Test Doubles and Mocking

DEVELOPMENTOR
DEVELOPING PEOPLE WHO DEVELOP SOFTWARE

Agenda

- Design to Interface and Testing
- Test Doubles
- Mocking Frameworks

Design to Interface and Testing

- Designing to interface decouples code
  - Allows the implementation to vary
  - Concrete implementation injected
- Can inject concrete types purely designed for testing
  - Designed to drive specific test conditions
  - Allow capture of "interesting" activity of SUT
- Concrete types that look like real types are known as Test Doubles

Different Kinds of Test Doubles

- There are a range of types of Test Doubles
  - Fakes
  - Stubs
  - Spies
  - Mocks
- Distinction is often blurred in practice
  - Spectrum of behavior

Fakes

- Dummy or heavily simplified version of abstraction
  - Useful when abstraction required but its behavior is not of interest

```
public class FakeLogger : ILogger
{
    public void Log(string
message)
    {
        // no-op
    }
}
```

Stubs

- Implementation is designed to return specific values
  - Drive test conditions in various ways

```
public class StubAccountRepository :
IAccountRepository
{
    public IEnumerable<Account>
GetOverdrawnAccounts()
    {
        return new[]
            {
                new Account(-300m),
                new Account(-600m),
                new Account(-100m),
            };
    }
}
```

Spies

- Give insight into behavior of SUT
  - Can record data passed from SUT

```
public class SpyLogger : ILogger
{
    public string Message { get; private
set; }

    public void Log(string message)
    {
        Message = message;
    }
}
```

Mocks

- Set expectations and verify the result
  - Similarities to Spies

```
public class MockLogger : ILogger{
    private int logCount, expected;

    public void Log(string 
message){
        logCount++;
    }
    public void 
ExpectedLogCount(int expected){
        this.expected = expected;
    }

    public bool Verify(){
        return logCount == 
expected;
    }
}
```

Where do Test Doubles Come From?

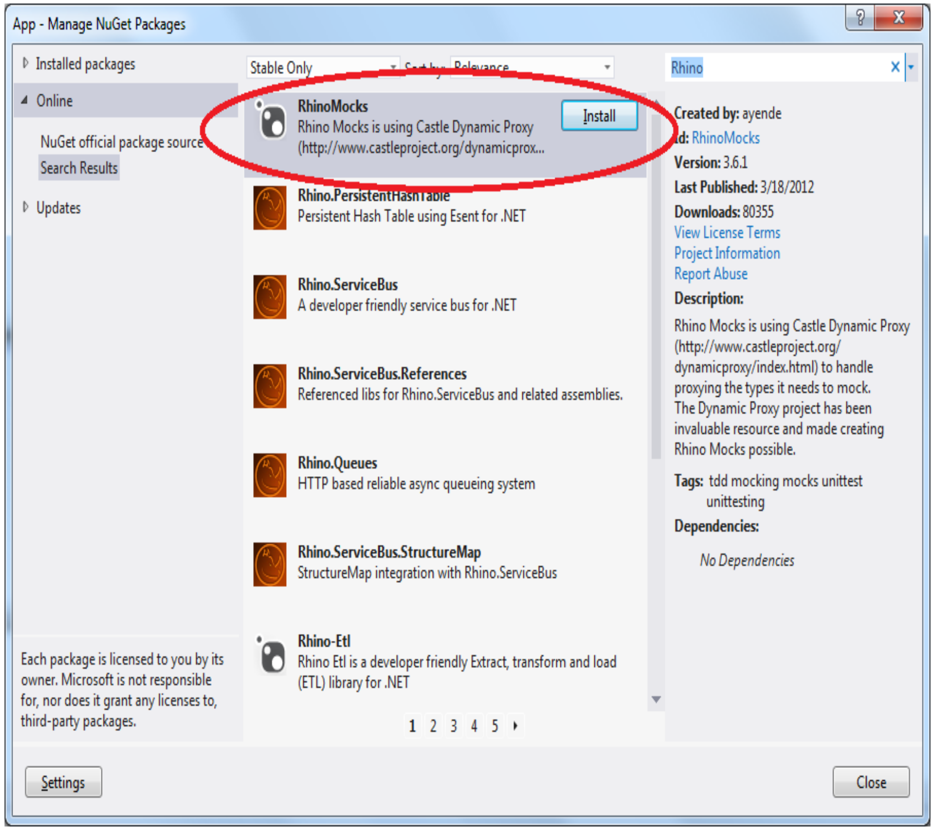
- Can roll own test doubles
  - Tuned to requirements
  - Have more code to maintain
- Can use a Mocking Framework
  - Generates necessary double on demand
  - Reduces your code base
  - Sometimes have to fight framework
- Nothing wrong with combining approaches as required

Mocking Frameworks

- Many Mocking Frameworks for .NET
  - Rhino Mocks
  - Moq
  - Nmock
  - TypeMock
  - EasyMock.NET
  - Microsoft Fakes (ships with VS Ultimate Edition)
- All do a similar job
  - Moq and Rhino most popular
  - TypeMock and Fakes provide extra functionality for legacy code

Rhino Mocks – an Example Mocking Framework

- Rhino Mocks open source framework from Ayende Rahien
  - Based on Castle Dynamic Proxy
  - Install as Nuget Package



RhinoMocks
Rhino Mocks is using Castle Dynamic Proxy (http://www.castleproject.org/dynamicprox...

