Design Patterns for Testability

DEVELOPMENTOR

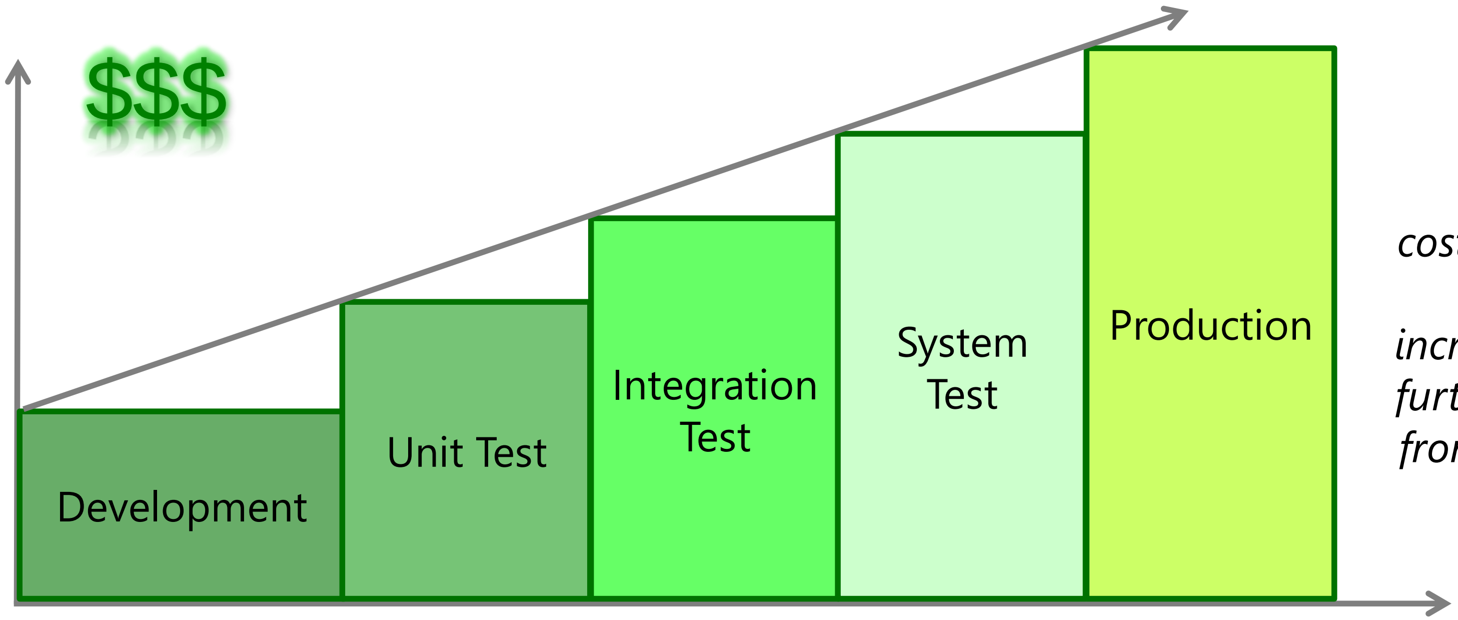
DEVELOPING PEOPLE WHO DEVELOP SOFTWARE

Testing [motivation]

- Testing is a necessary part of the development process – bugs in released products make us look bad

Unit Testing [motivation]

- Several good reasons to write unit tests
  - reduce bugs in new and existing features
  - can become part of the documentation
  - reduce the cost of change
  - improve design by forcing developers to think it through
  - defense against the dark arts.. er.. other programmers



cost of fixing bugs increases the further away from dev we get

FOUNDATIONS

SOLID principles (Bob Martin, 2003)

"*design principles found in highly testable code*"

- **S** **Single Responsibility** – there should never be more than one reason for a class to change
- **O** **Open/Closed Principle** – software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification
- **L** **Liskov Substitutability Principle** – subclasses should be suitable for their base classes
- **I** **Interface Segregation Principle** – prefer many client interfaces to one general purpose interface
- **D** **Dependency Inversion Principle** – depend upon abstraction, do not depend upon concretions

UNIT TESTING PRINCIPLES

Legacy Code

- What is the definition of Legacy Code?
- Code that relates to a no-longer supported or manufactured operating system or other computer technology
- Code inherited from someone else
- Code that exists for a reason that is no longer valid
- Code that you would rather replace than work with

- Code without [automated] tests - Michael Feathers (2004)

Why write tests?

