Unit Testing in .NET

Philosophy of Unit Testing

- What?
- Where?
- Why?
- How?
- What it **is not**?

- Test individual components (units)
- in isolation
- to confirm preexisting specification.

isolate and ~~validate~~ verify

- Inputs … -> expected results
- Test functions, methods.
- Use a framework to make it easier (xUnit or ?Unit)

- Write tests after creating specification, but before implementing methods.

- Best if written by the developer or the development team. Why?

- Test-driven development

- Supports regression testing

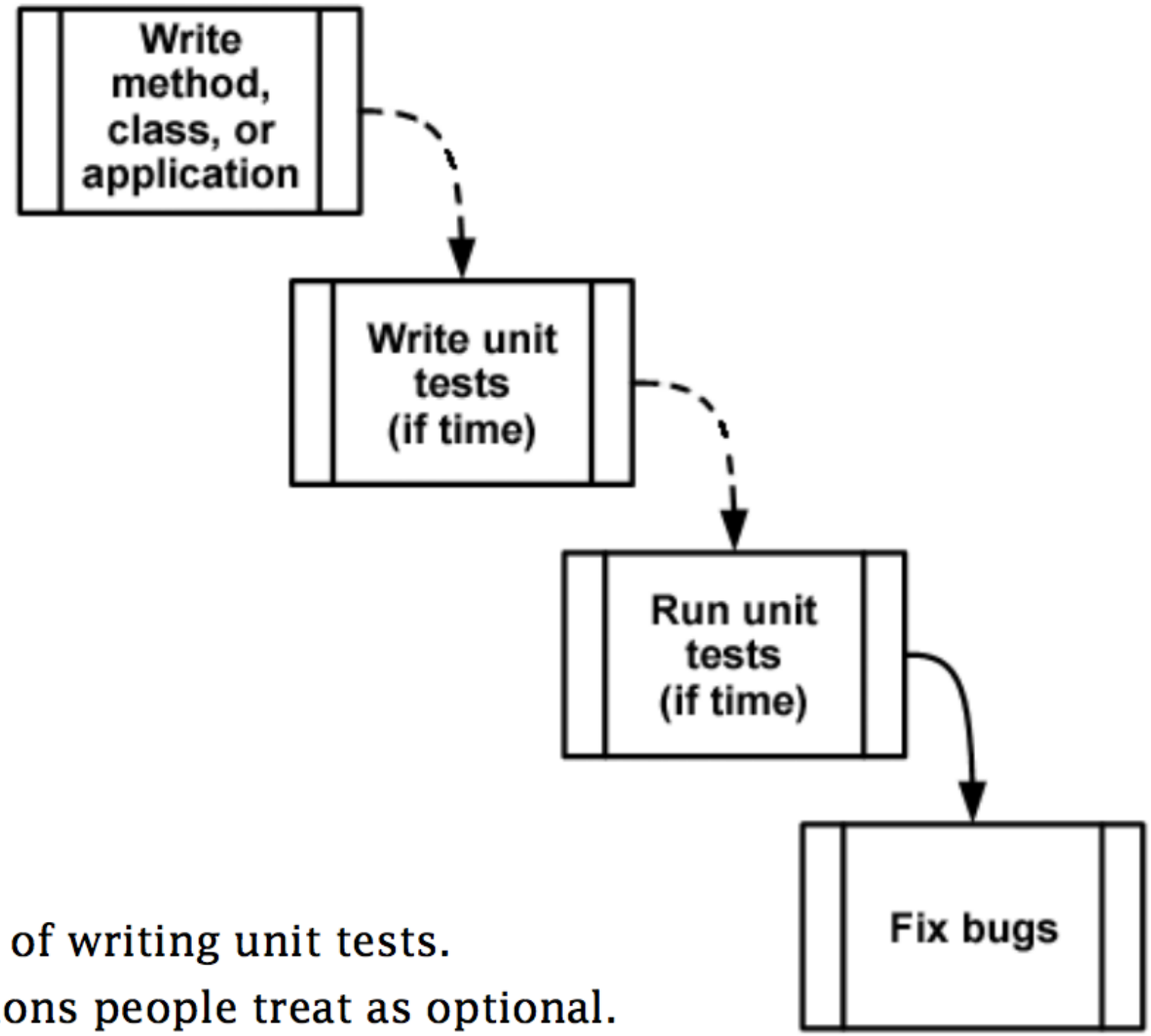


Figure 1.3 The traditional way of writing unit tests. The dotted lines represent actions people treat as optional.

