Unit Testing Overview

Table of Contents

1 Unit Testing 1

1.1 Overview 1

1.2 Unit Testing in .NET 2

1.3 Test Driven Development (TDD): 3

1.4 Disadvantages of Unit Testing: 6

2 Testable Code 8

2.1 Programming to Interfaces 8

2.2 Inversion of Control and Dependency Injection: 10

2.3 Mocking 14

2.4 Frameworks 16

2.5 MV* Patterns 18

3 Testing Database Code 21

4 Applied to our software 22

Table of Figures

Figure 1-1. A Simple Test Method 2
Figure 1-2. A Poorly Formatted Test Method (No AAA). 3
Figure 1-3. The TDD Workflow 4
Figure 1-4. Cost of Change vs. Development Time: TDD and Traditional Programming 6
Figure 2-1. Example of Using Interfaces 9
Figure 2-2. Traditional Coding 9
Figure 2-3. Coding to an Interface 9
Figure 2-4. Mechanical Controller 10
Figure 2-5. Loose Coupling 10
Figure 2-6. A Message Sender Program 11
Figure 2-7. Implementing IMessage in the TextMessage Class 11
Figure 2-8. Decoupling the MessageSender Class 12
Figure 2-9. Fully Decoupled MessageSender Class 12
Figure 2-10. Main Program Using the Decoupled MessageSender Class 13
Figure 2-11. Dependency Injection by Hand 13
Figure 2-12. Simple Mocking 15
Figure 2-13. Simple Mocking Using moq 16
Figure 2-14. MVC Role Diagram 19
Figure 2-15. MVP Variations 20

1 Unit Testing

1.1 Overview

A unit test is an automated piece of code that invokes a "unit of work" in the system and then checks a single assumption about the behavior of that unit of work.

A "unit of work" is a single logical functional use case in the system that can be invoked by some public interface (in most cases). A "unit of work" can span a single method, a whole class or multiple classes working together to achieve one single logical purpose that can be verified.

A *good* unit test is:

- Able to be fully automated.
- Has full control over all the pieces running (use mocks or stubs to achieve this isolation when needed; detailed in later sections).
- Can be run in any order if part of many other tests.
- Runs in memory (no database or file access, for example).
- Consistently returns the same result (you always run the same test, so no random numbers, for example. Save those for integration tests).
- Runs fast.
- Tests a single logical concept in the system.
- Readable.
- Maintainable.
- Trustworthy (when you see its result, you don't need to debug the code just to be sure).

Any test that doesn't live up to all these should be considered as an "integration test" and put it in its own "integration tests" project (or test procedure).

Roy Osherove - The Art of Unit Testing (http://artofunittesting.com/definition-of-a-unit-test/)

Integration tests are tests that evaluate pieces of the software (or the software as a whole) together. Unit tests do not replace integration tests, but if the code is written in a testable, loosely coupled fashion, the code becomes linear and the integration test results are usually just the sum of the individual unit tests. This is a huge benefit of writing unit-testable code (which is covered in more detail later).

From a system test engineer prospective, unit tests may include manual-step-by-step, human readable instructions, for the test engineer to follow. There are many potential downsides to this approach however, and these types of tests are best left as integration tests.

A direct quote from the Wikipedia article on unit testing helps illustrate this better: (http://en.wikipedia.org/wiki/Unit_testing)

"Unit testing is commonly automated, but may still be performed manually. ... The objective in unit testing is to isolate a unit and validate its correctness. A manual approach to unit testing may employ a step-by-step instructional document. However, automation is efficient for achieving this, and enables the many benefits listed in this article. Conversely, if not planned carefully, a careless manual unit test case may execute as an integration test case that involves many software components, and thus preclude the achievement of most if not all of the goals established for unit testing. To fully realize the effect of isolation while using an automated approach, the unit or code body under test is executed within a framework outside of its natural environment. In other words, it is executed outside of the product or calling context for which it was originally created.

Testing in such an isolated manner reveals unnecessary dependencies between the code being tested, and other units or data spaces in the product. These dependencies can then be eliminated. Using an automation framework, the developer codes criteria into the test to verify the unit's correctness. During test case execution, the framework logs tests that fail any criterion. Many frameworks will also automatically flag these failed test cases and report them in a summary. Depending upon the severity of a failure, the framework may halt subsequent testing. As a consequence, unit testing is traditionally a motivator for programmers to create decoupled and cohesive code bodies. This practice promotes healthy habits in software development. Design patterns, unit testing, and refactoring often work together so that the best solution may emerge."

System testability can be enhanced through developer/tester cooperation by the combination of automated unit tests and manual integration tests. From a developer's perspective, he should write the unit tests in code and then present to the tester a list of steps to run in addition to the unit test, which may involve other components of the software. For example, these types of tests may mimic what an actual user may do, while the unit tests strictly focus on testing the code itself. When these types of integration tests are combined with unit tests, the system is at optimal testability.

1.2 Unit Testing in .NET

In .NET, most unit tests are simply "test methods" in special "test classes" that explicitly test units of code.

