



Unit tests and mock objects

UA/DETI/TQS

Ilídio Oliveira (ico@ua.pt)

v2019-02-19.

Learning objectives

Explain the meaning of the “tests in isolation” principle

Distinguish between mocks, stubs and proxies

For a given test case, identify which services should be mocked

Read and write simple unit tests using JUnit and EasyMock



Unit testing tests units in isolation

Unit tests verify the “local” contract

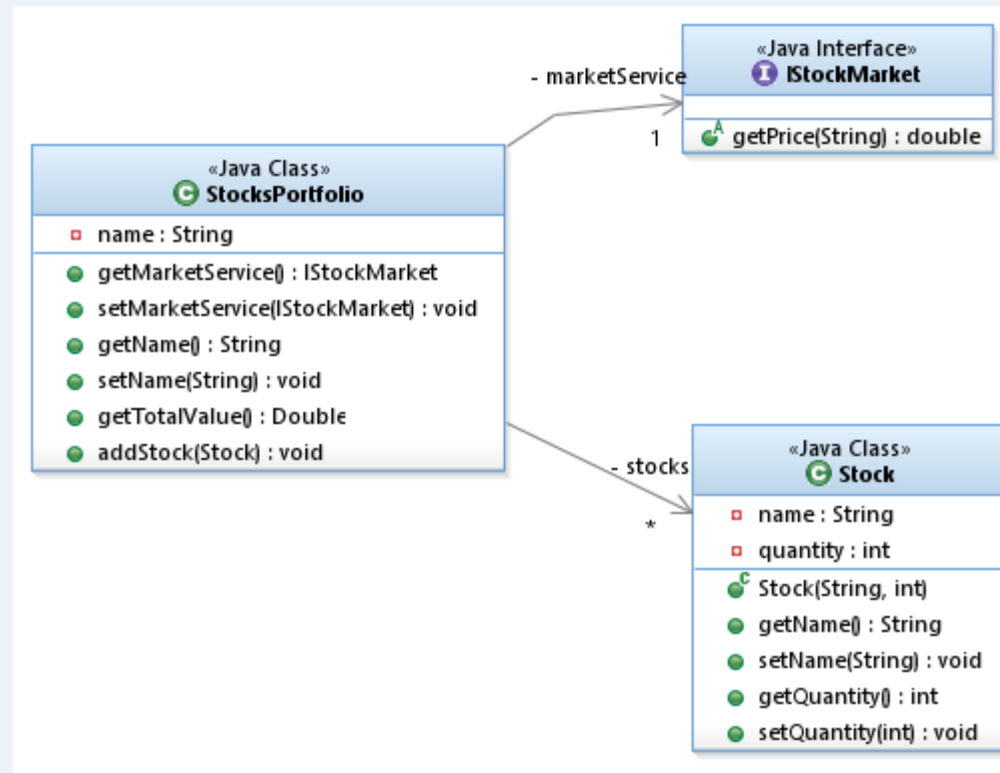
- ▶ StockPortfolio add/remove/find...

How about testing objects that REQUIRE collaboration from others?

- ▶ StockPortfolio queries StockMarket to update rates

To keep the isolation: “fake” the behavior of the remote/real object/service

- ▶ Just enough logic to get the tests done



Some StockPortfolio's methods depend on external services



Faking remote object behavior strategies

Stubs

- ▶ provide canned answers to calls made during the test
- ▶ usually not responding at all to anything outside what's programmed in for the test

Mocks

- ▶ objects pre-programmed with expectations. i.e. specification of the interactions they are expected to receive.
- ▶ verification will compare calls received against expectations

In-container testing

- ▶ containers are activated (by the test runner) to enable the testing environment
- ▶ requires mechanisms to deploy and execute tests in a container



Stubs approach: drawbacks

Stubs require implementing the same logic as the systems they are replacing

- ▶ often complex... stubs themselves need debugging!
- ▶ not very refactoring-friendly
- ▶ Stubs don't lend themselves well to fine-grained unit testing.