The Art of Unit Testing, by R Osherov

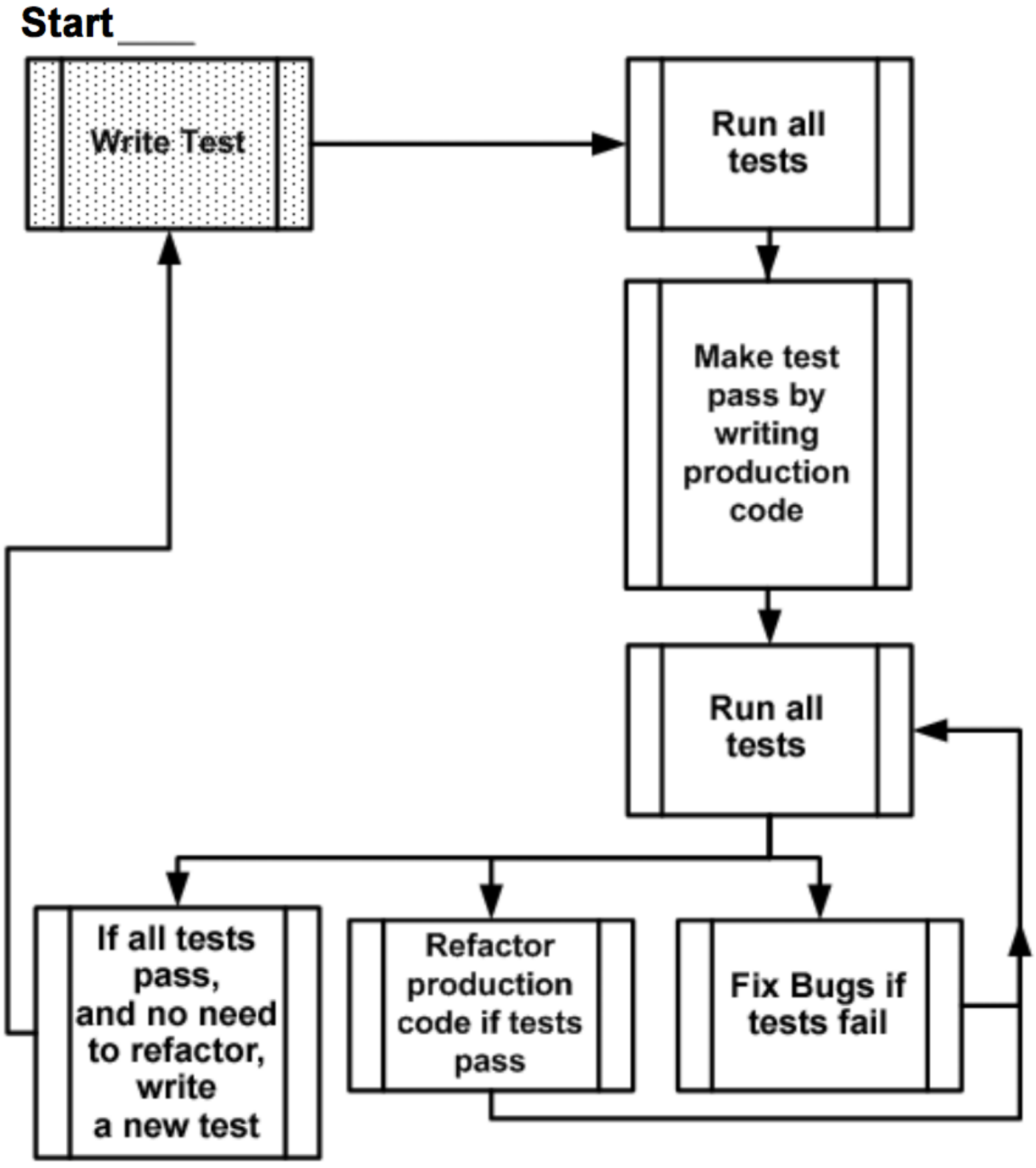


Figure 1.4 Test-driven development—a bird's-eye view. Notice the spiral nature of the process: write test, write code, refactor, write next test. It shows the incremental nature of TDD: small steps lead to a quality end result.

```java
/* ++++++++++ Test the HybridTST only ++++++++++ */

public void test0()
{
    HybridTST<Integer> t = new HybridTST<Integer>();
    assertTrue( t.isEmpty() );
}

public void test1()
{
    HybridTST<Integer> t = new HybridTST<Integer>();
    t.put("A",new Integer(0));
    assertFalse( t.isEmpty() );
}

public void test2()
{
    HybridTST<Integer> t = new HybridTST<Integer>();
    assertEquals( 0, t.size() );
}

public void test3()
{
    HybridTST<Integer> t = new HybridTST<Integer>();
    t.put("A",new Integer(0));
    assertEquals( 1, t.size() );
    assertFalse( t.isEmpty() );
}
```

```
public void test4()
{
    HybridTST<Integer> t = new HybridTST<Integer>();
    String[] arr = {"A","A","A","A","A"};
    for( String s: arr )
        t.put(s,new Integer(0));
    assertEquals( 1, t.size() );
    assertFalse( t.isEmpty() );
}

public void test5()
{
    HybridTST<Integer> t = new HybridTST<Integer>();
    String[] arr = {"A","B","C","D","E"};
    for( String s: arr )
        t.put(s,new Integer(0));
    assertEquals( 5, t.size() );
    assertFalse( t.isEmpty() );
}

public void test6()
{
    HybridTST<Integer> t = new HybridTST<Integer>();
    String[] arr = {"A","AB","ABC","ABCD","ABCDE"};
    for( String s: arr )
        t.put(s,new Integer(0));
    assertEquals( 5, t.size() );
    assertFalse( t.isEmpty() );
}
```

1 Unit Test

```
/**
 * Sets up the test fixture.
 *
 * Called before every test case method.
 */
protected void setUp()
{

}


public void test12()
{
    HybridTST<Integer> t = new HybridTST<Integer>();
    String[] arr = {"A","AB","ABC","ABCD","ABCDE","zef", "kra","ref","tem","are", "temperature"
    for( int i = 0; i < arr.length; ++i )
        t.put(arr[i],new Integer(i));
    for( int i = 0; i < arr.length; ++i )
        assertTrue(t.contains(arr[i]));
}


/**
 * Tears down the test fixture.
 *
 * Called after every test case method.
 */
protected void tearDown()
{
}
```

assertions

Run with TestRunner

Test Suite

Test Case

Unit test Unit test Unit test Unit test Unit test
Unit test Unit test Unit test Unit test Unit test
Unit test Unit test Unit test Unit test Unit test
Unit test Unit test Unit test Unit test Unit test
Unit test Unit test Unit test Unit test Unit test

Test Case

Unit test Unit test Unit test Unit test Unit test
Unit test Unit test Unit test Unit test Unit test
Unit test Unit test Unit test Unit test Unit test

Test Case

Unit test Unit test Unit test Unit test Unit test
Unit test Unit test Unit test Unit test Unit test
Unit test Unit test Unit test Unit test Unit test

Assertions

JUnit

```
assertEquals()
assertFalse()
assertNotNull()
assertNotSame()
assertNull()
assertSame()
…
```

By the way, you should use **assertions** in everyday programming

Java: `assert i > 0;`
C#: `Debug.Assert( i > 0 );`
Python: `assert i > 0`

http://junit.sourceforge.net/javadoc/org/junit/Assert.html

NUnit Assertions

Classic Model

```
Assert.isEmpty()
Assert.areEqual()
Assert.Null()
Assert.notNull()
…
```

Constraint Model

```
Assert.That()
```

https://github.com/nunit/docs/wiki/Classic-Model

https://github.com/nunit/docs/wiki/Constraint-Model

NUnit

constraint

```
Assert.That( aString, Is.Empty );
Assert.That( aString, Is.Not.Empty );
Assert.That( 2 + 2, Is.EqualTo(4.0) );
Assert.That( 5.5, Is.EqualTo( 5 ).Within(1.5).Percent );
Assert.That( "Hello!", Is.Not.EqualTo( "HELLO!" ) );
Assert.That( "Hello!", Is.EqualTo( "HELLO!" ).IgnoreCase );
```

Good Habits

- Try to make tests independent or *orthogonal* to other tests
- Test only one code unit at a time
- Don't name your tests: `test0`, `test1`, …
- Use setup and teardown to create a clean environment for each test
- Don't have unit tests access your database — use fake data (write a "mock" object that meets the same interface) ==> *break dependencies*
- Refactor to make your code more testable
- Discover a flaw at the component level? Write unit tests first to expose it (reproduce it). Then fix the error.
- Run tests prior to committing (or merging) your code