Most tests assert that values are equal to expected values, and also check and assert the state of any business objects.

```
public void TestCalculatorTwoPlusTwo()
{
    // Arrange
    int firstNumber = 2;
    int secondNumber = 2;
    var calculator = new Calculator();

    // Act
    int result = calculator.Add(firstNumber, secondNumber);

    // Assert
    Assert.AreEqual(4, result);
}
```

Figure 1-1. A Simple Test Method

This test will pass if the value of `result` is equal to the expected value, 4. If for some reason `result` is not equal to 4, this test will fail, which indicates that there must be a bug somewhere in the `Add` method of the `Calculator` class.

Good tests should follow the Arrange Act Assert (AAA) pattern. Compare the previous test (Fig. 1-1) to the following test (Fig. 1-2), which does the same thing:

```
public void PoorlyFormattedTest()
{
    int firstNumber = 1;
    var calculator = new Calculator();
    int secondNumber = 2;
    int result = calculator.Add(firstNumber, secondNumber);
    Assert.AreEqual(4, result);
}
```

Figure 1-2. A Poorly Formatted Test Method (No AAA).

In the AAA pattern, test methods are split into sections of arrange, act, and assert. With the AAA pattern, it's much easier to see:

- What is being set up and initialized in the arrange section.
- What method or operation is being executed in the act section.
- What determines the outcome of the test in the assert section.

Using the AAA pattern ensures that tests are easy to read, follow, maintain, and understand.

Jag Reehal - http://www.arrangeactassert.com/why-and-what-is-arrange-act-assert/

1.3 Test Driven Development (TDD):

In a nutshell, TDD means "write your tests first". The TDD workflow should look like this:

1. Write a test... just enough code to compile and fail.
2. Run the test and watch it fail.
3. Write or update your functional code so that your test should pass.
4. Run the test, hopefully watch it pass.
    a. If it passes, you're done, and start this process all over again.
    b. If it fails, go back to step 2.

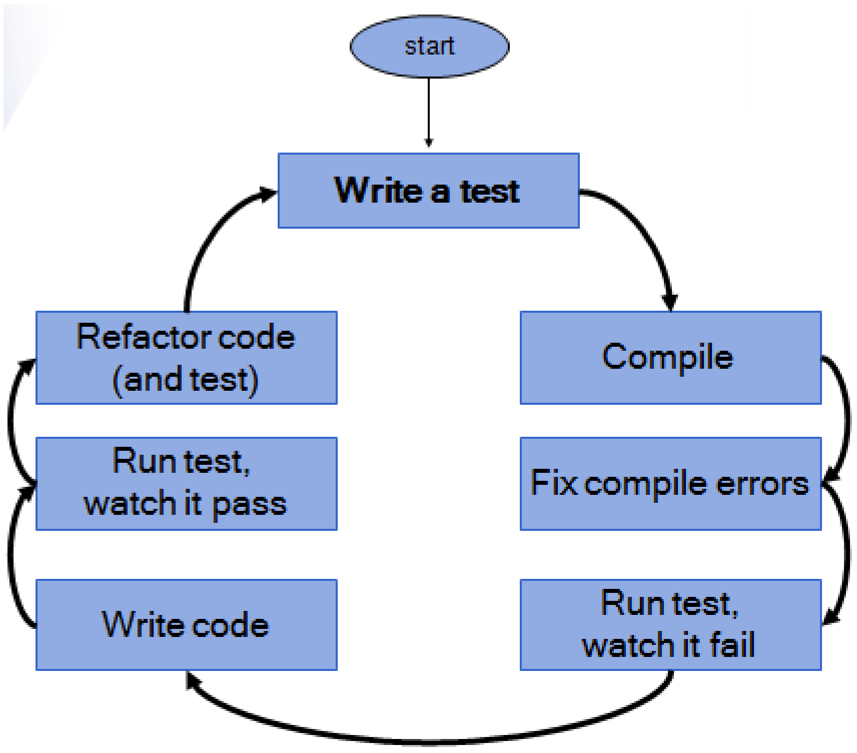

Figure 1-3. The TDD Workflow

Some advantages of Unit Testing in general:

1. **Better understanding of what you're going to write, BEFORE you write it!**

    - Unit Tests help you really understand the design of the code you are working on. Instead of writing code to do something, you start by outlining all the conditions you are subjecting the code to and what outputs you'd expect from that.

    - When you write the test cases first, you think more critically about the corner cases. It's then easier to address them when you write the code and ensure that they're accurate.

2. **Unit Tests allows you to make big changes to code quickly.**

    - You know it works now because you've run the tests, when you make the changes you need to make, you need to get the tests working again. This saves hours.

3. **Good unit tests can help document and define what something is supposed to do.**

4. **Unit tests find problems early in the development cycle.**

5. **An automated unit test suite watches over your code in two dimensions: time and space.**

6. **The tests and the code work together to achieve better code.**

    - Your code could be bad / buggy. Your TEST could be bad / buggy. In TDD you are banking on the chances of both being bad / buggy being low. Often it's the test that needs fixing but that's still a good outcome.

reefnet_alex - http://stackoverflow.com/questions/67299/is-unit-testing-worth-the-effort

Some advantages of TDD in general:

1. **TDD enforces the policy of writing tests a little better.**

2. **TDD results in developers being less afraid of changing existing code.**

3. **TDD reduces the need for manual testing, and makes manual testing easier overall.**

4. **The program's software development becomes more predictable and repeatable.**

5. **TDD speeds up development, resulting in a shorter development cycle.**

6. **TDD saves time (and hence money) in the long run by enforcing code that is less error prone.**

7. **TDD helps you to realize when to stop coding.**

    Your tests give you confidence that you've done enough for now and can stop tweaking and move on to the next thing.