Rhino.PersistentHashTable
Persistent Hash Table using Esent for .NET

Rhino.ServiceBus
A developer friendly service bus for .NET

Rhino.ServiceBus.References
Referenced libs for Rhino.ServiceBus and related assemblies.

Rhino.Queues
HTTP based reliable async queueing system

Rhino.ServiceBus.StructureMap
StructureMap integration with Rhino.ServiceBus

Rhino-Etl
Rhino Etl is a developer friendly Extract, transform and load (ETL) library for .NET

Created by: ayende
Id: RhinoMocks
Version: 3.6.1
Last Published: 3/18/2012
Downloads: 80355
View License Terms
Project Information
Report Abuse
Description:
Rhino Mocks is using Castle Dynamic Proxy (http://www.castleproject.org/dynamicproxy/index.html) to handle proxying the types it needs to mock. The Dynamic Proxy project has been invaluable resource and made creating Rhino Mocks possible.
Tags: tdd mocking mocks unittest unittesting
Dependencies:
No Dependencies

Each package is licensed to you by its owner. Microsoft is not responsible for, nor does it grant any licenses to, third-party packages.

Generating a stub

- Simple to generate a stub and decide method results on ad-hoc basis

```
IAccountRepository repo =
MockRepository.GenerateStub<IAccountRepository>();

repo.Stub(r => r.GetOverdrawnAccounts())
    .Return(new[]
            {
                new Account(-300m),
                new Account(-600m),
                new Account(-100m),
             });
```

Stubbing Based on Inputs

- Can stub multiple times base on input parameters

```
IAccountRepository repo =
MockRepository.GenerateStub<IAccountRepository>();

repo.Stub(r => r.GetRate(RateType.Current))
    .Return(0.02m);

repo.Stub(r => r.GetRate(RateType.HighInterest))
    .Return(0.1m);
```

Can Stub Ignoring Inputs

- Can generate a stub method that returns the same irrespective of inputs

```
IAccountRepository repo =
MockRepository.GenerateStub<IAccountRepository>();

repo.Stub(r => r.GetRate(RateType.Current))
    .IgnoreArguments()
    .Return(0.02m);
```

Stubs Can Throw Exceptions

- Need to test SUT if dependency throw exception
  - Can specify stub throw exception on invocation
  - Common to use with IgnoreArguments though not required

```
IAccountRepository repo =
MockRepository.GenerateStub<IAccountRepository>();

repo.Stub(r => r.GetRate(RateType.Current))
    .IgnoreArguments()
    .Throw(new ArgumentException());
```

Creating Mocks with Rhino Mocks

- Must decide how to respond to invocation
  - Whether to specify an implementation
- Must specify expectations
  - If a method is invoked
  - If how invoked is important
- Need to track state
  - Create an instance of MockRepository
- Different kinds of mocks
  - Dynamic Mock
  - Strict Mock
  - Partial Mock

Generating Mocks

- Use method on MockRepository instance
  - DynamicMock
  - StrictMock
  - PartialMock
- DynamicMock
  - Mock will do “default behavior” for non explicitly specified methods
- StrictMock
  - Mock will register failure for non explicitly specified methods
- PartialMock
  - Will allow mocking of abstract class, otherwise similar to DynamicMock

Setting Expectations

- Need to specify expected interaction
  - States invocation expected and what result (if any) to generate

```
Expect.Call(repo.GetRate(RateType.Current))
      .Return(0.02m);
```

Exercising Mock

- Need to demarcate set of operations that should create expectations
  - What is the SUT going to do to satisfy expectations
- Need to specify when expectations should be met
  - When expectations should all have been satisfied
- Rhino has evolved API – all work, your choice
  - Original Syntax
  - Using Syntax
  - Fluent Syntax

Mock Usage Example (Original API)

```
var mocks = new MockRepository();

IAccountRepository repo =
mocks.DynamicMock<IAccountRepository>();
var reporter = new Reporter(repo);

Expect.Call(repo.GetOverdrawnAccounts())
      .Return(new Account[]{});

mocks.ReplayAll();

reporter.Generate();

mocks.VerifyAll();
```

Interesting interaction must happen between these

Verifying APIs that Return Void

- Call to Expect must return something
  - Otherwise Return method doesn’t know what to do
- Overload which takes an Action delegate
  - Use when API doesn’t return anything

```
 Expect.Call(() =>
traceLogger.Log(null))
       .IgnoreArguments();
```

Summary

- Coding to abstraction decouples your code and can provide test versions
- Can roll own test doubles
- Mocking frameworks take the strain