Start Simple

- Test methods with no dependencies
- Make sure constructor doesn't have dependencies

NUnit Demo (2017)

- Add a new project to your solution: Visual C# -> Test
- Remove MSTest parts: reference, using and packages.config

```
Microsoft.VisualStudio.QualityTools.UnitTestFramework
//using Microsoft.VisualStudio.TestTools.UnitTesting
<package id="MSTest.TestAdapter" version="1.1.11" targetFramework="net452" /> …
```

- Add NUnit to your test project using NuGet, packages:

Had to delete %AppData$/NuGet/NuGet.Config to load "online" package sources

`NUnit v3.2.0`

`NUnit3TestAdapter v.3.0.9-ctp-9` (needs "Include prerelease" checked)

- Add MVC framework using NuGet (since separate project isn't an MVC app)

`Microsoft.AspNet.Mvc` (Match the version # of your main project, mine was v.5.2.3)

- Add reference to your own project

- Add Visual Studio Test Generator
- Optionally install Visual Studio Test Adapter as an extension rather than locally within the project
- Restart Visual Studio and finish installation

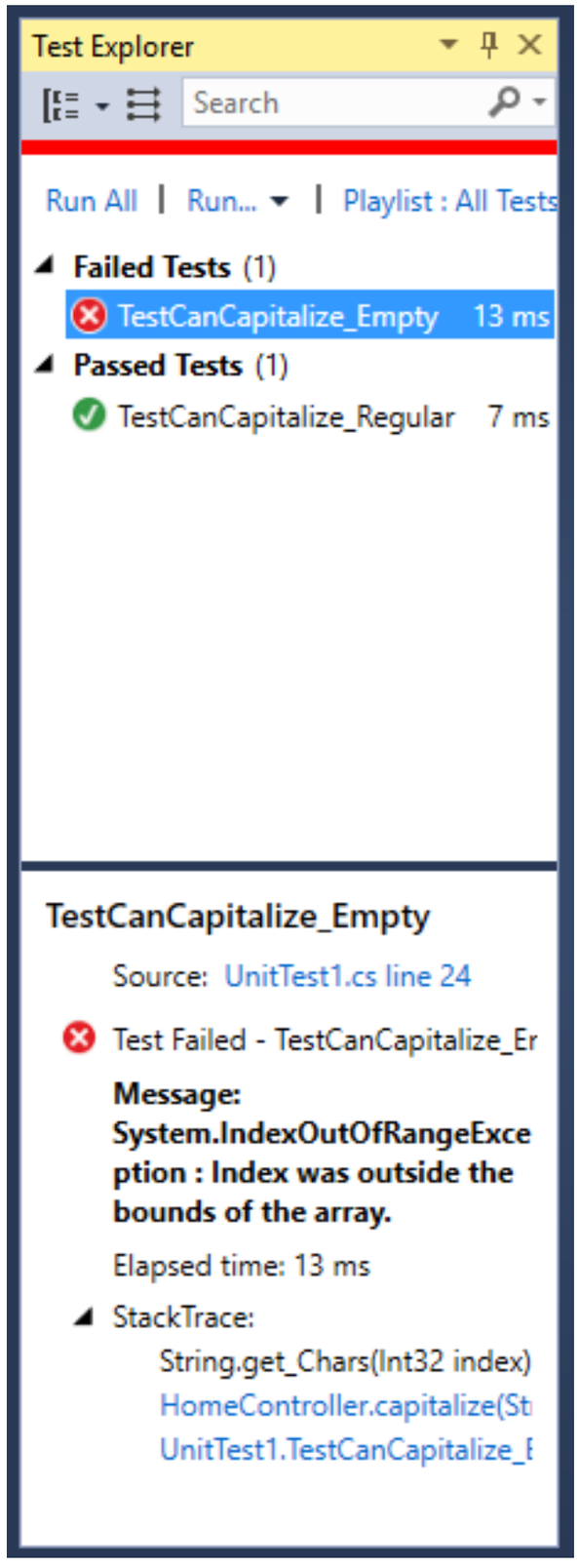
Find something to test

```
public class HomeController : Controller
{
    public string capitalize(string str)
    {
        return char.ToUpper(str[0]) + str.Substring(1);
    }
```