- Specification
  - Intent of what you want the code to do
  - Skip the comments write easily readable code
- Feedback
  - Are you satisfying the intent with the code
- Regression
  - Making sure it continues to satisfy the intent
- Granularity
  - When something goes wrong, *where* it went wrong

What is a UnitTest?

- Michael Feathers wrote “A set of Unit Testing Rules”

A test is not a unit test if:

• It talks to the database
• It communicates across the network
• It touches the file system
• It can’t run at the same time as other tests
• You have to do special things in your environment (such as editing configuration files) to run it.

Tests that do these things aren’t bad. Often they are worth writing, and they can be written in a unit test harness. However, it is important to be able to separate them from true unit tests so that we can keep a set of tests that we can run fast whenever we make our changes

What is a Unit Test?

- Code *unit* : smallest testable application part
  - Usually one (or more related) class member(s)
- *Unit test* : strict written contract that code unit must satisfy
  - Ensures code unit meets design, satisfies requirements, behaves as intended
  - Asserts conditions to be met

Unit test requirements

- Isolated & focused
  - Only test the SUT & only one thing at a time
- Self-contained
  - Don’t rely on external information or complex configuration
- Independent of other tests
- Fast (else we won’t run them)
- Repeatable
- Well documented, self-evident & maintainable

Test-driven development (TDD)

- For each new feature write a test first
  - Define code requirements before writing code
  - Test based on use case
  - Acceptance test to test the big picture
  - Unit test to test the implementation
- Paradigm shift
  - From “what code must I write to solve this problem?”
  - To “how will I know when I’ve solved this problem”

Other Testing

- Acceptance testing
  - Drives the product features
  - Is an end-to-end test
- Including deployment
  - If the app does not deploy, how do you know if it works?

Unit testing

- Test simplest thing possible
  - simple != simplistic
  - initially testing for success is better than testing for failure - can see forward movement
- Test must be readable
  - code is write once read many (by developers)
  - write assuming the SUT is in place (TDD)
  - write the infrastructure to make the test fail the way you expect
- Always (always, always) write failing tests first
  - gives more confidence when test passes

What to test

- Test behaviors not methods
  - name tests appropriately - names such as **Test1** and **Test2** are not appropriate
- If the tests are hard to write…
  - the code is hard to test
  - this can indicate problems
  - coupling, cohesion

Visual Studio Team Test

- Integrated into Visual Studio
  - setup / teardown for all tests
  - setup / teardown for individual tests
  - methods are annotated to make them a test case
  - executed from within Visual Studio, or through external testing tool
- Visual Studio supports multiple testing frameworks
  - pick your favourite, but still drive through the UI and MSBuild

Writing a Test Class

- Test class contains test cases
  - attributes identify test classes and methods

```
using Microsoft.VisualStudio.TestTools.UnitTesting;

[TestClass]
public class TestToDoList
{
   [TestMethod]
   public void ATestToRun()
   {
      // Test code goes here
   }
```

What does a test look like

- Follow a naming convention
  - names get long but aid understanding
  - SUTName_WhatAmITesting_ExpectedOutcome (Scott disagrees)
- Remember, names are read mostly
  - we don't call the methods directly

```
public void ListConstructor_PageSizePassedIsZero_ShouldThrowException()
```