8. **TDD helps with coding deadlock.**

    When faced with a large and daunting piece of work ahead writing the tests will get you moving quickly.

Another reason for TDD is to actually enforce writing the tests. Often when people do unit-testing without the TDD; they have a testing framework set up, write some new code, and then quit. They think that the code already works just fine, so why write tests? It's simple enough that it won't break, right? But now you've lost the advantages of doing unit-tests in the first place. Instead, write the tests first, and they're already there.

wbyoung - http://stackoverflow.com/questions/804569/why-should-i-use-test-driven-development

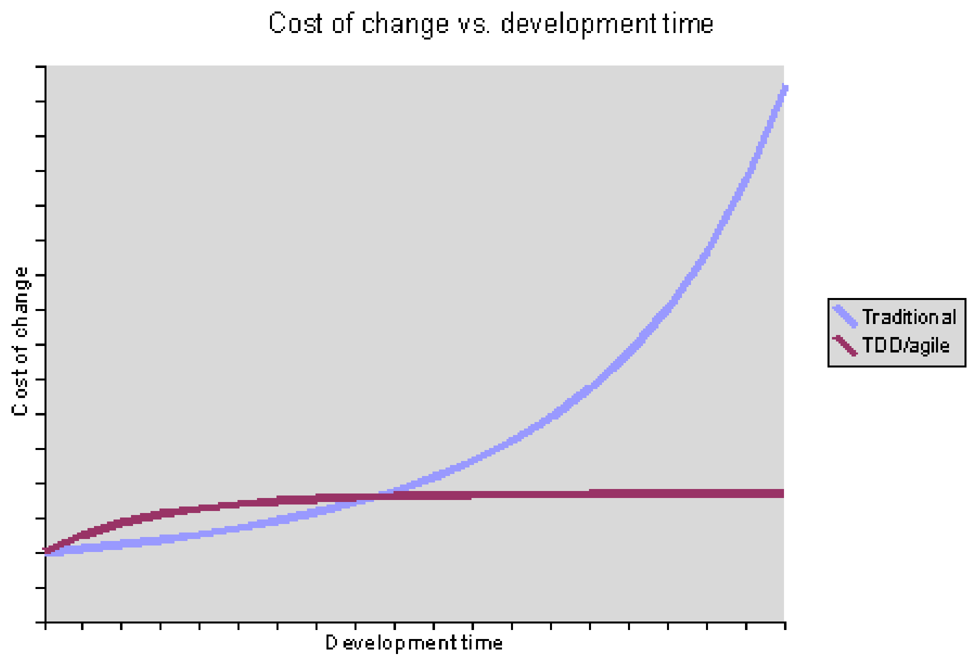

Figure 1-4. Cost of Change vs. Development Time: TDD and Traditional Programming

1.4 Disadvantages of Unit Testing:

Some disadvantages of unit testing (and TDD in some cases):

- **Unit testing requires a greater INITIAL time investment, as you have to write production code and test code.**
    - In the long run however, you are going to save time later while developing, especially for complex solutions because of easier maintenance and fewer bugs.
- **If you're just starting out, it can take some time to "get into it".**
    - Getting your brain into "testing mode" and writing "testable code" is a skill that must be learned.
- **Implementing unit testing in existing code that wasn't designed for it is laborious.**
    - The general solution to this problem is to simply start testing new code, and over time, eventually the old code will be re-written or refactored. At that point, unit testing is implemented.
- **Unit testing alone is not well suited for traditional "code-behind" GUI testing.**
    - GUI code can be made testable by implementing various MV* (MVC, MVC, MVVM, etc.) patterns.

- **Proper unit testing and TDD require cooperation and a dedicated team.**
    - Individual egos need to be checked at the door because code will constantly be under review and subject to rigorous testing. If one person decides he/she doesn't like unit testing, the code base as a whole will suffer.
- **TDD is easy to understand, but it is a tough discipline to master.**
    - Unit testing and TDD requires individuals who are willing to learn and teach themselves new skills.

http://stackoverflow.com/questions/64333/disadvantages-of-test-driven-development

http://stackoverflow.com/questions/3410609/how-to-best-present-unit-testing-to-management

On a whole, the benefits of TDD and unit testing in general have been shown to far outweigh the negatives on countless occasions. Unit testing and TDD is quickly becoming the industry standard for software development. The initial investment of extra effort in writing tests will allow you save time, effort, and money in the long run.

See the following white paper for an aggregate of empirical evidence on the effectiveness of TDD: http://www.agile-itea.org/public/deliverables/ITEA-AGILE-D2.7_v1.0.pdf

2 Testable Code

To take full advantage of unit testing, we need to *write testable code*.

Testable code is an umbrella term for code that is loosely coupled, has a well defined layered architecture (i.e. three-tier architecture), and is simply easy to write unit tests for.

