Lab 4.5 - Test Doubles (Mocking)

# Introduction

When a component depends on (makes use of) other components, it can be more difficult to unit test that component. This is because we need to isolate the component under test from the other components, but at the same time need to be able to control how those other components appear to behave when our main component calls them. We may want to change how the dependent component behaves for each test.

Setting this kind of test up by hand is possible, but very laborious and error-prone. To help, there are frameworks available that can help with setting up "mock" versions of dependent components (also known as "test doubles").

# Goals

During this lab you will:

1. Use a class with a dependency on class that does not exist yet
2. Create a new class to mimic the dependency (our "mock") class
3. Perform dependency injection, so that the class under test makes use of an object made from the mocked dependency, which is therefore under the control of the test

The exact process for doing this varies depending on whether you are completing the exercise in C#, Java or JavaScript.

# Starter project

There is a separate starter project for Java, C# and JavaScript. Open the relevant starter project, then follow the specific instructions below for your chosen language.

|  |
| --- |
| Java instructions |
| Add a dependency on EasyMock Edit the Maven **pom.xml** file, and add this child element to the **<dependencies>** section:  <dependency>  <groupId>org.easymock</groupId>  <artifactId>easymock</artifactId>  <version>4.2</version>  <scope>test</scope>  </dependency>  Save the changes to the **pom.xml** file back to disk!  Right click on the project, and choose **Maven > Update project...** Create unit tests Within the **src/main/test** folder, create a new JUnit Test Case, as follows:  **import** org.junit.Before;  **import** **static** org.easymock.EasyMock.*createNiceMock*;  **import** **static** org.easymock.EasyMock.*expect*;  **import** **static** org.easymock.EasyMock.*replay*;  **import** **static** org.easymock.EasyMock.*verify*;  **import** **static** org.junit.Assert.*assertEquals*;  **import** **static** org.junit.Assert.*fail*;  **import** org.junit.Before;  **import** org.junit.Test;  **import** com.qa.testing.easymock.ICalcMethod;  **import** com.qa.testing.easymock.IncomeCalculator;  **import** com.qa.testing.easymock.Position;  **public** **class** IncomeCalculatorTest {  **private** ICalcMethod calcMethod;  **private** IncomeCalculator calc;  @Before  **public** **void** setUp() **throws** Exception {  // NiceMocks return default values for unimplemented methods  calcMethod = *createNiceMock*(ICalcMethod.**class**);  calc = **new** IncomeCalculator();  }    @Test  **public** **void** testCalc1() {  // Setting up the expected value of the method call calc  *expect*(calcMethod.calc(Position.***BOSS***)).andReturn(70000.0).times(2);  *expect*(calcMethod.calc(Position.***PROGRAMMER***)).andReturn(50000.0);  // Setup is finished need to activate the mock  *replay*(calcMethod);  calc.setCalcMethod(calcMethod);  **try** {  calc.calc();  *fail*("Exception did not occur");  } **catch** (RuntimeException e) {  }  calc.setPosition(Position.***BOSS***);  *assertEquals*(70000.0, calc.calc(), 0);  *assertEquals*(70000.0, calc.calc(), 0);  calc.setPosition(Position.***PROGRAMMER***);  *assertEquals*(50000.0, calc.calc(), 0);  calc.setPosition(Position.***SURFER***);  *verify*(calcMethod);  }    @Test(expected = RuntimeException.**class**)  **public** **void** testNoCalc() {  calc.setPosition(Position.***SURFER***);  calc.calc();  }    @Test(expected = RuntimeException.**class**)  **public** **void** testNoPosition() {  *expect*(calcMethod.calc(Position.***BOSS***)).andReturn(70000.0);  *replay*(calcMethod);  calc.setCalcMethod(calcMethod);  calc.calc();  }    @Test(expected = RuntimeException.**class**)  **public** **void** testCalc2() {  // Setting up the expected value of the method call calc  *expect*(calcMethod.calc(Position.***SURFER***)).andThrow(**new** RuntimeException("Don't know this guy")).times(1);  // Setup is finished need to activate the mock  *replay*(calcMethod);  calc.setPosition(Position.***SURFER***);  calc.setCalcMethod(calcMethod);  calc.calc();  }  } Run the tests You should see that we have successfully been able to unit test the calculator class, with each test taking control over how the dependent object (which implements the interface ICalcMethod) behaves. |