Testing fundamentals

- Arrange, Act, Assert
- Test one thing and one thing only
  - does not mean only one assert

```
[TestMethod]
public void ToDoListEqual_WhenIdAndNameAreTheSame_ReturnsTrue()
{
    // Arrange
    ToDoList list1 = new ToDoList("Name") { Id = 1 };
    ToDoList list2 = new ToDoList("Name") { Id = 1 };

    // Act
    bool result = list1.Equals(list2);

    // Assert
    Assert.IsTrue(result);
}
```

Using Asserts

- Write tests by asserting truths
  - use **`Assert`** class static methods
  - several methods are available
  - support different types through generics

```
[TestMethod]
public void ToDoListEquals_WhenIdAndNameAreTheSame_ReturnsTrue()
{
    ...

    Assert.IsTrue(result);
}
```

Assert Class

- Static methods
  - `AreEqual`
  - `AreNotEqual`
  - `AreNotSame`
  - `AreSame`
  - `Fail`
  - `Inconclusive`
  - `IsFalse`
  - `IsInstanceOfType`
  - `IsNotInstanceOfType`
  - `IsNotNull`
  - `IsNull`
  - `IsTrue`

CollectionAssert Class

- Equivalent class for collections
  - `AllItemsAreInstancesOfType`
  - `AllItemsAreNotNull`
  - `AllItemsAreUnique`
  - `AreEqual`
  - `AreEquivalent (same elements)`
  - `AreNotEqual`
  - `AreNotEquivalent`
  - `Contains`
  - `DoesNotContain`
  - `IsNotSubsetOf`
  - `IsSubsetOf`

StringAssert Class

- Equivalent class for strings
  - `Contains`
  - `DoesNotMatch`
  - `EndsWith`
  - `Matches`
  - `StartsWith`

Approval testing

- How to assert that a UI looks correct?
- How to assert a DataGrid contains the correct information?

- Approval testing
  - NuGet package
  - Approve a golden copy
  - Make sure the output looks like that

Write the Code

- Write enough code so that the test passes (Scott disagrees)

```
public class ToDoList
{
    public override bool Equals(object obj)
    {
        return true;
    }
}
```

Add another test

- Write tests by asserting truths
  - use **`Assert`** class static methods
  - several methods are available
  - support different types through generics

```
[TestMethod]
public void ToDoListEqual_WhenNamesAreNotTheSame_ReturnsFalse()
{
    ToDoList list1 = new ToDoList("Name") { Id = 1 };
    ToDoList list2 = new ToDoList("Name1") { Id = 1 };

    bool result = list1.Equals(list2);

    Assert.IsFalse(result);
}
```

Refactor

- Make the tests go ‘green’
  - reduce duplication
  - improve design
  - (should be) one step at a time

```
public override bool Equals(object obj)
{
    if (obj == null || typeof(ToDoList) != obj.GetType())
        return false;

    ToDoList other = (ToDoList) obj;

    return (other.Id == Id && other.Name == Name);
}
```

Testing Exceptions

- Some code will throw exceptions
  - unexpected exceptions will fail the test

```
[TestMethod]
[ExpectedException(typeof(ArgumentException))]
public void ListCtor_IfPageSizePassedIsZero_ThrowsException()
{
    ListController controller = new ListController(null, 0);
}
```

Ignoring Tests

- Used when you do not want the test to run

```
[TestMethod]
[Ignore]
public void All_Products_Page_1_Is_At_Slash()
{
    Assert.AreEqual("/", GetOutboundUrl(
    new
    {
        controller = "Products",
        action = "List",
        category = (string)null,
        page = 1
    }));
}
```

