Version

1.0

UCSD Extension

Information Technology and Software Engineering

Test Driven Development

© 2013 Raymond Mitchell III

Contents

[Recommended Book 2](#_Toc366625029)

[The Road to Test Driven Development 3](#_Toc366625030)

[1.1 Software development methodologies 4](#_Toc366625031)

[1.2 Concepts behind TDD 8](#_Toc366625032)

[1.3 Benefits of TDD 11](#_Toc366625033)

[1.4 Quick example of TDD 14](#_Toc366625034)

[Introduction to Unit Testing 26](#_Toc366625035)

[1.5 What is a unit test? 27](#_Toc366625036)

[1.6 NUnit (a unit testing framework) 32](#_Toc366625037)

[1.7 Decoupling with mock objects 37](#_Toc366625038)

[Refactoring With Confidence 48](#_Toc366625039)

[1.8 What is Refactoring? 49](#_Toc366625040)

[1.9 Why Refactor? 50](#_Toc366625041)

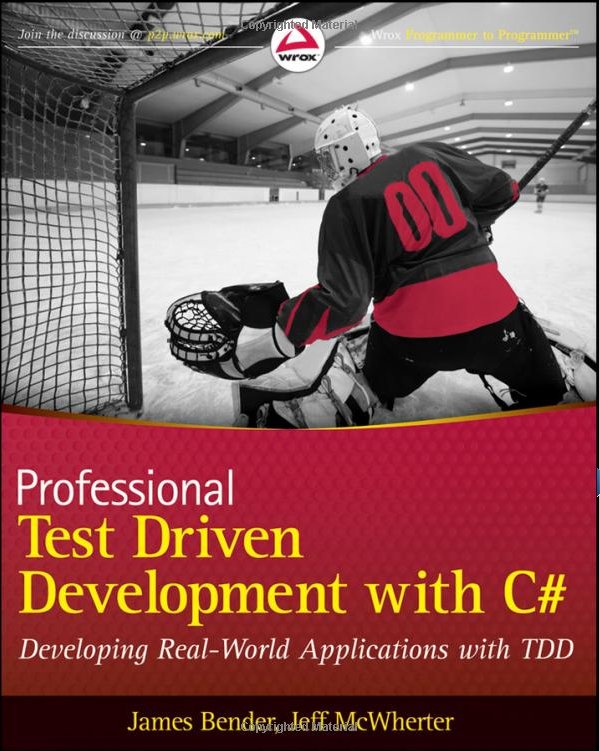
[1.10 Code Smells 54](#_Toc366625042)

[Conclusion 63](#_Toc366625043)

[1.11 Go forth and test! 64](#_Toc366625044)