Some guidelines for writing testable code:

- Write readable code - the first step in testing code is understanding it.
- Code/refactor to interfaces, not to implementations (makes mocking easier).
- Make public methods virtual if not using interfaces (again, makes mocking easier).
- Use dependency injection wherever possible (more on this later).
- Write smaller, more targeted methods and classes.
    - Classes should represent one thing, and their methods should do one thing (follow the single responsibility principle http://en.wikipedia.org/wiki/Single_responsibility_principle).
    - Write lots of small classes and methods in favor of few big classes and methods.
    - Keep classes limited to one file (exceptions may occur when classes are small and closely related).
- Avoid static and sealed classes, and singletons for cases that don't require them.
- Follow the Don't Repeat Yourself (DRY) principle, especially when writing tests. Avoid copy and pasting code.
- Remember the core tenets of object oriented programming (OOP): abstraction, encapsulation, inheritance, and polymorphism. Follow these at all times - smarter people than us invented them for a reason.
- For GUI and client development, follow a model-view-* pattern (MV*; model-view-controller, model-view-presented, model-view-viewmodel, etc.).

And perhaps most importantly...

- Always ensure a strict separations of concerns in your code. Break your code into pieces (units) of work that don't depend on each-other whenever possible. This falls in line with the OOP principles.

2.1 Programming to Interfaces

In general, programming to interfaces provides an easy way to enforce loose coupling and is a key concept in the "inversion of control" (IoC) technique. The concept itself is simple... instead of directly instantiating and using concrete classes, we instead use the interface of the class in the code. Programming to an interface embraces the OOP concepts of polymorphism and inheritance.

See the following example of an interface and a class that implements said interface in Figure 2-1:

```
interface IController
{
    string ControlSomething();
}

class RadioController : IController
{
    public string ControlSomething()
    {
        return "Controlled with radio";
    }
}
```

Figure 2-1. Example of Using Interfaces

The following code uses the RadioController class in the “traditional” way of coding in Figure 2-2:

```
RadioController controller = new RadioController();
string result = controller.ControlSomething();
Console.Write(result); // prints "Controlled with radio"
```

Figure 2-2. Traditional Coding

If we program to the interface instead, we would replace this with:

```
IController controller = new RadioController();
string result = controller.ControlSomething();
Console.Write(result); // prints "Controlled with radio"
```

Figure 2-3. Coding to an Interface

The second example (Fig. 2-3) is functionally equivalent to the first example (Fig. 2-2), except we've now coded to the interface IController. The interface IController is just a contract - it defines certain methods and fields that all classes that implement IController must have.

By coding to the interface, we have already removed much of the tight coupling present in the first example (Fig. 2-2).

Now let's examine another class, MechanicalController in Figure 2-4:

```
class MechanicalController : IController
{
    public string ControlSomething()
    {
        return "Controlled with gears";
    }
}
```

Figure 2-4. Mechanical Controller

If we take the code example presented in Figure 2-3 and replace it with the following (Fig. 2-5):

```
IController controller = new MechanicalController();
string result = controller.ControlSomething();
Console.Write(result); // prints "Controlled with gears"
```

Figure 2-5. Loose Coupling

Notice we only had to change the first line. Every other line that used the controller object would still execute, albeit it would have different outputs depending on the type of controller used.

The real benefit of this comes in unit testing. If we code strictly to interfaces, we can now create mock/fake objects that implement the interface to use in our tests, because we are basically saying "I need this functionality but I don't care where it comes from."

I highly suggest you read more about coding to interfaces in this great interview with Erich Gamma, co-author of the landmark book, *Design Patterns*: http://www.artima.com/lejava/articles/designprinciples.html

2.2 Inversion of Control and Dependency Injection:

Inversion of Control (IoC) is a high level design pattern aimed at reducing compile-time dependencies in your classes in favor of loosely coupled, run-time specified objects. Dependency injection is a more specific term, closely related to the IoC pattern, which describes how hard dependencies are de-coupled and bound at runtime.

What are dependencies? In the context of OOP, dependencies are any objects, services, or modules that are required for certain classes. The following example demonstrates a class with dependencies in Figure 2-6:

```
class MessageSender
{
    TextMessage message;

    public void SendMessage(string messageText)
    {
        message = new TextMessage();
        message.Send(messageText);
    }
}

class MainProgram
{
    static void Main(string[] args)
    {
        var messageSender = new MessageSender();
        messageSender.SendMessage("Hello world!");
    }
}
```

Figure 2-6. A Message Sender Program

The MessageSender class requires a concrete implementation of the TextMessage class (we don't care about the details of the TextMessage class yet… only that it has a Send method which takes a string as an argument) to send a message. This code is therefore tightly coupled - what would happen if we wanted to send a video message, for example, instead of a text message? We would need to replace TextMessage in our code with VideoMessage, and then ensure that everything still worked properly. If the VideoMessage class did not have a Send method, our program would fail.