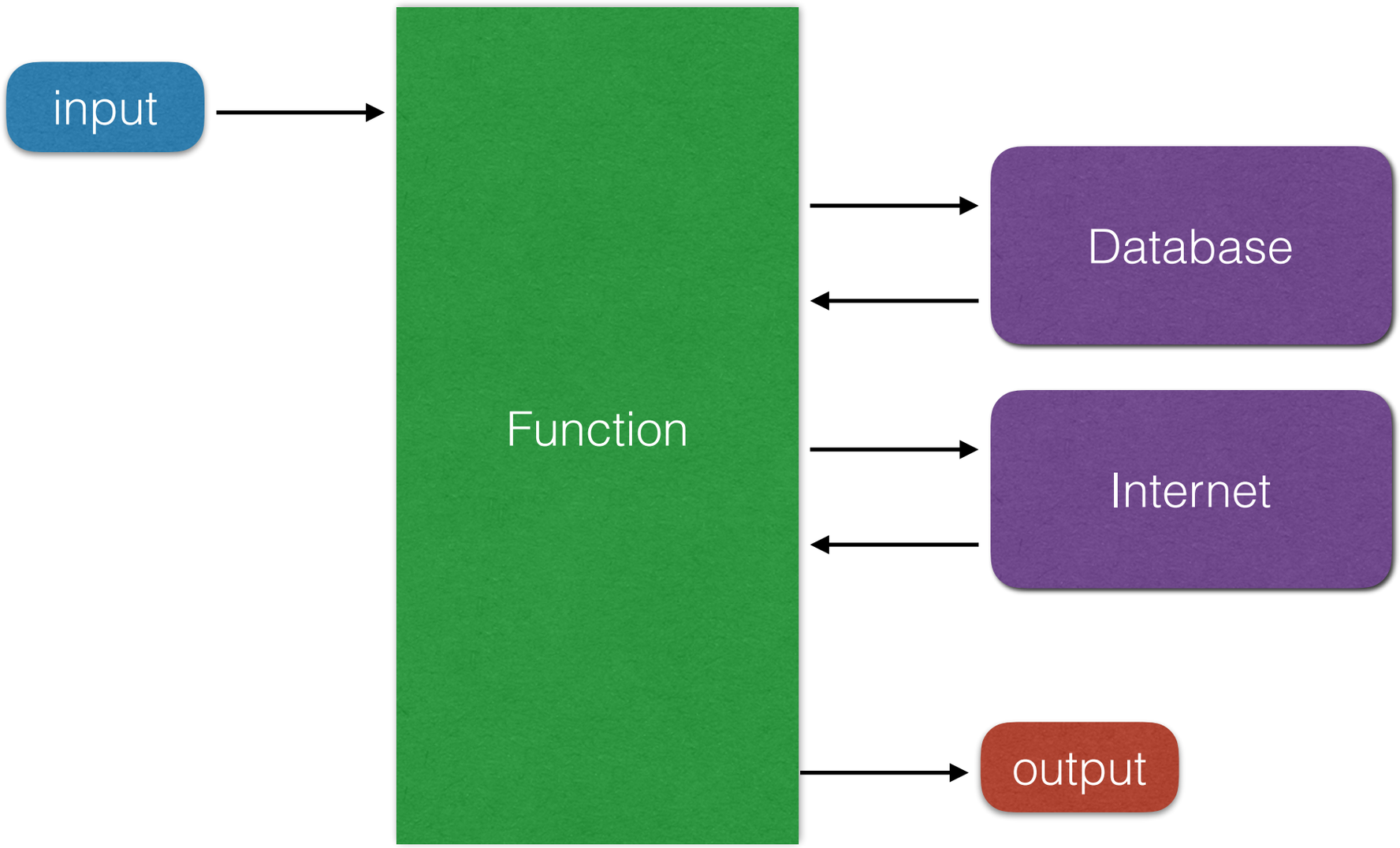
Write your
unit test class:

```
using System;
//using Microsoft.VisualStudio.TestTools.UnitTesting;
using NUnit.Framework;
using WebApplication1.Controllers;

namespace UnitTestProject1
{
    [TestFixture]
    public class UnitTest1
    {
        [Test]
        public void TestCanCapitalize_Regular()
        {
            HomeController c = new HomeController();
            string input = "my name is Horace";
            string expectedResult = "My name is Horace";

            string actualResult = c.capitalize(input);
            Assert.That(actualResult, Is.EqualTo(expectedResult));
        }

        [Test]
        public void TestCanCapitalize_Empty()
        {
            HomeController c = new HomeController();
            string input = "";
            string expectedResult = "";

            string actualResult = c.capitalize(input);
            Assert.That(actualResult, Is.EqualTo(expectedResult));
        }
    }
}
```

Run Tests: Test -> Windows -> Test Explorer



Harder



Not Unit Testable

```
public class ProductsController : Controller
{
    public int PageCount { get; set; } = 9;

    // GET: Products
    public ActionResult Category(int id, int page = 1)
    {
        using (AWData db = new AWData())
        {
            ProductSubcategory psc = db.ProductSubcategories.Find(id);
            IEnumerable<Product> products = psc.Products
                                              .OrderBy(p => p.Name)
                                              .Skip((page - 1) * PageCount)
                                              .Take(PageCount);

            ViewBag.Title = psc.Name;
            ViewBag.ID = id;
            ViewBag.Page = page;
            ViewBag.PageMax = Math.Ceiling(psc.Products.Count() / (double)PageCount);
            return View(products);
        }
    }
```

Not Unit Testable

```
/// <summary>
/// Example: Synchronous Get of multiple resources
/// </summary>
public JsonResult ExchangeRatesSynchronous(string currency = "EUR")
{
    // Each of these is a synchronous call to the respective API
    ExchangeRatesFromFixer erff = ExchangeRates.GetExchangeRatesFromFixer();
    ExchangeRatesFromOER eroer = ExchangeRates.GetExchangeRatesFromOER();

    if(!(  erff.IsValid && eroer.IsValid
        && erff.Rates.ContainsKey(currency) && eroer.Rates.ContainsKey(currency)
        && String.Equals(erff.Base,eroer.Base,StringComparison.OrdinalIgnoreCase)))
    {
        return Json(new { valid = false }, JsonRequestBehavior.AllowGet);
    }
    var data = new {    @base = "USD",
                        currency = currency,
                        date_day = erff.DateTimeStamp.ToString("o"),
                        date_now = eroer.DateTimeStamp.ToString("o"),
                        value_day = erff.Rates[currency],
                        value_now = eroer.Rates[currency],
                        difference = eroer.Rates[currency] - erff.Rates[currency]
                    };

    return Json(data, JsonRequestBehavior.AllowGet);
}
```

Why Not?

- “Uncontrolled” dependencies
- All or most functionality in 1 method, with no separation and no “control” over it
- Code is doing more than 1 thing

Arrange — Act — Assert

- **Arrange**

  Set up for the test. Configure the system one way or another. Must be able to do this.

- **Act**

  Execute or invoke something, with some input, to obtain some output. Must be able to do this.

- **Assert**

  Usually the easy part.

SOLID

- **S**ingle responsibility principle

  a class (module, method or function) should do one thing and one thing only

- **O**pen/closed principle

- **L**iskov substitution principle

  it should be possible to substitute in fakes, stubs or mocks without fundamentally changing anything

- **I**nterface segregation principle

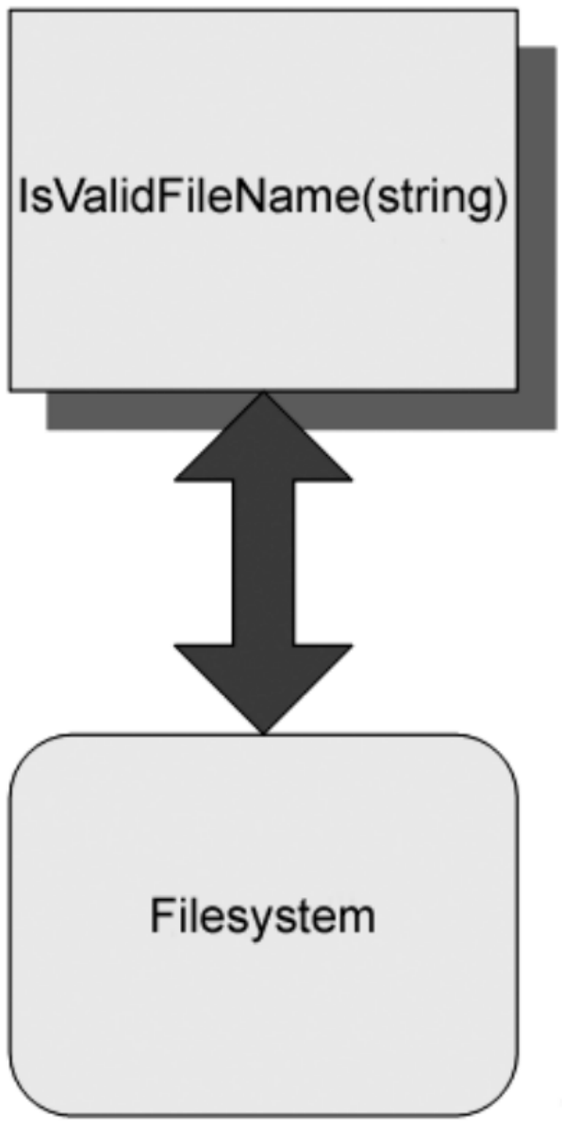
  create segregation of responsibilities and become agnostic of implementors by replacing concrete classes with interfaces (having the minimum of functionality required)