# Recommended Book

**Professional Test Driven Development with C#: Developing Real World Applications with TDD**



# The Road to Test Driven Development

## Software development methodologies

* *Test-Driven Development* (TDD) has become one of the most important concepts and practices in modern software development.
* To understand why this is, consider the history of the practice of creating software.
* Age of the mainframe
  + Software development for business began during the age of the mainframe.
  + The mainframe was a large, expensive piece of equipment.
  + Imagine if you wrote a program today but couldn't compile it until next Monday.
  + The limited access to computing resources often meant that testing, out of necessity, took a backseat to getting the product out the door.
  + Developers often worked in an iterative manner, scoping out specific pieces of a system and completing those, and then adding new features and functionality later.
* Age of mini-computers
  + Developers could write applications for the PC and know right away if their code worked.
  + Developers didn’t have to wait days to have their jobs scheduled and run.
* Age of third and fourth generation programming languages
  + Abstracted some of the more mundane tasks of their predecessors and allowed developers to be more productive by focusing on the business problem at hand.
  + Opened software development to a wider audience who didn't want to deal with the friction of languages such as Assembler and C.
  + Business and the business computer industry settled on a few base languages and their derivatives.
  + Developers become more attractive and marketable to business as their skills became more portable.
* Age of waterfall development
  + Business's need to plan brought about the waterfall project methodology.
  + Every software project, whose average time span was about two years, should have every phase from inception to delivery planned from the start.
  + Waterfall phases
    - Necessitated a long period at the beginning for requirements gathering.
    - After all the requirements were gathered, they were “thrown over the wall” to the architects.
    - The architects designed the system that would be built down to the smallest detail thenthrew the design over the wall to the developers.
    - Developers then built the system.
    - Quality Assurance (QA) department tested the application.
    - As soon as the application was validated, it was deployed to the users.
  + Software testing in a waterfall methodology was often a long, difficult, inefficient, and expensive process.
  + QA testers would test applications by manually running through test scripts.
    - Documents (sometimes hundreds of pages) that instructed the tester to carry out an action in the system and described the result the tester should observe.
    - *It could take a tester two or more weeks to completely regression-test the system.*
    - Often test scripts were written by the developer who created the system.
      * In these cases the scripts usually described how the system *would* act, not how it *should* act.
* From Waterfall to Iterative and Incremental
  + Problems with waterfall
    - Software development doesn't happen in a void.
    - A problem with the waterfall methodology is that all the requirements are gathered early on.
    - In business, requirements often change for a variety of reasons.
    - A shift in the company's strategic direction, or even something as simple as a mistake in the requirements-gathering phase could have serious repercussions for the downstream process.
    - Work must be estimated early — sometimes years before the actual work is to be done, and usually by someone who won't actually do the work.
      * Creates a house of cards in which one wrong estimate can again wreak havoc across the rest of the project plan
    - The architects aren't blameless either.
      * This era led to “ivory tower architects” who created designs for applications that in practice were impractical or, in some cases, impossible.
    - Developers didn't help the case either.
      * Many of them simply carried out the architect's design vision, whether or not it made sense.
    - Many times what was delivered to the business (two years after it had been requested) did not remotely resemble what was wanted or needed.
  + To solve some of the issues with waterfall, some development shops turned to the concept of *iterative* or *incremental* development.
    - Idea was to take a large waterfall project and divide it into several smaller waterfall projects.
    - Each subproject would have a defined scope and delivery target and upon completion would feed into the next iteration of the larger project.
    - This was an improvement, because it resulted in smaller projects that were easier to define and got software in front of users much faster.
    - Eventually this eveloved into agile methodologies (see lecture on scrum for more details).

## Concepts behind TDD

* History of TDD starts in 1999 with a group of developers who championed a set of concepts known as Extreme Programing (XP).
* XP is an agile based methodology that is based on...
  + Recognizing what practices in software development are beneficial and dedicating the bulk of the developers time and effort to those practices under the philosophy “if some is good, more is better.”
* Key component of XP is test-first programming.
* TDD grew out of XP as some developers found they were not ready to embrace some of the more, at the time, radical concepts, yet found the promise of improved quality that was delivered by the practice of TDD compelling.
* *TDD as a design methodology*
  + When used as an application design methodology...
    - TDD works best when the business user is engaged in the process to help the developer define the logic that is being created.
    - This is necessary to ensure that the developers understand the business requirements behind the feature they are developing.
    - Ensures that the final product is in line with the needs of the business.
    - Helps ensure that the scope of the feature is adhered to and helps the developer understand what *done* really means with respect to the current feature in development.
* *TDD as a development practice*
  + As a development practice, TDD is deceptively simple...
    - In TDD you start by writing a test (instead of starting by writing a feature followed by a test).
    - **Known as *test first development******.***
    - Initially it might seem a bit awkward.
    - However, by writing your test first, what you really are doing is creating the requirement you are designing for in code.
    - You create an executable version of the requirement that is composed of your test.
    - Until these tests pass, your code does not satisfy the business requirement.
  + Your first test...
    - When you write your first test, the first indication that it fails is the fact that the application does not compile.
    - This is because your test is attempting to instantiate a class that has not been defined, or it wants to use a method on an object that does not exist.
    - The first step is simply to create the class you are testing and define whatever method on that class you are attempting to test.
    - At this point your test will still fail, because the class and method you just created don't do anything.
    - The next step is to write just enough code to make your test pass.
      * Should be the simplest code you can create that causes the test to pass.
      * Goal is **not** to write code based on what might be coming in the requirement.
      * Until that requirement changes, or a test is added to expose that lack of functionality, it doesn't get written.
      * This prevents you from writing overly complicated code where a simple algorithm would suffice.
      * *One of the goals of TDD is to create code that is easy to understand and maintain.*
  + *As soon as your first test is passing, add more tests...*
    - You should try to have enough tests to ensure that all the requirements of the feature being tested are being met.
    - As part of this process, you want to ensure that you are testing your methods for multiple input combinations.
    - This includes values that fall outside the approved range.
      * These are called *negative tests*.
    - e.g. If your requirement says that your interest calculation method should handle only percentage rates up to 20%...
      * see what happens if you try to call it with 21%.
    - e.g. If your method takes string arguments...
      * what happens if you pass in an empty string?
  + **When the entire requirement has been expressed in tests, and all the tests pass, you're done.**

