Part 11 - Test doubles

Dimitri Merejkowsky

EFREI 2022-2023

Test Doubles

What is a test double

A class that will be used *only* for testing.

Types of doubles

- Dummy
- Stub
- Spy
- Mock
- Fake

Note: unfortunately, many people use those terms to mean more or less the same thing.

Still a good idea to know the various types that exist.

Let's see some examples

Dummy

Dummy (1)

```
interface Session {
  Date lastLoginTime();
}


class Report {
  String generate(Date date, Session session) {
    if (date.inTheFuture()) {
      throw new InvalidDateException();
    }

    Date loginTime = session.lastLoginTime();
    Duration timeSinceLastLogin = date - loginTime;
    // ...
  }
}
```

Dummy (2)

We are writing a sad path test to check we throw the correct exception when the date is in the future.

Note that we need a session object to call the method but we never use any of its methods during the test!

Dummy (3)

```
class DummySession implements Session {
  @Override
  Date lastLoginTime()  {
     return null;
  }
}

class ReportTests {
  @Test
  public void throw_on_invalid_date() {
    var inTwoDays = Date.now().shiftDays(2);
    var session = new DummySession();
    var report = new Report();
    assertThrows(InvalidDateException.class, () -> {
        report.generate(inTwoDays, session);
    });
  }
}
```

Dummy (4)

A dummy is a test double which is only used so that the code compiles.

Its methods are never called (hence they often return null).

Stub

Stub (1)

```
interface Authenticator {
  boolean login(String username, String password);
}


class LoginController {
  private final Authenticator authenticator;

  public LoginController(Authenticator authenticator) {
    this.authenticator = authenticator;
  }

  // ...
}
```

Stub (2)

```
class LoginController {
  // ...
  HttpResponse generateResponse() {
    boolean loginSuccess = authenticator.login(
      username, password
    );
    if (loginSuccess) {
      return new HttpResponse(200, ...);
    } else {
      return new HttpResponse(403, ...);
    }
  }
}
```

We want to test `generateResponse()`.

Stubs (3)

```
class RejectingAuthenticator implements Authenticator {
  @Override
  boolean login(String username, String password) {
     return false;
  }
}

class AcceptingAuthenticator implements Authenticator {
  @Override
  boolean login(String username, String password) {
     return true;
  }
}
```

Those are stubs because the **return value** matters

Stubs (4)

```
@Test
void generate_403_error_if_auth_fails() {
  var authenticator = new RejectingAuthenticator();
  var controller = new LoginController(authenticator);

  var response = controller.generateResponse()

  assertEquals(403, response.statusCode());
}
```

Stubs (5)

```
@Test
void test_generate_200_status_if_auth_succeeds() {
  var authenticator = new AcceptingAuthenticator();
  var controller = new LoginController(authenticator)

  var response = controller.generateResponse()

  assertEquals(200, response.statusCode());
}
```

Spy

A test double that *records* what happened.

Spy (1)

```
record Attempt(String username, String password) {}

class SpyAuthenticator implements Authenticator {
  private final List<Attempt> attempts;

  public SpyAuthenticator() {
    attempts = new ArrayList<>();
  }

  @Override
  public boolean login(String username, String password) {
    // ↓ here
    attempts.add(new Attempt(username, password));
    return false;
  }
}
```

Spy (2)

Use it:

```
@Test
void generate_403_error_if_auth_fails() {
  var authenticator = new SpyAuthenticator();
  var controller = new LoginController(authenticator);

  var response = controller.generateResponse();

  assertEquals(403, response.statusCode());
  // New!
  var expected = List.of(
    new Attempt("username", "password")
  );
  assertEquals(expected, spyAuthenticator.getAttempts());
```

Mock

A test double that *knows what should happen*.

Mock (1)

```
class MockAuthenticator implements Authenticator {
  boolean returnValue;
  Attempt expected;
  Attempt actual;

  MockAuthenticator calledWith(
    String username,
    String password
  ) {
    expected = new Attempt(username, password);
    return this;
  }

  void willReturn(boolean value) {
    returnValue = value;
  }

  // ...
```

Mock (2)

```
class MockAuthenticator implements Authenticator {

  boolean login(String username, String password) {
    actual = new Attempt(username, password);
    assertEquals(expected, actual);
    return returnValue;
  }
}
```

Mock (3)

```
@Test
void generate_a_403_error_if_auth_fails() {
  var mockAuthenticator = new MockAuthenticator();
  mockAuthenticator
    .calledWith("login", "bad pass")
    .willReturn(false);

  var controller = new LoginController(authenticator);

  var response = controller.generateResponse();

  assertEquals(403, response.statusCode());
}
```

Fake

Fake (1)

Replicates some logic of the production class

```
class FakeAuthenticator implements Authenticator {
  @Override
  public boolean login(String username, String password) {
    return !username.startsWith("locked-");
  }
}
```

Fake (2)

Use it:

```
@Test
void return_403_for_locked_users() {
  var authenticator = FakeAuthenticator()
  var controller = new LoginController(authenticator);

  var response = controller.generateResponse(
    "locked-user",
    "password"
  )

  assertEquals(403, response.statusCode());
}
```

Fake (3)

- warning: grows with the rest of the code
- but sometimes you need them for integration tests
- so you need to TDD your Fakes!

Mock library

Sometimes it's better to use a "mock library"

Mock library (1)

```java
import static org.mockito.Mockito.*;

@Test
void using_mockito() {
  Authenticator mockAuth = mock(Authenticator.class);
  when(mockAuth.login("username", "password"))
    .thenReturn(true);

  var controller = new LoginController(mockAuth);
  // ..
  verify(mockAuth).login("username", "password");
}
```

Recap

- dummy - almost empty
- stub - return value means something
- spy - remember what happens
- mock - knows what should happen
- fake - replicate production code logic