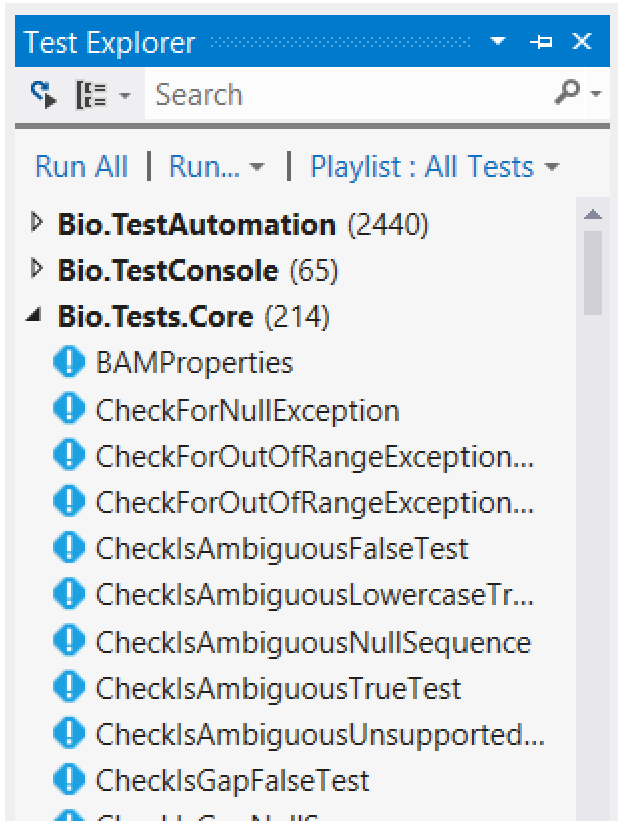
Initializing Tests and Classes

- Common code can be refactored
  - Per class initialization
  - Per test initialization
- Use attributes

```
[ClassInitialize]
[ClassCleanup]
[TestInitialize]
[TestCleanup]
```

Running Tests – Visual Studio

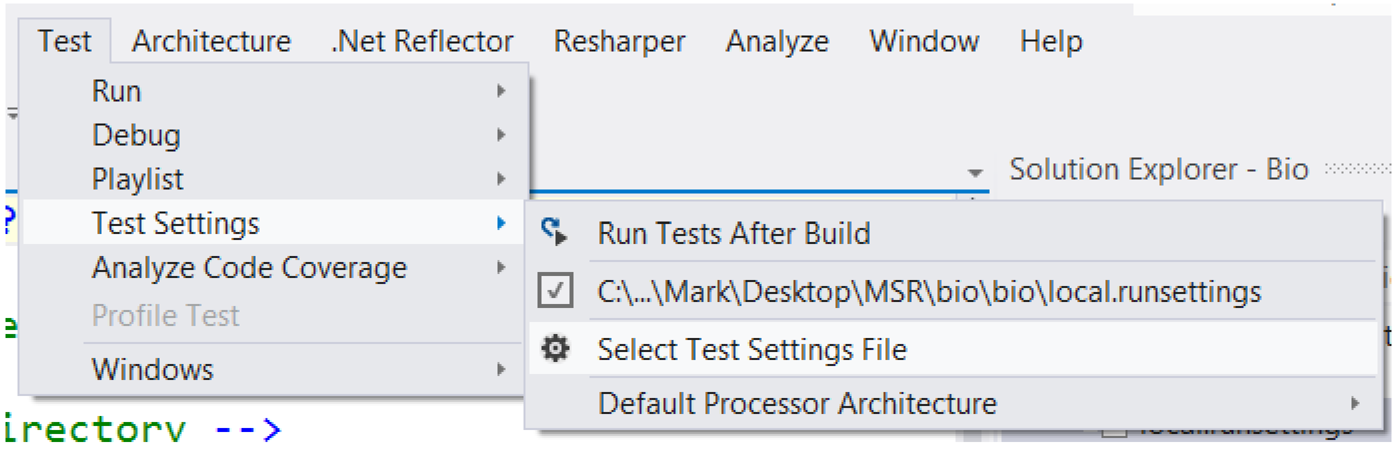
- Test Explorer
  - displays all tests found in solution
  - can execute some or all of the tests
  - can also debug tests…