use stubs to replace a full-blown external system

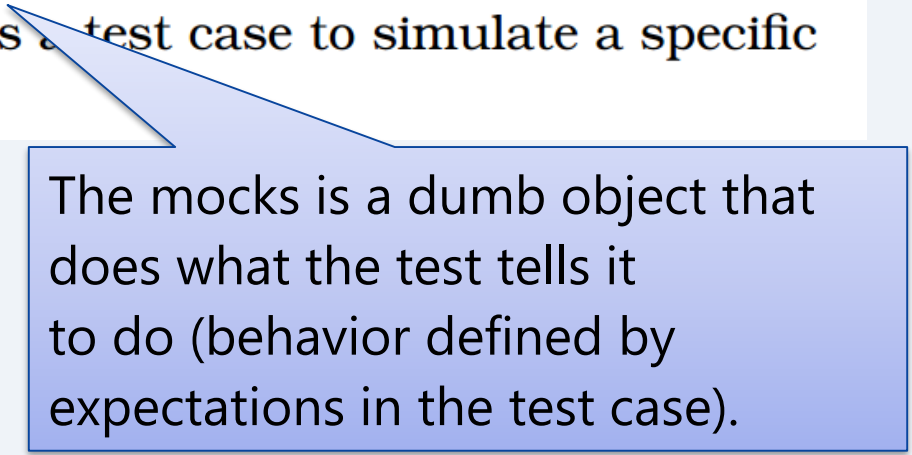
- ▶ a file system, a connection to a server, a database, ...

DEFINITION A *stub* is a piece of code that's inserted at runtime in place of the real code, in order to isolate the caller from the real implementation. The intent is to replace a complex behavior with a simpler one that allows independent testing of some part of the real code.



Mock objects approach

The **mock object** (or simply the *mock*) is a test double. It allows a test case to describe the calls expected from one module to another. During test execution the mock checks that all calls happen with the right parameters and in the right order. The mock can also be instructed to return specific values in proper sequence to the code under test. A mock is not a simulator, but it allows a test case to simulate a specific scenario or sequence of events.¹



The mock is a dumb object that does what the test tells it to do (behavior defined by expectations in the test case).

DEFINITION *Expectation*—When we're talking about mock objects, an *expectation* is a feature built into the mock that verifies whether the external class calling this mock has the correct behavior. For example, a database connection mock could verify that the `close` method on the connection is called exactly once during any test that involves code using this mock.

For example, Figure 2 depicts a test of object A. To fulfil the needs of A, we discover that it needs a service S. While testing A we mock the responsibilities of S without defining a concrete implementation.

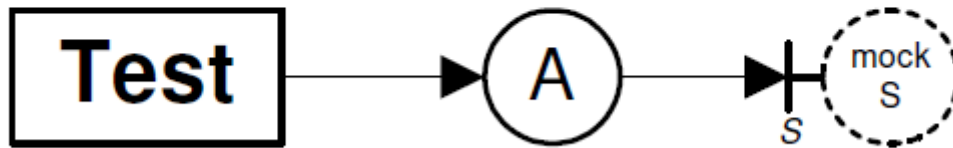


Figure 2. Interface Discovery

Once we have implemented A to satisfy its requirements we can switch focus and implement an object that performs the role of S. This is shown as object B in Figure 3. This process will then discover services required by B, which we again mock out until we have finished our implementation of B.

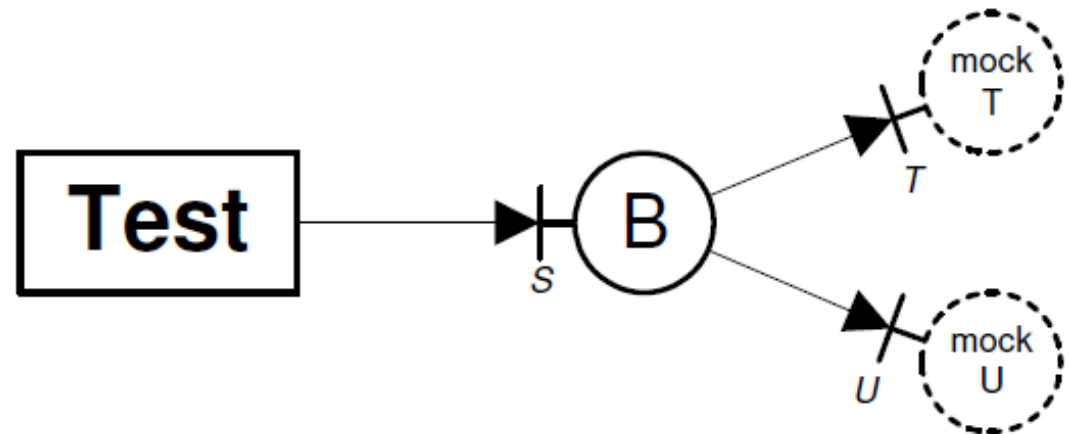


Figure 3. Iterative Interface Discovery

We continue this process until we reach a layer that implements real functionality in terms of the system runtime or external libraries.