- **D**ependency inversion principle (and Inversion of Control)

  invert the “high-level module depends on low-level module” pattern. Break direct dependencies by using dependency injection pattern and providers.

https://en.wikipedia.org/wiki/SOLID_(object-oriented_design)

The problem that arises, as depicted in figure 3.1, is that, once this test depends on the filesystem, we're performing an integration test, and we have all the associated problems: integration tests are slower to run, they need configuration, they test multiple things, and so on.



This is the essence of *test-inhibiting* design: the code has some dependency on an external resource, which might break the test even though the code's logic is perfectly valid. In legacy systems, a single class or method might have many dependencies on external resources over which your test code has little, if any, control. Chapter 9 touches more on this subject.

Figure 3.1 Our method has a direct dependency on the filesystem. Our design of the object model under test inhibits us from testing it as a unit test; it promotes integration testing.

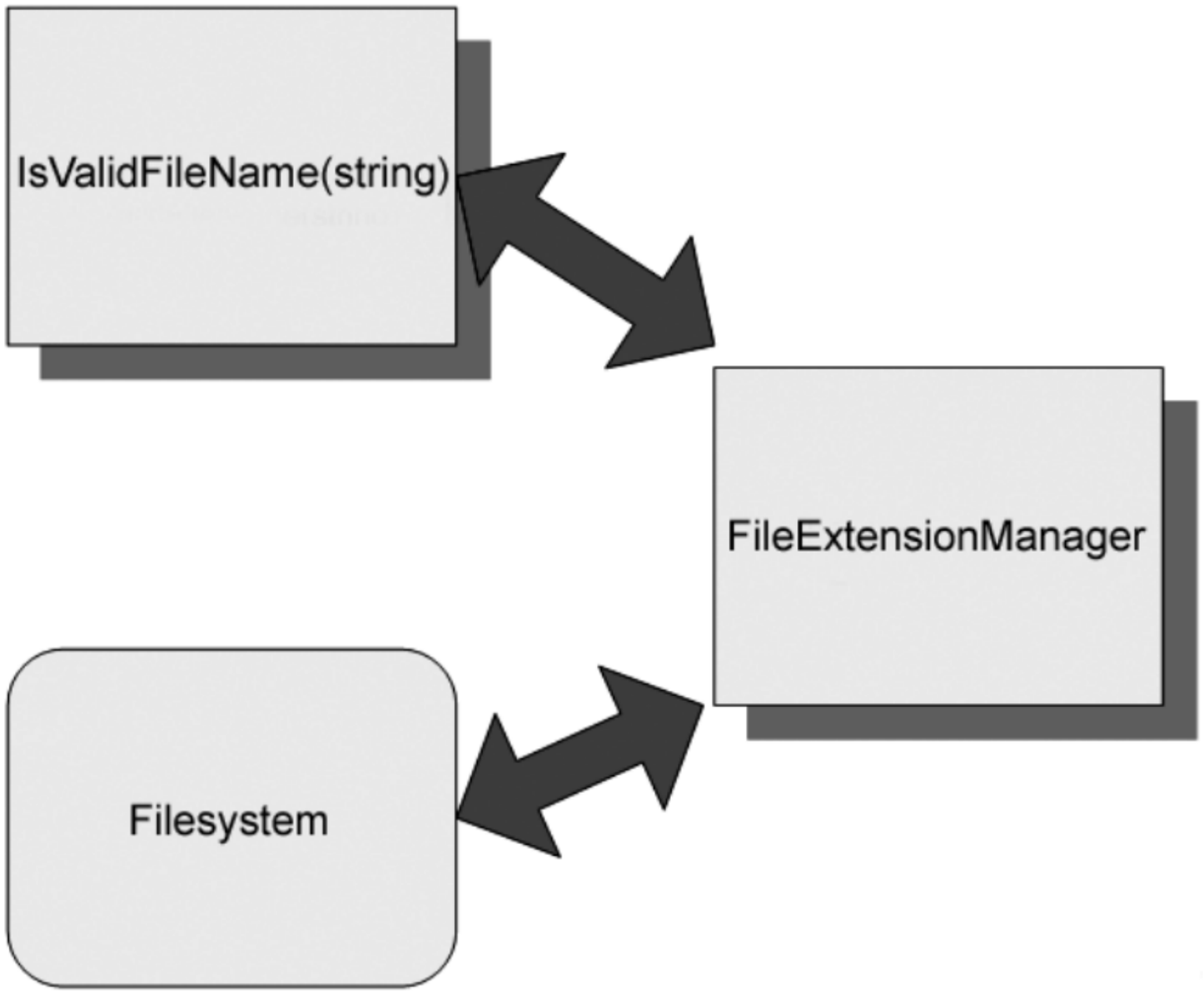


Figure 3.3 Introducing a layer of indirection to avoid a direct dependency on the filesystem. The code that calls the filesystem is separated into a `FileExtensionManager` class, which will later be replaced with a stub in our test.