Bio.TestAutomation (2440)
Bio.TestConsole (65)
Bio.Tests.Core (214)
- BAMProperties
- CheckForNullException
- CheckForOutOfRangeException...
- CheckForOutOfRangeException...
- CheckIsAmbiguousFalseTest
- CheckIsAmbiguousLowercaseTr...
- CheckIsAmbiguousNullSequence
- CheckIsAmbiguousTrueTest
- CheckIsAmbiguousUnsupported...
- CheckIsGapFalseTest

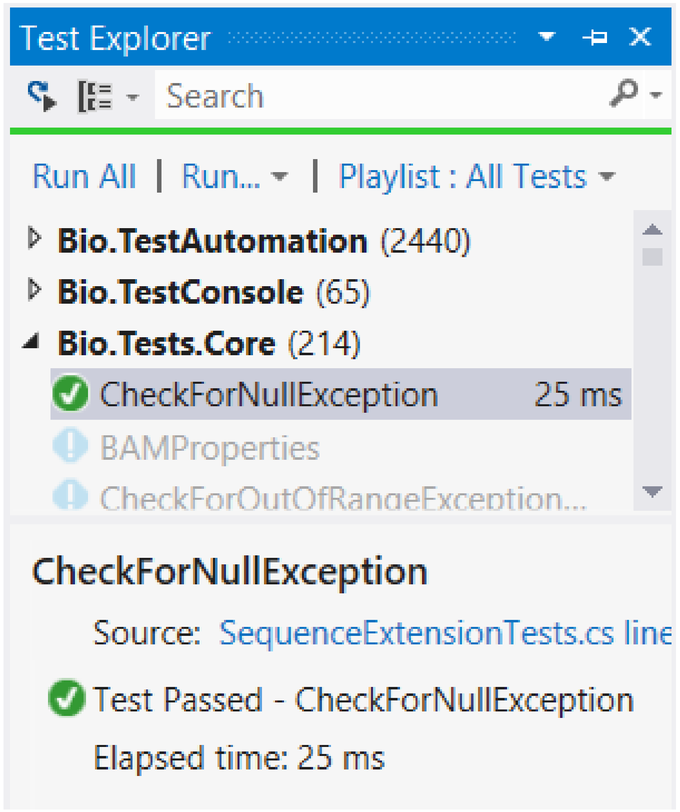
Configuring Test Runs

- Specify data about test runs
  - where to run; enable code coverage; etc…
- Can have multiple configurations per project
  - set active configuration in Test menu
- New .runsettings provides much faster test runs in VS2012
  - allows other testing frameworks to be configured
  - see http://msdn.microsoft.com/en-us/library/jj635153.aspx



Test Results

- Appear in test explorer



Bio.TestAutomation (2440)
Bio.TestConsole (65)
Bio.Tests.Core (214)
✓ CheckForNullException 25 ms
BAMProperties
CheckForOutOfRangeException...

CheckForNullException

Source: SequenceExtensionTests.cs line

✓ Test Passed - CheckForNullException

Elapsed time: 25 ms

Details are presented in lower pane

Inversion of Control

Coding to Concrete Types

- Tempting to create instances of dependent classes in code
  - tightly couples you to implementation
  - difficult to test
  - beware of `new` keyword

```
public User Logon( string userId, string password)
{
    LDAPRepository repo = new LDAPRepository();

    LDAPUser ldapUser = repo.Authenticate(userId, password);

    return new User{ Name = ldapUser.Name,
                     Groups = …};
}
```

What to abstract

- External I/O
  - File access
  - Database access
  - Communication (APIs, services)
- Time
- Environment

Coding to Abstraction

- Coding to abstract types decouples software
  - can replace implementation
  - can test more easily with mock version
- Extract interface from concrete implementation
  - refactoring support can do this
  - use interface in component
  - may involve extra work
  - may need to build façade to eliminate implementation types

```
public User Logon( string userId, string password)
{
    IUserRepository repo = ????

    User user = repo.Authenticate(userId, password);

    return user;
}
```

