- Implement the lower-level functionality first and only compose the higher-level functionality once all the ingredients are present.
Test Double

• A stand-in for something that would be real in the program (~Hollywood’s stunt double)

• Five primary types of test doubles:
  • Dummy – a double that blows up when used
  • Stub – a double with hard-coded return values
  • Spy – a double that be can interrogated to verify for correctness
  • Mock – a spy that verifies itself
  • Fake – a behavioral mimic
Example

• Imagine you are writing and testing a program that controls a rocket launching

```java
static void launchRocket(Rocket rocket, LaunchCode code) {
    // ...
}
```

• Can’t interact with a real, live rocket

• Need to double that rocket

• Rely on the idea of a rocket, and allow the runtime to provide a rocket to work with

Rocket interface
• For the situation when an expired or invalid code is given, use a dummy to ensure the rocket is not fired

• (+) Simple

• (-) May not be intuitive; no traditional setup-act-assert test structure

class DummyRocket implements Rocket {
    @Override
    void launch() {
        throw new RuntimeException();
    } ...

    static void launchRocket(Rocket rocket, LaunchCode code) {
        try {
            rocket.launch()
        } catch (Exception e) {
        }
    }

    @Test
    void givenExpiredLaunchCode_RocketNotLaunched() {
        launchRocket(new DummyRocket(), expiredCode);
    }
}
Spy

class SpyRocket implements Rocket {
    private boolean launchWasCalled = false;
    @Override
    void launch() {
        launchWasCalled = true;
    }
    boolean launchWasCalled() {
        return launchWasCalled;
    }
}

@Test
void givenExpiredLaunchCode_RocketNotLaunched() {
    SpyRocket spy = new SpyRocket();
    launchRocket(spy, expiredCode);
    assertEquals(false, spy.launchWasCalled);
}

• Use a spy so that a test can interrogate
• (+) More readable; traditional setup-act-assert test structure
• (-) Tests are coupled to the implementation (must know about the implementation, instead of just focusing on behavioral outputs)
Mock

@Test
test givenUnencryptedLaunchCode_RocketNotLaunched() {
  MockRocket mockRocket = new MockRocket();
  launchRocket(mockRocket, unencryptedCode);
  assertEquals(false, mockRocket.launchWasCalled());
  assertEquals(true, mockRocket.disableWasCalled());
}

@Test
test givenExpiredLaunchCode_RocketNotLaunched() {
  MockRocket mockRocket = new MockRocket();
  launchRocket(mockRocket, expiredCode);
  assertEquals(false, mockRocket.launchWasCalled());
  assertEquals(true, mockRocket.disableWasCalled());
}

void verifyCodeRedAbort() {
  assertEquals(false, mockRocket.launchWasCalled());
  assertEquals(true, mockRocket.disableWasCalled());
}

• For multiple tests with duplicate assertions, the duplicated assertions may be moved to a helper method in a mock class
class MockRocket implements Rocket {
    private boolean launchWasCalled = false;
    private boolean disabledWasCalled = false;

    @Override
    void launch() {
        launchWasCalled = true;
    }

    boolean launchWasCalled() {
        return launchWasCalled;
    }

    boolean disabledWasCalled() {
        return disabledWasCalled;
    }

    void verifyCodeRedAbort() {
        assertEquals(false, launchWasCalled);
        assertEquals(true, disabledWasCalled);
    }
}

The only method that calls launchWasCalled() and disabledWas Called() -- refactor MockRocket
```java
class MockRocket implements Rocket {
    private boolean launchWasCalled = false;
    private boolean disabledWasCalled = false;

    @Override
    void launch() {
        launchWasCalled = true;
    }

    @Override
    void disable () {
        disableWasCalled = true;
    }

    void verifyCodeRedAbort() {
        assertEquals(false, launchWasCalled);
        assertEquals(true, disableWasCalled);
    }
}
```
@Test
void givenExpiredLaunchCode_RocketNotLaunched() {
    MockRocket mockRocket = new MockRocket();
    launchRocket(mockRocket, expiredCode);
    assertEquals(false, mockRocket.launchWasCalled());
    assertEquals(true, mockRocket.disableWasCalled());
}

@Test
void givenUnencryptedLaunchCode_RocketNotLaunched() {
    MockRocket mockRocket = new MockRocket();
    launchRocket(mockRocket, unencryptedCode);
    assertEquals(false, mockRocket.launchWasCalled());
    assertEquals(true, mockRocket.disableWasCalled());
}

- The tests can no longer interrogate the mock through its public interface. They can only verify that a code red abort happened.
Mock (5)

@Test
void givenExpiredLaunchCode_RocketNotLaunched() {
    MockRocket mockRocket = new MockRocket();
    launchRocket(mockRocket, expiredCode);
    mockRocket.verifyCodeRedAbort();
}

@Test
void givenUnencryptedLaunchCode_RocketNotLaunched() {
    MockRocket mockRocket = new MockRocket();
    launchRocket(mockRocket, unencryptedCode);
    mockRocket.verifyCodeRedAbort();
}

• Notice the duplicate set up, refactor the tests
Refactor a spy to a mock
Refactor the code to clean up and remove code smells (extract and move)
(+): Decrease duplication, centralize the assertions, increase maintainability
(-): To understand the tests, must inspect the mock
Summary

• Test doubles serve various purposes including:
  • Indirect input provisioning
  • Recording of indirect output
  • Immediate verification of interactions

• Fake it till you make it
  • Fast or independent from environmental influences

• Verify behavior with mocks

• Increase efficiency with mock frameworks
  • EasyMock (http://easymock.org/) – used in Koskela book
  • Jmock (http://www.jmock.org/)
  • Mockito (http://site.mockito.org/) – popular