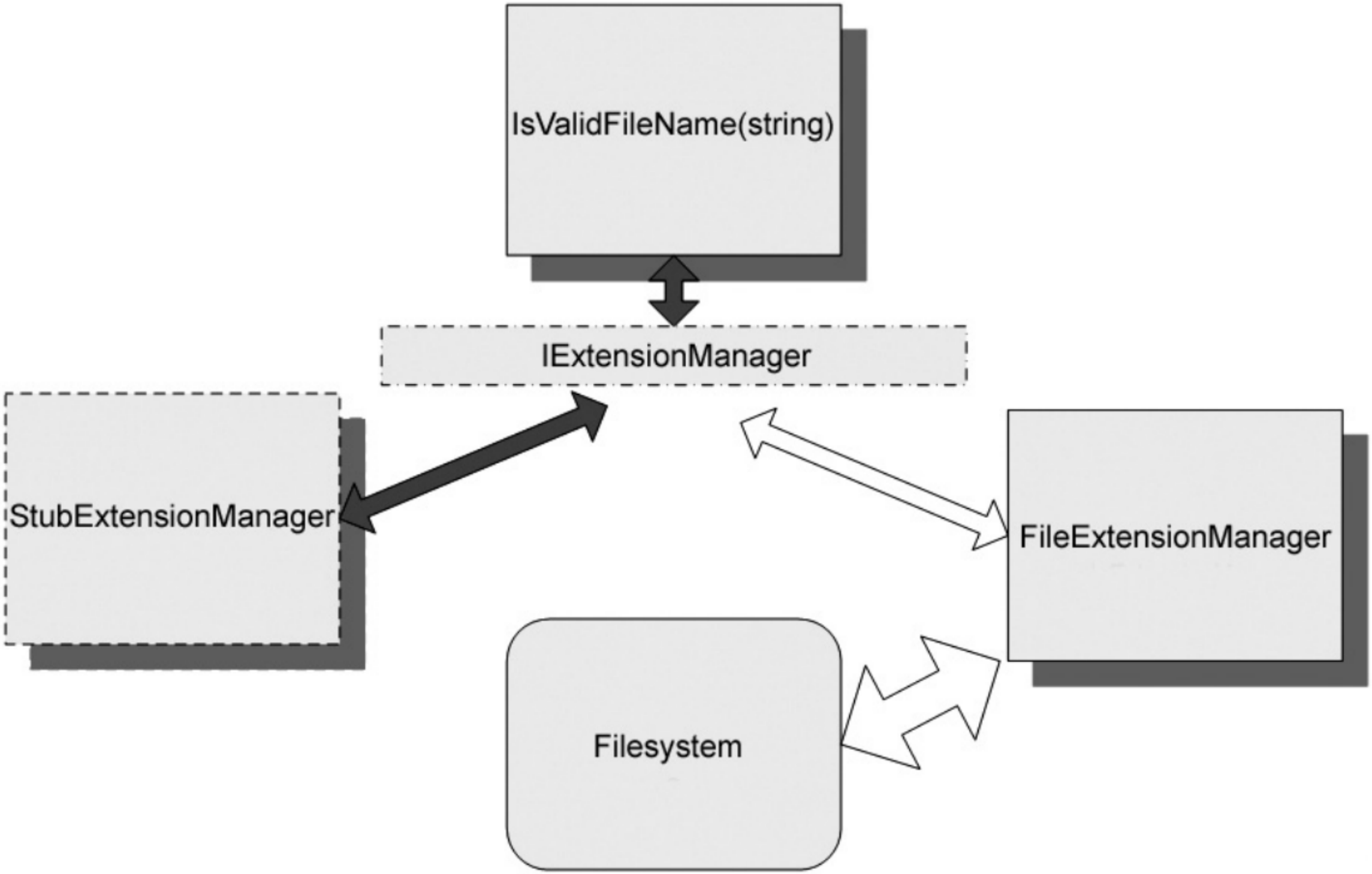


Figure 3.4 Introducing a stub to break the dependency

3.4 Refactoring our design to be more testable

I'm going to introduce two new terms that will be used throughout the book: *refactoring* and *seams*.

DEFINITION *Refactoring* is the act of changing the code's design without breaking existing functionality.

DEFINITION *Seams* are places in your code where you can plug in different functionality, such as stub classes. (See Michael Feathers' book, *Working Effectively with Legacy Code*, for more about seams.)

If we want to break the dependency between our code under test and the filesystem, we can use common design patterns, refactorings, and techniques, and introduce one or more *seams* into the code. We just need to make sure that the resulting code does exactly the same thing. Here are some techniques for breaking dependencies:

- Extract an interface to allow replacing underlying implementation.
- Inject stub implementation into a class under test.
- Receive an interface at the constructor level.
- Receive an interface as a property get or set.
- Get a stub just before a method call.

```
// Method under test
public bool IsValidLogFileName(string fileName)
{
    FileExtensionManager mgr = new FileExtensionManager();
    return mgr.IsValid(fileName);
}

class FileExtensionManager
{
    public bool IsValid(string fileName)
    {
        //read some file here
    }
}
```

```
// Method under test
public bool IsValidLogFileName(string fileName)
{
    IExtensionManager mgr = new FileExtensionManager();
    return mgr.IsValid(fileName);
}

// The real one (i.e. for production)
class FileExtensionManager : IExtensionManager
{
    public bool IsValid(string fileName)
    {
        //read some file here
    }
}

public interface IExtensionManager
{
    bool IsValid (string fileName);
}
```

```
// The fake one
public class StubExtensionManager:IExtensionManager
{
    public bool IsValid(string fileName)
    {
        return true;
    }
}
```

Just a stub for now

Problem?

One Solution: *inject via constructor*

```
public class LogAnalyzer
{
    private IExtensionManager manager;

    // Use this one to create real one
    public LogAnalyzer ()
    {
        manager = new FileExtensionManager();
    }

    // Use this one to make another one, or fake
    public LogAnalyzer(IExtensionManager mgr)
    {
        manager = mgr;
    }

    // Method under test
    public bool IsValidLogFileName(string fileName)
    {
        return manager.IsValid(fileName);
    }
}
```