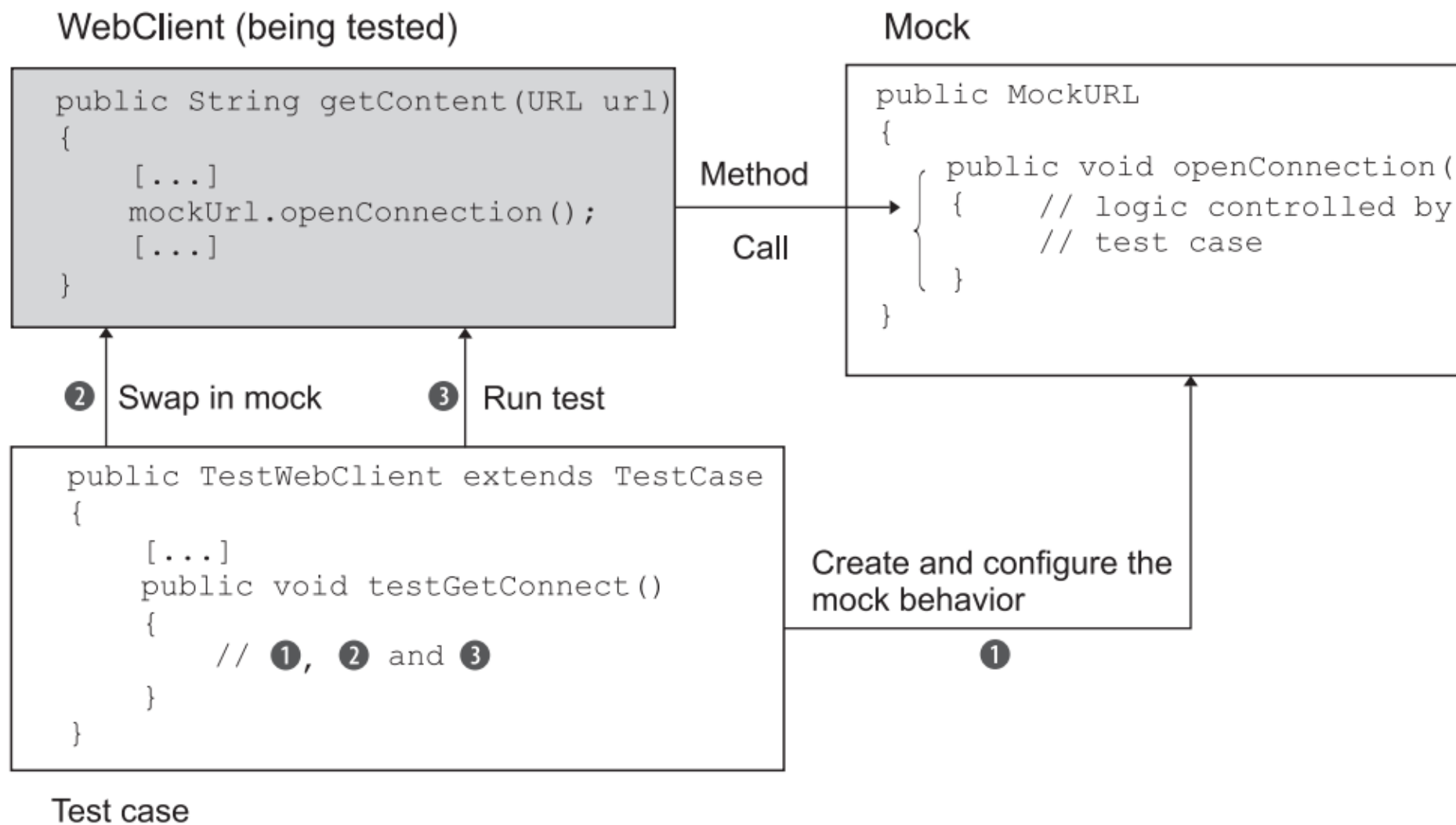


Figure 7.3 The steps involved in a test using mock objects



Stubs

Explicitly implements a simplified version of the target object behavior

- ▶ Contains some business logic

Usually coarse

Mocks

Provides a generated object to respond to part of the target object's contract

- ▶ No explicit implementation