|  |
| --- |
| C# instructions |
| Add a dependency on FakeItEasy Right-click on the **QACalculatorTests** project, and select **Manage NuGet Packages...**  Browse for the package **FakeItEasy**, and click to install it. Create unit tests Within the **QACalculatorTests** project, create a new test class in the file **IncomeCalculatorTests.cs**, as follows:  using System;  using FakeItEasy;  using NUnit.Framework;  using QACalculator;  namespace QACalculatorTests  {  class IncomeCalculatorTests  {  private ICalcMethod calcMethod;  private IncomeCalculator calc;  [SetUp]  public void SetUp()  {  // Make a fake implementation of ICalcMethod which returns default values  calcMethod = A.Fake<ICalcMethod>();  // Make an instance of the real class under test  calc = new IncomeCalculator();  }  [Test]  public void TestCalc1()  {  // Configure the ICalcMethod to behave how we want  A.CallTo(() => calcMethod.Calc(Position.BOSS)).Returns(70000.0);  A.CallTo(() => calcMethod.Calc(Position.PROGRAMMER)).Returns(50000.0);  // Set the calc method  calc.SetCalcMethod(calcMethod);  calc.SetPosition(Position.BOSS);  double income1 = calc.Calc();  Assert.AreEqual(70000.0, income1);  double income2 = calc.Calc();  Assert.AreEqual(70000.0, income2);  calc.SetPosition(Position.PROGRAMMER);  double income3 = calc.Calc();  Assert.AreEqual(50000.0, income3);  // Verify that the ICalcMethod.Calc method was called the right number of times  A.CallTo(() => calcMethod.Calc(Position.BOSS)).MustHaveHappened(2, Times.Exactly);  A.CallTo(() => calcMethod.Calc(Position.PROGRAMMER)).MustHaveHappenedOnceExactly();  }  [Test]  public void TestNoCalc()  {  // Set the position  calc.SetPosition(Position.SURFER);  // Calling Calc before setting the CalcMethod should throw exception  Assert.Throws<Exception>(() => calc.Calc(), "CalcMethod not yet maintained");  }  [Test]  public void TestNoPosition()  {  // Set the calc method  calc.SetCalcMethod(calcMethod);  // Calling Calc before setting the position should throw exception  Assert.Throws<Exception>(() => calc.Calc(), "Position not yet maintained");  }  [Test]  public void TestCalc2()  {  // Set up a fake ICalcMethod which throws an exception  // when asked for the income of a SURFER  A.CallTo(() => calcMethod.Calc(Position.SURFER)).Throws(new Exception("Don't know this guy!"));  // Set up the calc method and position  calc.SetCalcMethod(calcMethod);  calc.SetPosition(Position.SURFER);  // If the ICalcMethod.calc method throws an exception, then it should  // continue to bubble up out fo the IncomeCalculator.Calc method  Assert.Throws<Exception>(() => calc.Calc(), "Don't know this guy!");  }  }  } Run the tests You should see that we have successfully been able to unit test the calculator class, with each test taking control over how the dependent object (which implements the interface ICalcMethod) behaves. |

|  |
| --- |
| JavaScript instructions |
| Install mocha, chai and sinon The first two packages allow us to do unit testing in general, and the last one allow us to work with test doubles (mocks / stubs).  **npm install --save-dev mocha chai sinon** Create unit tests Create a new JavaScript file called **IncomeCalculator.test.js**, with the following content:  const { expect } = require('chai');  const sinon = require('sinon');  const { Position, ICalcMethod, IncomeCalculator } = require('./IncomeCalculator');  describe('IncomeCalculator.calc()', function () {      let calc;      let calcMethod;      beforeEach(() => {          calcMethod = sinon.stub(new ICalcMethod());          calc = new IncomeCalculator();      });      it('calculates income using ICalcMethod object', function () {          // Mock the ICalcMethod to return specific incomes          calcMethod.calc.withArgs(Position.BOSS).returns(70000);          calcMethod.calc.withArgs(Position.PROGRAMMER).returns(50000);          // Configure the ICalcMethod object          calc.setCalcMethod(calcMethod);          calc.setPosition(Position.BOSS);          const income1 = calc.calc();          expect(income1).to.equal(70000);          calc.setPosition(Position.BOSS);          const income2 = calc.calc();          expect(income2).to.equal(70000);          calc.setPosition(Position.PROGRAMMER);          const income3 = calc.calc();          expect(income3).to.equal(50000);          // Verify that ICalcMethod.calc was called expected number of times          sinon.assert.callCount(calcMethod.calc, 3);      });      it('throws exception if called without calc method set', function () {          calc.setPosition(Position.BOSS);          expect(() => calc.calc()).to.throw('Calc method not yet maintained');          sinon.assert.notCalled(calcMethod.calc);      });      it('throws exception if called without position set', function () {          calc.setCalcMethod(calcMethod);          expect(() => calc.calc()).to.throw('Position not yet maintained');          sinon.assert.notCalled(calcMethod.calc);      });      it('throws exception if ICalcMethod.calc throws exception', function () {          calcMethod.calc.withArgs(Position.SURFER).throws(new Error("Don't know this guy!"));          calc.setCalcMethod(calcMethod);          calc.setPosition(Position.SURFER);          expect(() => calc.calc()).to.throw("Don't know this guy!");      });  }); Run the tests Add an NPM task to the **package.json** file to use mocha to run the tests:  **scripts: {**  **"test": "mocha ."**  **}**  Run the tests using:  **npm run test**  You should see that we have successfully been able to unit test the calculator class, with each test taking control over how the dependent object (which implements the interface ICalcMethod) behaves. |