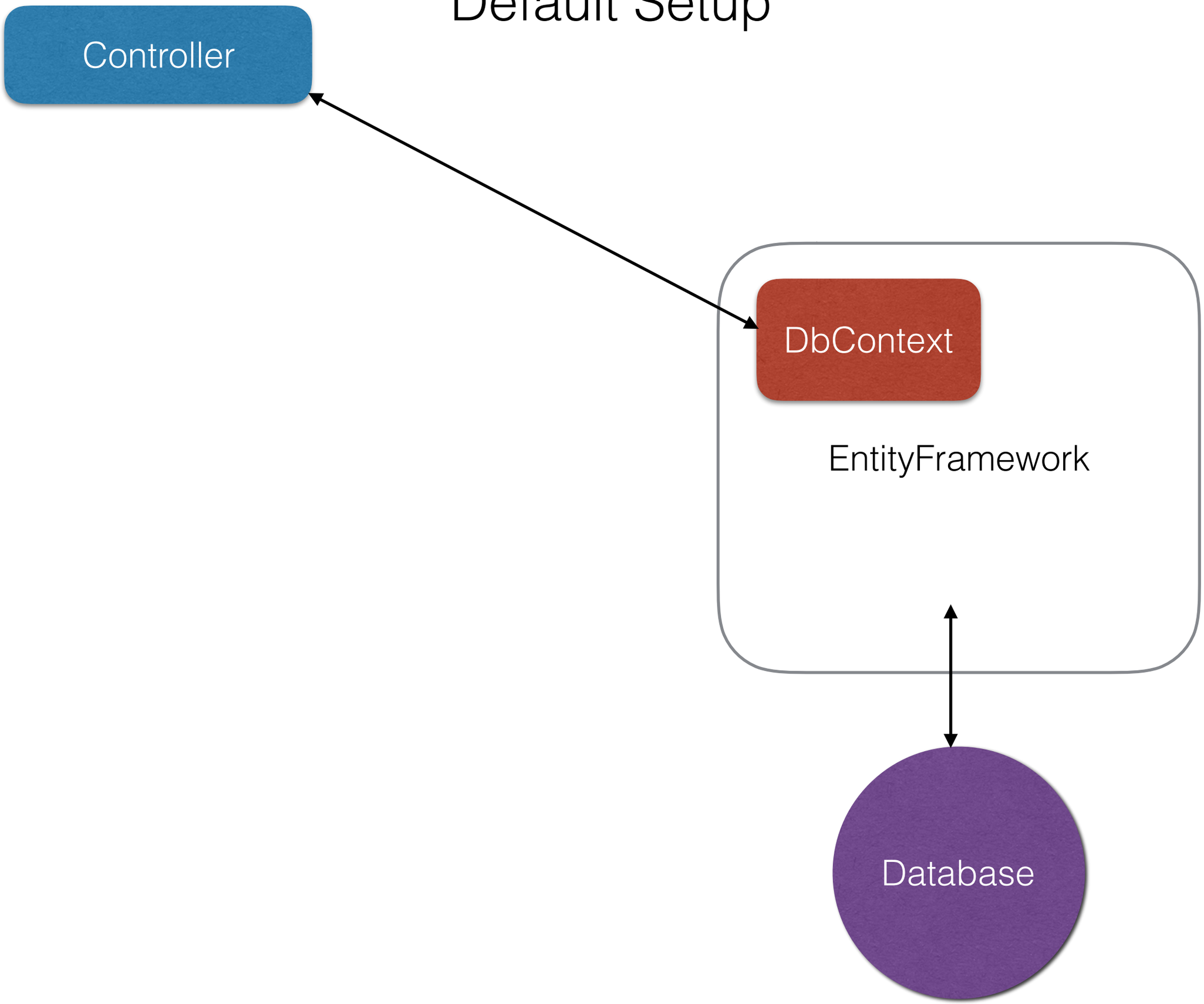
will be other logic here, i.e. test the name not just the extension

```
public interface IExtensionManager
{
    bool IsValid(string fileName);
}

// Fake one for testing
internal class StubExtensionManager : IExtensionManager
{
    public bool ShouldExtensionBeValid;

    public bool IsValid(string fileName)
    {
        return ShouldExtensionBeValid;
    }
}
```

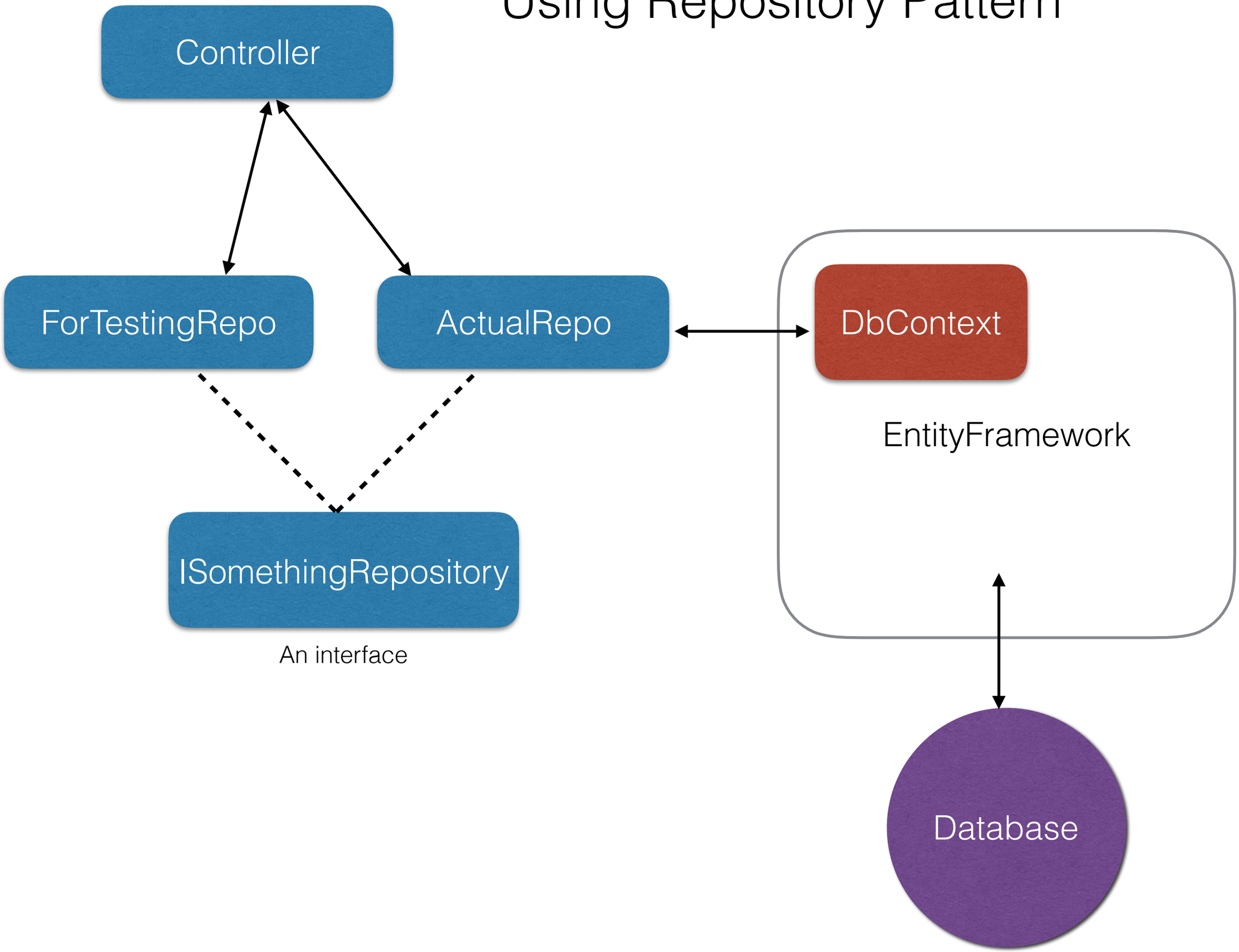
```
[TestFixture]
public class LogAnalyzerTests
{
    [Test]
    public void
    IsValidFileName_NameShorterThan6CharsButSupportedExtension_ReturnsFalse()
    {
        StubExtensionManager myFakeManager = new StubExtensionManager();

        // Define correct behavior
        myFakeManager.ShouldExtensionBeValid = true;

        //create analyzer and inject stub
        LogAnalyzer log = new LogAnalyzer (myFakeManager);

        //Assert logic assuming extension is supported
        bool result = log.IsValidLogFileName("short.ext");

        Assert.IsFalse(result,"File name with less than 5 chars should have
                               failed the method, even if the extension is
                               supported");
    }
}
```