## Benefits of TDD

Creating test code *seems* like a long and convoluted process. There are benefits, however:

* *TDD ensures quality code from the start.*
  + Developers are encouraged to write only the code needed to make the test pass and thus fulfill the requirement.
  + If a method has less code, it's only logical that the code has fewer opportunities for error.
* *TDD is more likely to result in code that follows the SOLID principles (because code that is not easily testable likely violates these principles).*
  + **SOLID**: Acronym for five basical principles of object-oriented programming and design:
    - **Single responsibility principle**
      * A class should have only a single responsibility.
    - **Open-closed principle**
      * Software entities should be open for extension, but closed for modification.
    - **Liskov substitution principle**
      * Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.
    - **Interface segregation principle**
      * Many client-specific interfaces are better than one general-purpose interface.
    - **Dependency inversion principle**
      * One should “depend on abstractions instead of depending on concretions.
* *TDD ensures a high degree of fidelity between the code and the business requirements.*
  + If your requirements are written as tests, and your tests all pass, you can say with a high degree of confidence that your code meets the needs of the business.
* *TDD encourages the creation of simpler, more focused libraries and APIs.*
  + TDD turns development a bit on its head, because the developer writing the interface to the library or API is also its first consumer.
  + Gives you a new perspective on how the interface should be written, and you know instantly if the interface makes sense.
* *TDD encourages communication with the business.*
  + To create these tests, you are encouraged to interact with the business users.
  + This way, you can make sure that the input and output combinations make sense, and you can help the users understand what they are building.
* *TDD helps keep unused code out of the system.*
  + Most developers have written applications in which they designed interfaces and wrote methods based on what might happen.
  + Leads to systems with large parts of code or functionality that are never used.
  + This code is expensive.
  + You expend effort writing it, and even though that code does nothing, it still has to be maintained.
  + It also makes things cluttered, distracting you from the important working code.
* *TDD provides built-in regression testing.*
  + As changes are made to the system and your code, you always have the suite of tests you created to ensure that tomorrow's changes do not damage today's functionality.
* *TDD puts a stop to recurring bugs.*
  + You've probably been in situations where you are developing a system and the same bug seems to come back from QA repeatedly.
  + You think you've finally tracked it down and put a stop to it, only to see it return two weeks later.
  + With TDD, as soon as a defect is reported, a new test is written to expose it.
  + When this test passes, and continues to pass, you know the defect is gone for good.
* *When developing applications with testability in mind, the result is an architecture that is open, extensible and flexible.*
  + Dependency Injection (covered later) is a key component of both TDD and a loosely coupled architecture.
  + This results in a system that by virtue of its architecture is robust, easy to change, and resistant to defects.

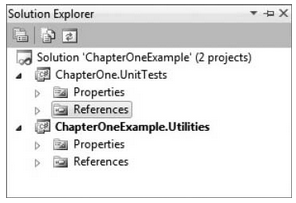
## Quick example of TDD

Example of what it's like to develop a feature for a system using TDD.