Dependency Injection

- If not able to create dependencies in component where do they come from?
  - must be passed to the component, known as **Dependency Injection**
- Three types of Dependency Injection
  - parameter Injection
  - setter Injection
  - constructor Injection

Parameter Injection

- Pass dependency as method parameter
  - useful for operation specific strategies

```
public User Logon( string userId,
                   string password,
                   IUserRepository repo )
{
    User user = repo.Authenticate(userId, password);

    return user;
}
```

Setter Injection

- Have property where dependency can be set
  - good for optional dependencies
  - consider null pattern to remove null checks from code

```
IUserRepository repo;

public IUserRepository Repository
{
   get{ return repo;}
   set{ repo = value ?? new NullRepository();}
}

public User Logon( string userId, string password )
{
    User user = Repository.Authenticate(userId, password);
    return user;
}
```

Constructor Injection

- Pass dependency to constructor of type
  - most common form
- Can also have default constructor with concrete type
  - maintains interface if changing an existing code base

```
IUserRepository repo;

public OrderingSystem(IUserRepository repo)
{
   this.repo = repo
}

public User Logon( string userId, string password)
{
   User user = repo.Authenticate(userId, password);
   return user;
}
```

Who Creates Dependencies

- Can define dependencies in config and use creator method
  - creator creates concrete class and injects dependencies

```
public class OrderSystemCreator
{
  public OrderingSystem CreateOrderingSystem()
  {
    string repoString =
          ConfigurationManager.AppSettings["repo"];
    Type repoType = Type.GetType(repoString);

    IUserRepository repo =
          (IUserRepository)Activator.CreateInstance(repoType);

    return new OrderingSystem(repo);
  }
}
```

... But what if the dependency has dependencies?

Inversion of Control Containers

- Building custom factories breaks down quickly
  - Complex dependency trees hard to construct
- Inversion of Control Containers take control of complex construction
- Many IoC Containers available for .NET
  - Unity
  - Castle Windsor
  - StructureMap
  - Ninject
  - Spring.NET
  - Autofac

Unity

- Unity is a traditional IoC container
  - from Patterns and Practices
  - maps abstractions to concrete types
  - can be configured in code or configuration file
  - open source project (http://unity.codeplex.com)
- Dependencies can be supplied via
  - constructor parameters (most common)
  - public properties[1]
  - method parameters

Registering services in Unity

- Can map abstract types to concrete types
  - can name the type so register multiple implementations
  - can control over lifetime of created object
    - by default a new instance is returned each time

```
UnityContainer container = new UnityContainer();

container.RegisterType<IUserLookup, LDAPRepository>();
```

- Can map abstract types to specific instances
  - allows creation of objects with runtime parameters
  - inherently a singleton with this mapping

```
UnityContainer container = new UnityContainer();

container.RegisterInstance<IAuthenticationService>(this);
```

Locating registered services in Unity

- Call `Resolve<T>` to get dependency instance
  - Unity creates instance and supplies any needed dependencies

```
IOrderLookup orderSystem = container.Resolve<IOrderLookup>();

IOrder order = orderSystem.FindOrderById(...);
```

- `ResolveAll<T>` returns `IEnumerable` of dependencies
  - needed if more than one type can satisfy request

```
List<IAuditor> auditers = new List<IAuditor>
      (container.ResolveAll<IAuditor>());

foreach (var auditor in auditors) { ... }
```

Named Mappings

- Sometimes need to register more than one concrete type for an abstraction
  - can give a mapping a name
  - pass name to **`Resolve`**

```
UnityContainer container = new UnityContainer();

container.RegisterType<IUserLookup, LDAPRepository>("win");
container.RegisterType<IUserLookup, FormsRepository>("db");

...

var catalog = container.Resolve<IUserLookup>("db");
```

Constructor Injection

- Dependency passed to constructor of type
  - most common form in Prism
- Can also have default constructor with concrete type
  - maintains interface if changing an existing code base

```
IUserRepository repo;

public OrderingSystem(IUserRepository repo)
{
  this.repo = repo
}
```