- ▶ Behavior specified by expectations

Fine-grained



Fine grained

Test in isolation

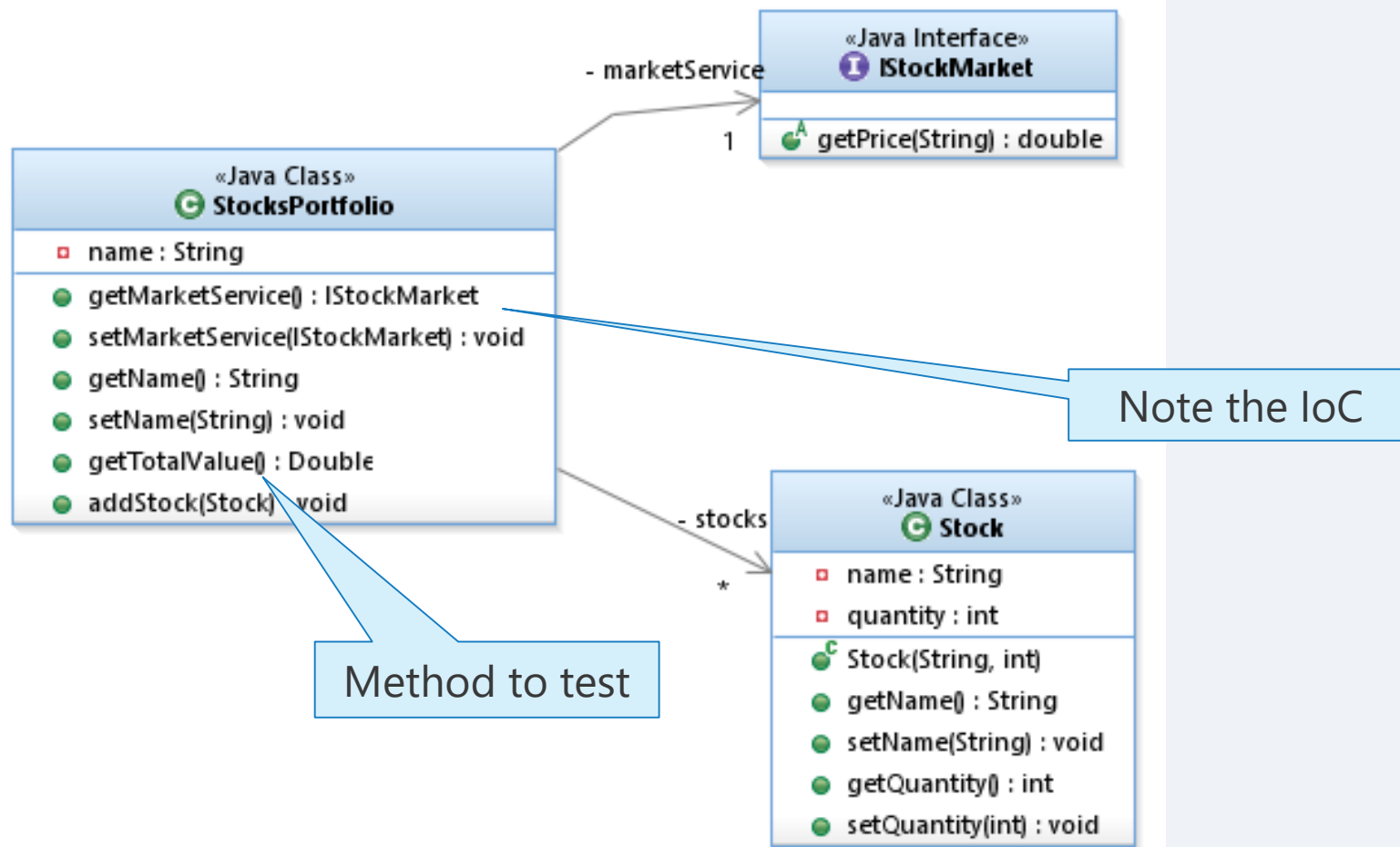
- ▶ can test even if implementations are not yet available (as long the contract is known)
- ▶ don't need to wait for all the parts to test

Fine-grained tests

- ▶ method level
- ▶ precise messages that pinpoint the cause of the breakage.