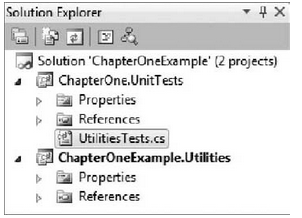
* You have been asked to create a feature that counts occurrences of a character in a string.
* You are working in an existing solution, with an existing project structure.
* The class you'll implement this method on does not exist.
* Assume for this example that your unit-testing frameworks have been referenced in your project.
  + *I’ll cover unit-testing frameworks in more detail later.*

Writing the code:

1. The current C# solution:



1. First step is to create a class in our unit test project that will contain our unit tests:



1. Create test class to hold the tests:

**namespace ChapterOne.UnitTests**

**{**

**public class UtilitiesTests**

**{**

**}**

**}**

1. Write your first test.
   1. This test passes in the string mysterious and asks the library to count the occurrences of the letter y

using NUnit.Framework;

namespace ChapterOne.UnitTests

{

public class UtilitiesTests

{

[Test]

public void ShouldFindOneYInMysterious()

{

// Arrange

var stringToCheck = “mysterious”;

var stringToFind = “y”;

var expectedResult = 1;

var classUnderTest = new StringUtilities();

// Act

var actualResult =

classUnderTest.CountOccurences(stringToCheck,

stringToFind);

// Assert

Assert.AreEqual(expectedResult, actualResult);

}

}

}

1. Compile & run the tests.
   1. Code does not compile because you have not yet implemented the class “StringUtilities”.
2. Make the code compile.
   1. Add a new class called StringUtilities with a method called CountOccurances matching the required interface.

using System;

namespace ChapterOneExample.Utilities

{

public class StringUtilities

{

public int CountOccurences(string stringToCheck,

string stringToFind)

{

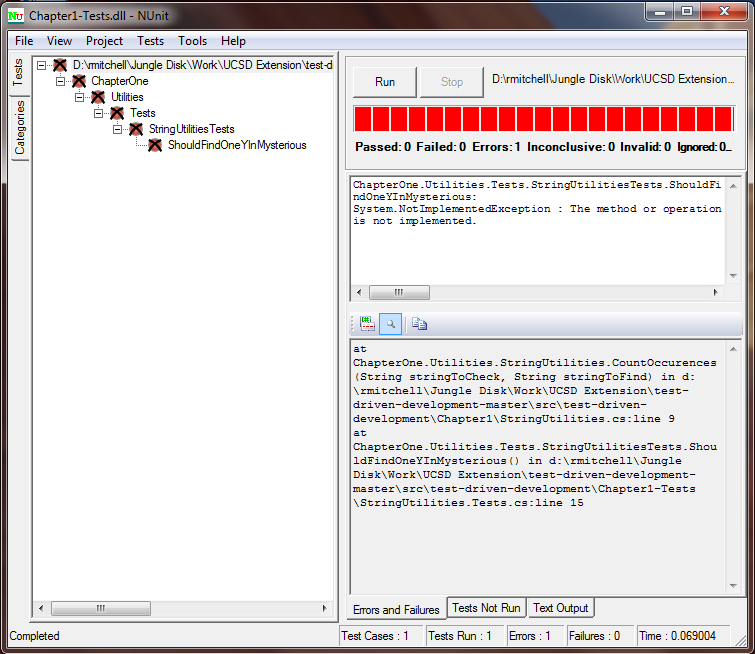
**throw new NotImplementedException**();

}

}

}

1. Compile & run the tests.
   1. Code compiles.
   2. Test fails because NotImplementException is thrown.



* 1. Might seem silly to take each of these steps.
     1. It’s important to see your tests fail before you write your methods.
     2. This ensures that you are writing only enough code to make the test pass.

1. Write code to make this test pass:

using System;

namespace ChapterOneExample.Utilities

{

public class StringUtilities

{

public int CountOccurences(string stringToCheck,

string stringToFind)

{

var stringAsCharArray = stringToCheck.ToCharArray();

var stringToCheckForAsChar =

stringToFind.ToCharArray()[0];

var occuranceCount = 0;

for (var characterIndex = 0;

characterIndex < stringAsCharArray.GetUpperBound(0);

characterIndex++)

{

if (stringAsCharArray[characterIndex] ==

stringToCheckForAsChar)

{

occuranceCount++;