Property Injection

- Dependency assigned to property
  - unity general used to create type (i.e. **`Resolve<T>`**)
  - happens *after* construction
  - useful if type cannot have parameters in constructor

```
class OrderingSystem
{
    [Dependency]
    IUserRepository UserRepository { get; set; }

    public OrderingSystem()
    {
    }
}
```

attribute used to identify dependency requirement

Registering dependencies in Unity in configuration

- Unity supports configuration-driven registration
  - more flexible than code-based registration, but can be fragile
  - requires **`Microsoft.Practices.Unity.Configuration`**

```
<configuration>
   <configSections>
      <section name="unity"
          type="Microsoft.Practices.Unity.Configuration
                         .UnityConfigurationSection,
                Microsoft.Practices.Unity.Configuration" />
   </configSections>
   <unity>
      <container>
         <register type="Interfaces.IOrderService"
                   mapTo="Services.OrderService" />
      </container>
   </unity>
</configuration>
```

Summary

- Dependency Injection makes code loosely coupled and unit testable
- IoC brings sanity in DI systems
- Unity is Microsoft’s IoC Container

Test Doubles and Mocking

Agenda

- What are doubles?
- Continuum of doubles
- Mocking frameworks
- Different approaches to mocking

What are Doubles?

- Replacements for part of your code
  - depended on Components (DOC) of SUT
  - double provides same API as DOC

Why Doubles?

- Return values of components may not be repeatable
  - Date/time values
- Calls may be 'risky' or may be charged for
  - calling live web services during test
- Parts of the application are 'slow'
  - database access
  - file access
- Unit tests should not rely on external resources
  - databases
  - web Services

Mocks

- Provide behavior verification
  - check that the SUT calls correct methods …
  - … with the correct parameters …
  - … in the correct order
- Painful to build on your own
  - too much work – repeated over and over!
  - tooling becomes very helpful here

Creating mocks and stubs

- Typically created with a tool
  - Moq
  - RhinoMocks
  - TypeMock (heavy, IL-rewriting)
  - NMock
  - NSubstitute
  - Visual Studio Stubs/Fakes (Moles) (heavy)

Using Moq

```
var mock = new Mock<IFoo>();
mock.Setup(foo => foo.DoSomething("ping")).Returns(true);
```

Summary

- Doubles allow for awkward parts of the SUT to be tested
- There is a continuum of doubles
- Can create our own
- Can use a mocking framework

Behavior Driven Development

Why BDD

- Behavior-driven development was developed by Dan North as a response to the issues encountered teaching test-driven development
  - Where to start in the process
  - What to test and what not to test
  - How much to test in one go
  - What to call the tests
  - How to understand why a test fails

BDD and Gherkin

- Scenario 1: Refunded items should be returned to stock

  Given a customer previously bought a black sweater from me
  And I currently have three black sweaters left in stock
  When the customer returns the sweater for a refund
  Then I should have four black sweaters in stock

- Scenario 2: Replaced items should be returned to stock

  Given that a customer buys a blue garment
  And I have two blue garments in stock
  And three black garments in stock.
  When the customer returns the blue garment for a replacement in black
  Then I should have three blue garments in stock
  And two black garments in stock

Another example

- Given a 5 by 5 game
- When I toggle the cell at (3, 2)
- Then the grid should look like

```
.....
.....
.....
..X..
.....
```

- When I toggle the cell at (3, 1)
- Then the grid should look like

```
.....
.....
.....
..X..
..X..
```

SpecFlow

- Binds Gherkin syntax to .NET code

- A VisualStudio extension
- And a NuGet package
- Runnable in ReSharper
- Uses Nunit by default

Web testing

- Two types of automated web testing
  - Unit tests (easy if using MVC)
  - Front end tests

Selenium

- http://www.seleniumhq.org/
- Automates browsers