Now let's use the concept of programming to an interface to illustrate dependency injection. If we define the interface IMessage and implement IMessage in our TextMessage class as follows in Figure 2-7:

```
interface IMessage
{
    void Send(string messageContent);
}

class TextMessage : IMessage
{
    string messageContent;

    public void Send(string messageContent)
    {
        this.messageContent = messageContent;
        Console.WriteLine("Sending text message: " + messageContent);
    }
}
```

Figure 2-7. Implementing IMessage in the TextMessage Class

Now we can modify our MessageSender class to use this interface, instead of the concrete implementation, as in Figure 2-8:

```
class MessageSender
{
    IMessage message;
    public void SendMessage(string messageText)
    {
        message = new TextMessage();
        message.Send(messageText);
    }
}
```

Figure 2-8. Decoupling the MessageSender Class

We're already on the right path here. This code is much more loosely coupled. All we have to do is change the line that instantiates the TextMessage instance if we want to send a different type of message. All the remaining code will work. This should be familiar by now since it was covered in the previous section. But here's where dependency injection (DI) comes in. We can STILL do better in regards to coupling, because the IMessage dependency is still tightly coupled with the MessageSender class. What if we instead passed our message to MessageSender on instantiation, through its constructor? Consider the following example in Figure 2-9:

```
class MessageSender
{
    IMessage message;

    public MessageSender(IMessage message)
    {
        this.message = message;
    }

    public void SendMessage(string messageText)
    {
        message = new TextMessage();
        message.Send(messageText);
    }
}
```

Figure 2-9. Fully Decoupled MessageSender Class

Using the version of the MessageSender class presented in Figure 2-9, our main program now becomes the following (Fig. 2-10):

```
class Program
{
    static void Main(string[] args)
    {
        var messageSender = new MessageSender(new TextMessage());

        // prints Sending text message: Hello world!
        messageSender.SendMessage("Hello world!");
    }
}
```

Figure 2-10. Main Program Using the Decoupled MessageSender Class

At this point, we moved the dependency of the IMessage concrete implementation up one level to the main program method. We can now change (re-wire) the 5th line of our program class to use any type of message we want, as long as it implements IMessage. For example, we can send a video message along with a text message using the same MessageSender class (Fig. 2-11):

```
class VideoMessage : IMessage
{
    string filename;

    public void Send(string filename)
    {
        this.filename = filename;
        Console.WriteLine("Sending video message with filename: " + filename);
    }
}

class Program
{
    static void Main(string[] args)
    {
        // Send a text message.
        var messageSender = new MessageSender(new TextMessage());

        // Prints Sending text message: Hello world!
        messageSender.SendMessage("Hello world!");

        // Send a video message.
        messageSender = new MessageSender(new VideoMessage());

        // Prints Sending video message with filename: leeroy_jenkins.avi
        messageSender.SendMessage("leeroy_jenkins.avi");
    }
}
```

Figure 2-11. Dependency Injection by Hand

This is called *dependency injection by hand* and demonstrates the simplest form of dependency injection: constructor injection. Notice that the MessageSender class no longer cares what type of message it is sending... all it cares about is "I have a message and I have to send it." The details of the type of message are handled by the individual message classes, and the IMessage interface links them all together. This demonstrates separations of concerns, as well as the single responsibility principle, and is immensely powerful. At this point, the code can be extended to handle multiple types of messages with just a few lines of code, rather than refactoring the base classes themselves.

Now it may seem that if we have a bunch of message types and we want to change them, or swap them in and out, we have to do all this wiring by hand and it could become tedious. Even worse is if the individual messages had dependencies themselves, since those would have to all be wired up too. IoC containers (dependency injection frameworks; Ninject, StructureMap, Unity, Spring, etc.) can make this process easier for you by doing all the heavy lifting. Without going into too much detail, an IoC container has a separate file (or class) where you specify bindings. A binding would be something like “bind the interface IMessage to the TextMessage class", and then everywhere in your code where IMessage is requested as a dependency, it gets filled in with the concrete implementation of TextMessage. These bindings are recursive, so if TextMessage had any dependencies itself, they would also be resolved automatically. If you want to change the concrete implementations of the dependencies, all you need to do is update the bindings. Of course, this is a very naive and general description of an IoC container, so please look at the individual documentation of the frameworks for more detail. They are very powerful and almost necessary in unit testing, particularly for providing mock objects.

2.3 Mocking

Mocking is a programming concept that allows mock (fake) objects to be created and used in unit tests. In short, mocking is creating objects that simulate the behavior of real objects. Mock objects allow us to isolate a class (or unit) from its dependencies to test. Used with dependency injection, mocking is incredibly powerful in the realm of unit testing.

We’ll start with a simple example of mocking. Using out previous example of IMessage and the MessageSender class, let's write a test for MessageSender's SendMessage method. Suppose that if a message's send fails, the method will throw a SendFailedException. We want to test the behavior of MessageSender in the case that this exception occurs. However, the only way we can get the method to throw that exception is if our message sender application has no connection to send on. While we could possibly replicate this scenario in the testing environment by disconnecting everything, it would be much easier just to make, use, and send a mock message that always throws that exception. Consider the following code in Figure 2-12:

```
class MockTextMessage : IMessage
{
    public void Send(string messageContent)
    {
        throw new SendFailedException();
    }
}

[TestFixture]
public class TestMessageSender
{
    [Test]
    public void TestSendFailedException()
    {
        // Arrange
        var message = new MockTextMessage();
        var messageSender = new MessageSender(message);

        // Act
        // Assert
        Assert.Throws<SendFailedException>(() =>
            messageSender.SendMessage("Hello world!"));
    }
}
```

Figure 2-12. Simple Mocking

Instead of using a real TextMessage and disconnecting something so that it fails and throws an exception, we simply created a mock object which ALWAYS throws an exception on send, regardless of everything else. This way, we can test how MessageSender behaves in the situation of such an exception, without having to manually do cause TextMessage to fail, which may be difficult or impossible in the development/testing environment.