}

}

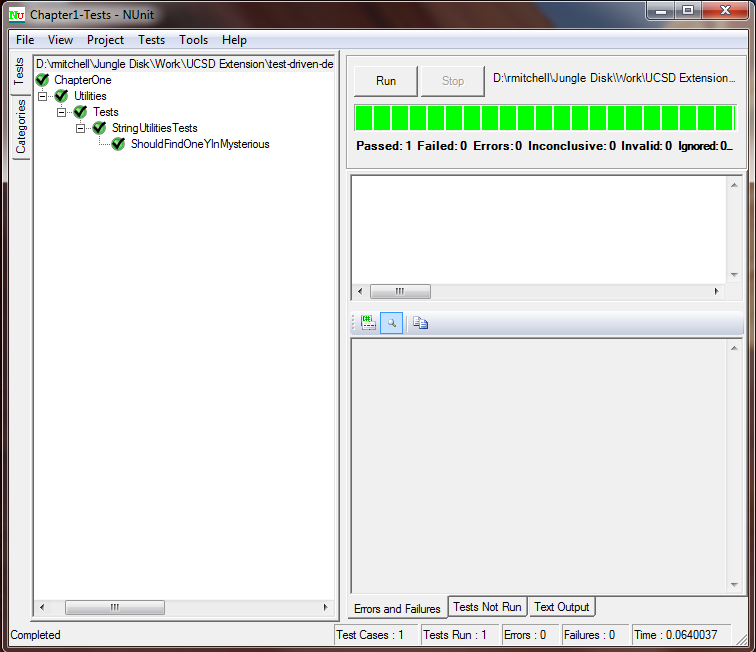
return occuranceCount;

}

}

}

1. Compile & run tests.
   1. Code compiles.
   2. Tests pass.



1. Code working, let’s add test to verify that it finds multiple instances of a character.

[Test]

public void ShouldFindTwoSInMysterious()

{

// Arrange

var stringToCheck = “mysterious”;

var stringToFind = “s”;

var expectedResult = 2;

var classUnderTest = new StringUtilities();

// Act

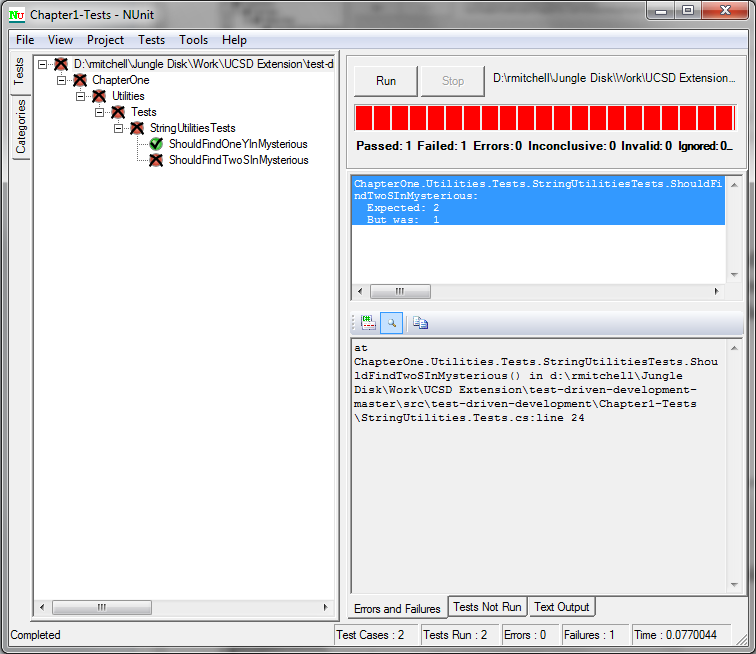
var actualResult = classUnderTest.CountOccurences(stringToCheck,

stringToFind);

// Assert

Assert.AreEqual(expectedResult, actualResult);

