Unit testing and mocking

Ronnie Holm

June 15, 2009

Overview

- No preching
- Introduction to terminology
- Examples

Example of a unit test

- Easy to test method with no side effects

```
public class Calculator {
    public double ComputeProduct(double a, double b) {
        return a * b;
    }
}

[TestClass]
public class CalculatorTest {
    [TestMethod]
    public void Should_compute_product() {
        Calculator c = new Calculator();
        Assert.AreEqual(16.3, c.ComputeProduct(3.14156, 5.2), 0.1);
    }
}
```

Unit test, meet real world

- Methods delegate to other methods and objects
- Unit test framework affects code under test by providing input and observing output
- To test one and only one property at a time, external dependencies of code under test must be controlled
- Unit test framework alone is insufficient for testing but the simplest of methods
- Unit test framework is not only for unit testing

A unit test challenge

```
public class Calculator1 {
    public double ComputeProduct(double a, double b) {
        return Math.Round(a * b, GetPrecision());
    }

    public int GetPrecision()
    {
        return Int32.Parse(
            System.Configuration.
                ConfigurationSettings.AppSettings["Precision"]);
    }
}
```

- Code takes external dependency on config setting
- Side effect that cannot be abstracted away using a unit test framework
- Other examples
  - Retrieve url of the web
  - Execute a process
  - Other objects in general

Inversion of Control

- Dependency injection is one way of IoC
- Dependency injected into object under test
- During unit testing, a fake dependency that implements the interface of the real dependecy is injected
- Unit test controls dependency and hence controls object
- Fake dependencies are easily implemented by hand, but it's repetitive and tedious
- Therefore, people use a mocking framework like Rhino Mocks or TypeMock

Unit test challenge: solution

```
public class Calculator2 {
    private IConfigReader _configReader;

    public Calculator2(IConfigReader configReader) {
        _configReader = configReader;
    }

    public double ComputeProduct(double a, double b) {
        return Math.Round(a * b, GetPrecision());
    }

    public int GetPrecision() {
        return Int32.Parse(_configReader.GetSetting("Precision"));
    }
}

public interface IConfigReader {
    string GetSetting(string key);
}

public class ConfigReader : IConfigReader {
    public string GetSetting(string key) {
        return System.Configuration.
            ConfigurationSettings.AppSettings[key];
    }
}
```

```
public class FakeConfigReader : IConfigReader {
    public string GetSetting(string key) {
        return key == "Precision" ? "2".ToString() : null;
    }
}

[TestClass]
public class Calculator2Test {
    [TestMethod]
    public void Should_compute_product() {
        IConfigReader fakeConfigReader = new FakeConfigReader();
        Calculator2 c = new Calculator2(fakeConfigReader);
        Assert.AreEqual(16.34, c.ComputeProduct(3.14156, 5.2));
    }
}
```

How mocking works

- FakeConfigReader is a hand-written mock
- FakeConfigReader is what Rhino Mocks or TypeMock automatically generates
- Rhino Mocks emits MSIL implementing IConfigReader. In memory an assembly is created and compiled, and subsequently loaded into the MSTest AppDomain
- TypeMock intercepts calls by hooking into the CLR and emitting MSIL as it intercepts calls
- Whatever approach, the unit test looks the same

Mocking by TypeMock, take 1

```
[TestClass]
public class Calculator2TestWithTypeMock {
    [TestMethod, Isolated]
    public void Should_compute_product() {
        // arrange
        IConfigReader fakeConfigReader = Isolate.Fake.Instance<IConfigReader>();
        Isolate.WhenCalled(() => fakeConfigReader.GetSetting("Precision")).WillReturn("2");
        Calculator2 c = new Calculator2(fakeConfigReader);

        // act
        double d = c.ComputeProduct(3.14156, 5.2);

        // assert
        Isolate.Verify.WasCalledWithExactArguments(() => fakeConfigReader.GetSetting("Precision"));
        Assert.AreEqual(16.34, d);
    }
}
```

Mocking by TypeMock, take 2

```
[TestClass]
public class Calculator2TestWithTypeMock1 {
    [TestMethod, Isolated]
    public void Should_compute_product() {
        // arrange
        Calculator1 calculator = Isolate.Fake.Instance<Calculator1>(Members.CallOriginal);
        Isolate.WhenCalled(() => calculator.GetPrecision()).WillReturn(2);

        // act
        double d = calculator.ComputeProduct(3.14156, 5.2);

        // assert
        Isolate.Verify.WasCalledWithExactArguments(() => calculator.GetPrecision());
        Assert.AreEqual(16.34, d);
    }
}
```

Properties of a good unit test

- Resist urge to morph unit test into integration test
- Use clever naming of test cases
- No more than 10-15 lines of code
- Verify one and only one property of code under test
- Idempotence property of utmost importance
- Individual tests run in milliseconds
- Test suite runs in a couple of seconds

Conclusion

- Unit testing and mocking doesn't have to be hard
- Even legacy code can be unit tested
- Red-Green refactoring is widely popular in some form
- Unit testing is as much about design as test