Notice that we had to manually define a mock class, MockTextMessage, in the code to invoke this test. Mocking frameworks, like moq for .NET, allow us to create mock objects on the fly, fluently, directly in our unit tests. Here's an example of the same test as above, except this time using moq to define our mock classes (Fig. 2-13):

```
[TestFixture]
public class TestMessageSender
{
    [Test]
    public void TestSendFailedException()
    {
        // Arrange
        var mock = new Mock<IMessage>();
        mock.Setup(mockMsg => mockMsg.Send(It.IsAny<string>()))
            .Throws<SendFailedException>();

        IMessage message = mock.Object;
        var messageSender = new MessageSender(message);

        // Act
        // Assert
        Assert.Throws<SendFailedException>(() =>
            messageSender.SendMessage("Hello world!"));
    }
}
```

Figure 2-13. Simple Mocking Using moq

First the mock is created of type IMessage (Mock<IMessage>()). The Mock.Setup method creates a mock object with a method Send, which takes any string (It.IsAny<string>) as an argument, and throws a SendFailedException on invocation. The newly created mock object is accessed via the Mock.Object property, and then it is used in the test.

As demonstrated, the mocking framework has saved us from having to define a separate mock class, and can be used fluently directly in the unit test.

2.4 Frameworks

An outline of some of the various unit testing and related frameworks for .NET is given below.

Visual Studio Unit Testing Framework (MSTest):

- Microsoft proprietary software (closed source).
- Built into Visual Studio (VS).
- Default unit testing framework for .NET; easy to use and get started with.
- Only one version per VS version - releases are tied to VS releases.
- No native support for x64 testing (requires workaround).
- Visual Studio must be installed to run tests.
    - Therefore the build server must have Visual Studio (or VS runtime; 500+ MB).
- Works on Windows operating systems only - no platform interoperability.

NUnit

http://www.nunit.org/

- Open source (and hence free) unit testing framework, written in C#.
    - .NET port of JUnit – the defacto standard unit testing tool for Java for over a decade.
- Large user base and support community.
- Good documentation.
- Faster than MSTest when running tests - this is a biggie.
- Native x64 support.
- Platform interoperability - works with Mono.
- Losely coupled with the MS development stack.
- Very lightweight (Nuget package available).
- Works outside of visual studio.
- Easy to integrate with TeamCity.
- Frequent version updates, if desired.
- Many integrated runners (Resharper, TestDriven.NET, command line utility, GUI tool).
- Runs in the background.
- Potentially more flexible than MSTest.
- Slightly more complicated than MSTest (if you look for it).

moq

https://github.com/Moq/moq4

- Probably the most popular and widely supported mocking framework for .NET.
- Good documentation.
- Open source.
- Lightweight and easy to install (Nuget package available).
- Takes full advantage of .NET Linq expression trees and lambda expressions.

- Extremely simple and straightforward API (low learning curve).
- Moq's fluent API encourages easy to write and readable code.

Ninject

http://www.ninject.org/

- Open source IoC container focused on simplicity and ease of use.
- Lightweight distribution (~100 kB) - contains only what you need.
    - Extras available via extensions.
- Easy to install and use (Nuget package available).
- Fluent API with Linq-like syntax (similar to moq).
- No XML-based configuration binding by default - everything done in code.
    - XML binding is available through an extension if you prefer this method.
- Supports constructor injection, setter method injection, and property setter injection.

2.5 MV* Patterns

Model-view-star patterns are design patterns largely derived from the model-view-controller pattern. The star usually takes the form of one of three names: controller (MVC), presenter (MVP), and viewmodel (MVVM). There are plenty of differences between these design patterns, but their core purpose is all the same: separation of concerns, layered architecture, and loose coupling. I won't cover MVVM, but in a nutshell it is based heavily on data binding. It should be noted that it is increasingly becoming the main design pattern of web-based UI applications, and is extremely powerful with the proper framework (.NET's WPF for desktop applications; AngularJS, KnockoutJS, for web apps). There is plenty of information regarding MVVM on the internet if you are interested in learning more, since a detailed approach to these patterns is beyond the scope of this document.

In the model-view-controller pattern (MVC), the software is divided into three components: the models, views, and controllers (Fig. 2-14). The bottom layer is the model, which represents business rules, logic, application data, etc. The view is the user interface or any output representation of information to user. The controller takes user input, and changes the model/view respectively. The model doesn't depend on the controller or the view. The view depend on the controller. The controller may depend on both the model and the view, but only through contracts (interfaces) so that it remains loosely coupled. In most MVC applications, the controller is the entry point and it serves up the proper view and model, depending on the user input.