}

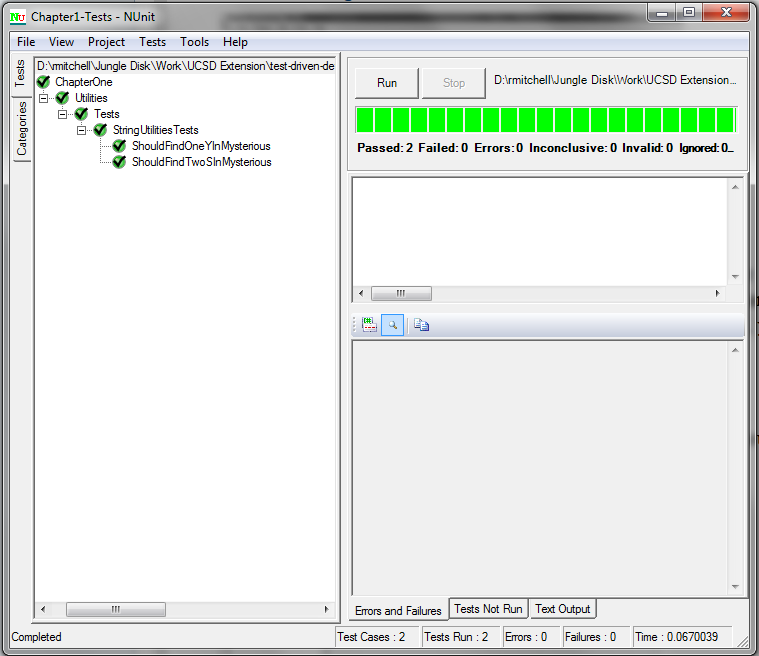


1. Fix bug in code.
   1. for loop is looping through the target string one fewer time than is needed (string length – 1).

for (var characterIndex = 0;

characterIndex <= stringAsCharArray.GetUpperBound(0);

characterIndex++)



1. You are given a new requirement...
   1. Business user wants search to be case-insensitive.
   2. First step is to write a test that expresses this new requirement.

[Test]

public void SearchShouldBeCaseSenstive()

{

// Arrange

var stringToCheck = “mySterious”;

var stringToFind = “s”;

var expectedResult = 2;

var classUnderTest = new StringUtilities();

// Act

var actualResult =

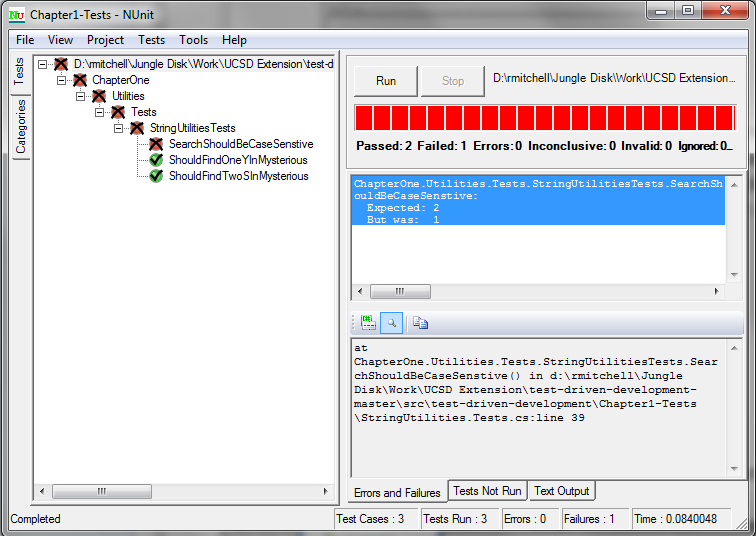
classUnderTest.CountOccurences(stringToCheck,

stringToFind);

// Assert

Assert.AreEqual(expectedResult, actualResult);