Example



→ see also: [DeveloperZone](#)



Pattern to help with testing

Design patterns in action: Inversion of Control

Applying the IoC pattern to a class means removing the creation of all object instances for which this class isn't directly responsible and passing any needed instances instead. The instances may be passed using a specific constructor, using a setter, or as parameters of the methods needing them. It becomes the responsibility of the calling code to correctly set these domain objects on the called class.²

One last point to note is that if you write your test first, you'll automatically design your code to be flexible. Flexibility is a key point when writing a unit test. If you test first, you won't incur the cost of refactoring your code for flexibility later.



EasyMock workflow

1. create a mock object, given an interface
2. set expectations (specification of the expected calls)
 1. methods used, parameters, return values
 2. order (and number of times) of methods call
3. publish the specification (EasyMock.replay)
4. verify that expectations were met (EasyMock.verify)



EasyMock expectations

```
EasyMock.expect(mock.getPrice("EBAY")).andReturn(42.00);
```

```
EasyMock.expect(mock.getRate(  
    (String) EasyMock.matches(" [A-Z] [A-Z] [A-Z]"),  
    (String) EasyMock.matches(" [A-Z] [A-Z] [A-Z]"))) andReturn(1.5);
```

```
EasyMock.expect(mock.getRate("USD", "EUR")).andThrow(new  
IOException());
```

```
EasyMock.expect(mock.getPrice("EBAY")).andReturn(42.00).times(3);
```

<http://easymock.org/user-guide.html#verification-calls>



EasyMock expectations – order of calls

Normal — `EasyMock.createMock()`

- ▶ All of the expected methods must be called with the specified arguments; the order does not matter. Calls to unexpected methods cause test failure.

Strict — `EasyMock.createStrictMock()`

- ▶ All expected methods must be called with the expected arguments, in a specified order. Calls to unexpected methods cause test failure.

Nice — `EasyMock.createNiceMock()`

- ▶ All expected methods must be called with the specified arguments in any order. Calls to unexpected methods do not cause the test to fail. Nice mocks supply reasonable defaults for methods you don't explicitly mock

<http://easymock.org/user-guide.html#mocking-strict>



Use dependency injection

```
@RunWith(EasyMockRunner.class)
public class MathApplicationTester {
```

```
    // @TestSubject annotation is used to identify class which is going to use the mock
```

```
    @TestSubject
```

```
    MathApplication mathApplication = new MathApplication();
```

```
    // @Mock annotation is used to create the mock object to be injected
```

```
    @Mock
```

```
    CalculatorService calcService;
```

```
    @Test
```

```
    public void testAdd(){
```

```
        // add the behavior of calc service to add two numbers
```

```
        EasyMock.expect(calcService.add(10.0,20.0)).andReturn(30.00);
```

```
        //activate the mock
```

```
        EasyMock.replay(calcService);
```

```
        // test the add functionality
```

```
        Assert.assertEquals(mathApplication.add(10.0, 20.0),30.0,0);
```

```
    }
```

```
}
```

```
public class MathApplication {
    private CalculatorService calcService;

    public void setCalculatorService(CalculatorService calcService){
        this.calcService = calcService;
    }

    public double add(double input1, double input2){
        return calcService.add(input1, input2);
    }
}
```



When to use a mock object?

supplies non-deterministic results

- ▶ e.g. the current time or the current temperature.

has states that are difficult to create or reproduce

- ▶ e.g. a network error or database error.

is slow

- ▶ e.g. a large network resource.

does not yet exist (test driven development) or may change behavior;

would have to include information and methods exclusively for testing purposes.



References

P. Tahchiev, F. Leme, V. Massol, and G. Gregory, JUnit in Action, Second Edition. Manning Publications, 2010.

Langr, J., Hunt, A. and Thomas, D., 2015. *Pragmatic Unit Testing in Java 8 with JUnit*. Pragmatic Bookshelf.