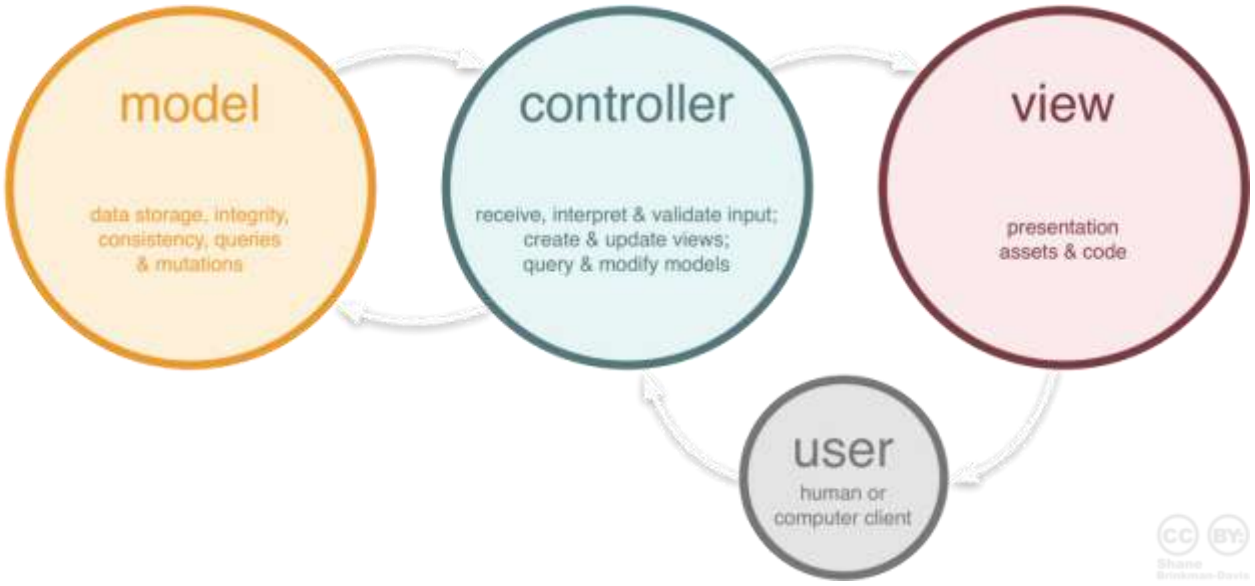
model
data storage, integrity, consistency, queries & mutations
controller
receive, interpret & validate input; create & update views; query & modify models
view
presentation assets & code
user
human or computer client
Shane Brinkman-Davis

Figure 2-14. MVC Role Diagram

MVC benefits from being extremely testable. Since everything is loosely coupled, or not coupled at all, nearly everything can be tested, including the views! Since the views usually don't contain any logic however, they are usually not tested via automated means, but instead manually for visual inconsistencies.

Since views are defined by interface, unit tests can make use of mock views to supply to controllers. These mock views can raise events and simulate user input, allowing the controllers to be extensively tested. Additionally, tests can provide mock models to controllers since they are also referenced by interface.

Models can also be unit tested easily since they are isolated from both the views and controllers by design.

http://stackoverflow.com/questions/2056/what-are-mvp-and-mvc-and-what-is-the-difference

A variation on the MVC pattern is the MVP, or model-view-presenter pattern. MVP is well suited for Windows Forms, as MVC and MVVM just don't work well with the WinForms architecture. The key difference between MVP and MVC is that in MVP, the view is the entry point to the application and delegates strictly to the middle-man, which in this case is the presenter. The presenter contains all the UI business logic for the view, and all invocations from the view delegate directly to the presenter. The presenter may also listen to events from the model, to update the view. Again, the model remains completely separate from the presenter and the view. In WinForms, the view would be represented by a form's code-behind file, which would implement a view interface. The views would then directly instantiate a presenter, and pass itself through the presenter's constructor as an interface.

There are two main variations with MVP: the passive view and supervising controller. In the passive view variation, the view is as dumb as possible and contains almost no logic. The view would only expose setter properties which the presenter would update. The model and view are completely separate from each other. This provides a maximum testability surface and a clean separation of concerns between the model, view, and presenter, but is also more coding work, since the presenter has to update all the properties of the view manually.

In the supervising controller variation, the presenter handles all user input, but the view directly binds to part (or all) of the model, through data binding. The presenter supplies the view with the model so it can bind to it. The view's data is then updated via the presenter and direct data binding on the model. This requires less overall code than the passive view method because of data binding, but it also reduces the testability surface and encapsulation since the view and model are now coupled.