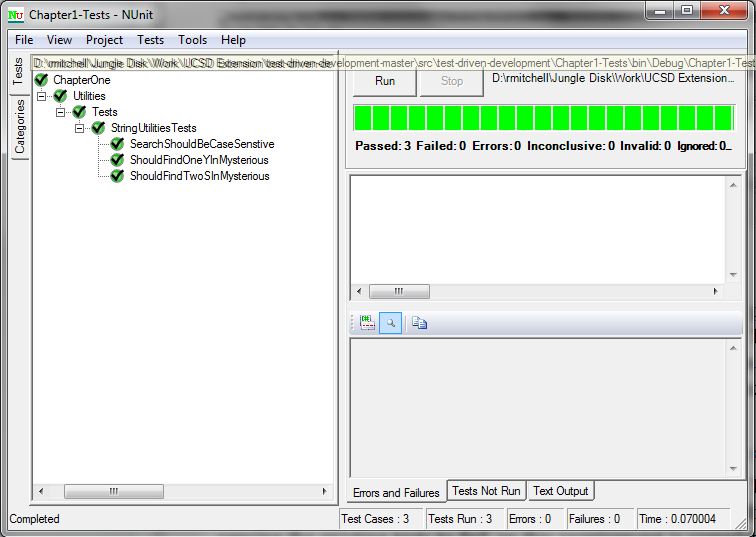
}



1. Implement code to make new test pass.
   1. Convert both the string you are searching and the character you are searching for to uppercase before running the search algorithm.

var stringAsCharArray = stringToCheck.ToUpper().ToCharArray();

var stringToCheckForAsChar = stringToFind.ToUpper().ToCharArray()[0];



1. You release first version of your string utility class and soon you receive your first defect.
   1. When user passes in null as string to be searched a null reference exception is thrown.
   2. You discuss with business user what would be appropriate result – it is decided that a value of -1 should be returned in this case instead of a null reference exception being thrown.
   3. First, you write a test to expose the bug.

[Test]

public void ShouldBeAbleToHandleNulls()

{

// Arrange

string stringToCheck = null;

var stringToFind = “s”;

var expectedResult = -1;

var classUnderTest = new StringUtilities();

// Act

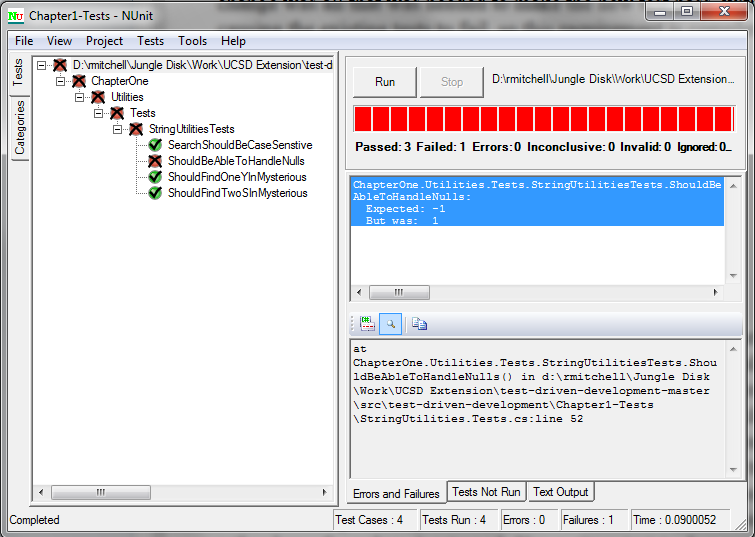
var actualResult = classUnderTest.CountOccurences(stringToCheck,

stringToFind);

// Assert

Assert.AreEqual(expectedResult, actualResult);

}



1. Update code to make test pass.
   1. Validate incoming arguments and return appropriate response if data fails validation.

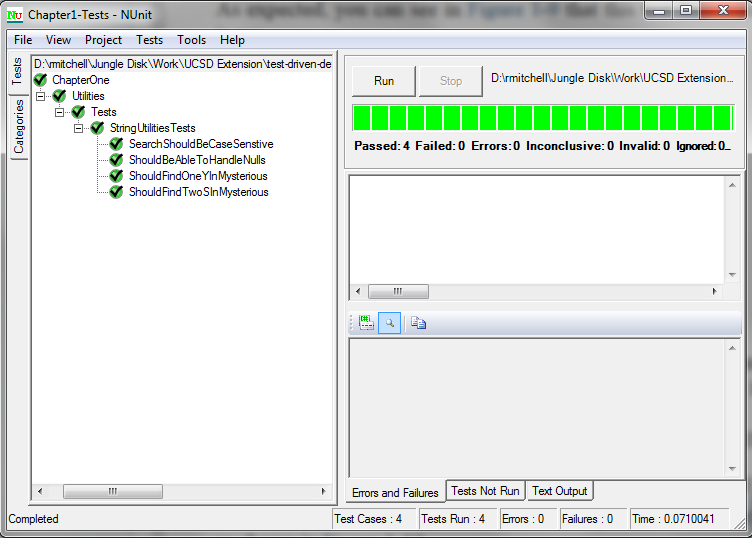
public int CountOccurences(string stringToCheck, string stringToFind)