- Up Next …
  - Testing code that reads/writes to a database
  - Use a “mocking framework” to make it easier, i.e.
    - Moq: https://github.com/Moq/moq4
    - Use `[SetUp]` and `[TearDown]` to seed your mock object

Default Setup



Using Repository Pattern



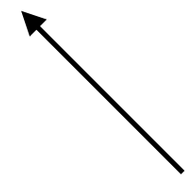
```
public class SomeController : Controller
{
    //private MyContext db = new MyContext();
    private ISomethingRepository myRepo;

    public SomeController()
    {
        this.myRepo = new EFSomethingRepository();
    }

}

public interface ISomethingRepository
{
    IQueryable<Something> Somethings { get; }
    Something Save(Something something);
    void Delete(Something something);
}


public class EFSomethingRepository : ISomethingRepository
{
    private MyContext db = new MyContext();

    // methods to give needed access to db, according to interface
}
```

Need to modify to allow constructor injection

moq

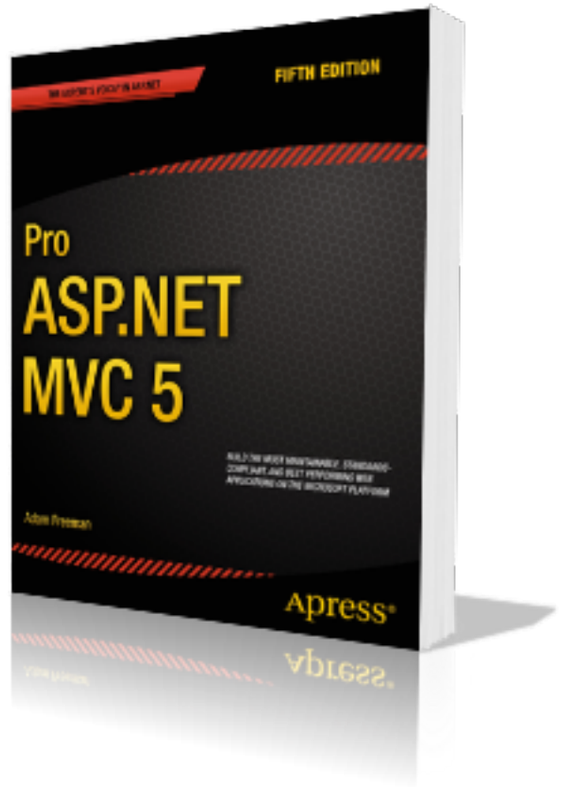
The most popular and friendly mocking framework for .NET

```
[Test]
public void Something_test_something()
{
    // Setup mock "db"
    Mock<ISomethingRepository> mock = new Mock<ISomethingRepository>();

    mock.Setup(m => m.Somethings).Returns( new Something[]
        {
            new Something {Id = 1, FirstName = "Hi", LastName = "Ho" },
            new Something {Id = 1, FirstName = "Lets", LastName = "Go" }
            }.AsQueryable());

    SomeController controller = new SomeController( mock.Object);

    // use the controller to test some functionality
    //var actualResult = controller.

    //Assert.Test(...)
}
```



Inject mock object

Sports Store Example from



Entity - Single table

```
namespace SportsStore.Domain.Entities {
    public class Product {
        public int ProductID { get; set; }
        public string Name { get; set; }
        public string Description { get; set; }
        public decimal Price { get; set; }
        public string Category { get; set; }
    }
}
```

Break dependency with Repository pattern

```
namespace SportsStore.Domain.Abstract {
    public interface IProductRepository {

        IEnumerable<Product> Products { get; }
    }
}
```

```
namespace SportsStore.Domain.Concrete {

    public class EFProductRepository : IProductRepository {
        private EFDbContext context = new EFDbContext();

        public IEnumerable<Product> Products {
            get { return context.Products; }
        }
    }
}
```

```
namespace SportsStore.WebUI.Controllers {

    public class ProductController : Controller {
        private IProductRepository repository;
        public int PageSize = 4;

        public ProductController(IProductRepository productRepository) {
            this.repository = productRepository;
        }

        public ViewResult List(int page = 1) {
            ProductsListViewModel model = new ProductsListViewModel {
                Products = repository.Products
                .OrderBy(p => p.ProductID)
                .Skip((page - 1) * PageSize)
                .Take(PageSize),
                PagingInfo = new PagingInfo {
                    CurrentPage = page,
                    ItemsPerPage = PageSize,
                    TotalItems = repository.Products.Count()
                }
            };
            return View(model);
        }
    }
}
```

constructor injection

pagination

Unit Test Pagination

```
namespace SportsStore.UnitTests {

    [TestClass]
    public class UnitTest1 {

        [TestMethod]
        public void Can_Paginate() {

            // Arrange
            Mock<IProductRepository> mock = new Mock<IProductRepository>();
            mock.Setup(m => m.Products).Returns(new Product[] {
                new Product {ProductID = 1, Name = "P1"},
                new Product {ProductID = 2, Name = "P2"},
                new Product {ProductID = 3, Name = "P3"},
                new Product {ProductID = 4, Name = "P4"},
                new Product {ProductID = 5, Name = "P5"}
            });

            ProductController controller = new ProductController(mock.Object);
            controller.PageSize = 3;

            // Act
            ProductsListViewModel result = (ProductsListViewModel)controller.List(2).Model;

            // Assert
            Product[] prodArray = result.Products.ToArray();
            Assert.IsTrue(prodArray.Length == 2);
            Assert.AreEqual(prodArray[0].Name, "P4");
            Assert.AreEqual(prodArray[1].Name, "P5");
        }
```