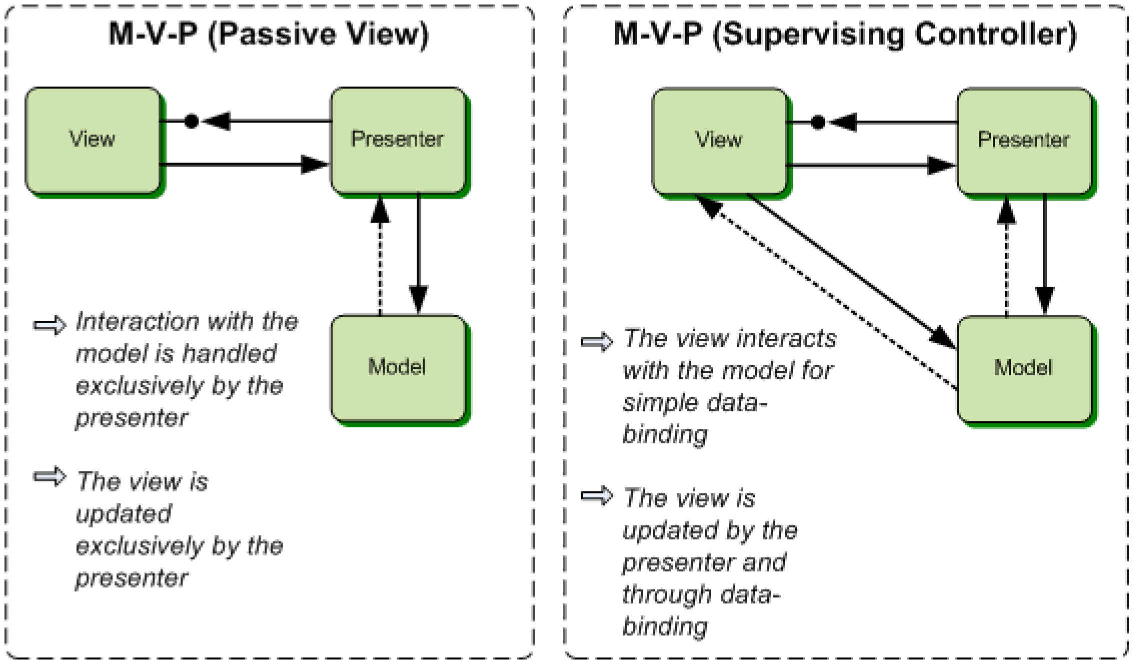
M-V-P (Passive View)
View
Presenter
Model
Interaction with the model is handled exclusively by the presenter
The view is updated exclusively by the presenter
M-V-P (Supervising Controller)
View
Presenter
Model
The view interacts with the model for simple data-binding
The view is updated by the presenter and through data-binding

Figure 2-15. MVP Variations

In GUI driven applications, it is highly recommended to implement one of these patterns if you want to do proper unit testing and achieve large code coverage. It also encourages readable, flexible, and maintainable code, by enforcing encapsulation and separation of concerns.

3 Testing Database Code

There are various options for testing database code and data access layers. For example, how would we test code that actually executes database queries and returns real data? The most widely recommended approach is to use a combination of mocks and stubs and a unit test database. If the code is decoupled enough, methods that handle the data and methods that actually access and return the data should be independent from each other. For example, if we have a method that requires a list of all usernames in the database, it should not actually access the database itself to get those usernames. Instead, it should use a class or method specifically designed to query and return the specified data, called a data access object (DAO). DAO's don't care about anything except hitting the database and inserting/returning data, if necessary.

If the method that requires the usernames uses a DAO, it is now loosely coupled with the database. Even further, if we code the username method to an interface and use dependency injection with the DAO, at testing time we can supply it with a mock DAO. Our mock DAO may return a hard coded set of usernames, specified in the test, for example, for the username method to use. All that is left to test at this point is the DAO itself.

As mentioned above, we can test DAOs by having a unit test database. If some of the DAO tests require data to be inserted, this can be handled by special "setup" and "teardown" scripts that get ran before and after the tests. Additionally, all tests should specifically back-out all transactions to ensure that the database is left in a consistent state.

Additionally, there are various DB unit testing frameworks out there that integrate well with existing frameworks. Some examples include DbUnit and NDbUnit (for .NET). Usage of these tools may be helpful in some situations, but they could be unnecessary.

Some more discussion on database testing can be found at:
http://www.bennorthrop.com/Home/Blog/unit_testing_daos_and_database_code.php

4 Applied to our software

Unit testing can be done on just about any application, as long as it is written in a testable fashion. Unfortunately, our software as a whole (client, server, and database) is very tightly coupled and hence difficult to test. That's not saying it's impossible, and the best way to start implementing tests is just to start with new code.

Any new version, release, patch, etc., should be written with unit tests. In this case, all new code is testable, and as old code is determined necessary to be refactored (or fixed), development should refactor that code to be testable as well. Additionally, tests should be written for the refactored code. Eventually, most of the code will be covered by unit tests.

Parts of the client can be refactored to follow the MVP design pattern, to enable automated testing of various parts of the system. The services can be refactored as well to establish more of a "separation of concerns" architecture, with well defined data access and logic layers. Unit tests can then be implemented here as well, and should be easier than testing the client.

The database and data access methods can be tested by employing a development database that is strictly used for testing. All data access code can be tested against this database, and the database itself will be, for all intents and purposes, static. It will be updated as the code and tests are updated, and it would be a good idea if this database was located on a central location such as the build server.

Eventually, a large majority of the code will be covered by automated unit tests. These tests will be ran on every build, and they can also be ran manually via command line or GUI tools. Nightly or weekly builds should be ran to ensure that the tests are constantly exercised. Additionally, developers can run the entire test suite at any time in Visual Studio.

If you’re a developer, I highly, highly recommend that you check out a productivity enhancement tool such as ReSharper (http://www.jetbrains.com/resharper/). It integrates fully with NUnit as a fully featured test runner, and is amazing for helping you optimize code and productivity in ways you would never imagine. Download the free trial to get started. If you just want a test runner, TestDriven.NET includes a personal license which can be used for free or evaluation.

Here’s a guide showing three ways to run NUnit from within Visual Studio: http://www.dev102.com/2008/03/22/3-ways-to-run-nunit-from-visual-studio/