{

if (stringToCheck == null)

return -1;

var stringAsCharArray = stringToCheck.ToUpper().ToCharArray();



* 1. Test verifies bug is fixed...
     1. ...and ensures that bug doesn’t reappear in the future

# Introduction to Unit Testing

## What is a unit test?

Many different definitions have arisen for the term “Unit test”.

* Component test
* Integration test
* User acceptance test (UAT)

**Unit test definition**

* Test designed to test one unit of work
  + One requirement for one method

**Benefits of working writing tests at this level**

* Test is confined to only one fact for one unit of work
* You don’t have to know details of rest of system (unless you have dependencies on these parts of system)
* Writing tests is easy since they are so granular
* If a test fails you can be reasonably sure the issue lies within your unit of work
  + You don’t have to “go down the rabbit hole” searching for which layer of code has the issue

**All unit tests share same characteristics**

* Isolated from other code
* Isolated from other developers
* Targeted
* Repeatable
* Predictable

**What is not a unit test?**

* *Tests that cross boundaries between code you are testing and other parts of system*
  + Includes everything from other classes and services in the same app to databases, web services, and other external depencies
  + When your tests start to bleed into these other classes, services, and systems, your tests start to lose their focus, and when they fail, it becomes more difficult to target the defective code.
  + The idea behind a unit test is that you test only what's in that method.
  + When it fails, you don't want to have to hunt through several layers of code, database tables, or the documentation for third-party products to find a possible answer.
* *Tests that test more than one thing*
  + Unit tests should test only one thing.
  + According to the Single Responsibility Principle each class or method should have one reason to change.
  + A logical and practical extension of this principle is that each class and method should have one purpose and do only one thing.
  + As a result, each unit test should test only one set of requirements.
  + In the StringUtility example, the method did one thing: It counted the occurrences of a particular character in a string.
    - It didn't tell you where the character was found in the string, if the instances were of the same case, or any information other than how many occurrences of the character there were.
    - According to the Single Responsibility Principle, those functions should be served by other methods in the StringUtility class.
  + **If you find yourself in a situation where your tests are testing more than one thing, you probably have violated the Single Responsibility Principle.**
    - What this means in terms of unit testing is that you have unfocused tests that are difficult to understand.
    - Also, your tests are likely to become brittle and fragile over time as more inappropriate functionality is added to the method or class.
    - Diagnosing problems also becomes a challenge, because your functionality can take multiple paths.
      * Determining which of these paths contains the defect can be difficult.
      * Many times the interaction of these branches can be the cause of the defects.
    - Defects of this type are also difficult to fix, because a change to one branch can cause unpredictable behavior in the other.
* *Tests that are unpredictable*
  + Unit tests should be predictable.
  + When you call a method on a class for a given set of input parameters, the result should always be consistent.
  + Sometimes this principle may seem hard to adhere to.
    - For example, if you are writing a commodity trading application, it's likely that the price of gold will be one value at 9 a.m. and a different value at 2:30 p.m.
  + There are also potential situations where you may need to build the ability to create randomization functionality into your system.
  + In these cases a good design principle is to abstract the functionality that provides unpredictable data into another class or method that can be mocked in your unit test.
    - Mocking is discussed further later
    - In the case of your commodity system, the service that provides the time-sensitive price should be mocked.
      * Depending on your needs, you may want that test implementation to always return the same value.
      * Or you may want it to provide a predetermined value given a specific time.
    - In the case of randomization, it would be a good design practice to wrap the randomization in a service class that can also be mocked by your tests to always return the same value or sequence of values.
    - **These predictable sets of data allow you to write tests that are specific and exercise your code effectively without your having to worry about getting different results in each test run from dependencies you have on other parts of the system.**

## NUnit (a unit testing framework)

* Many tools can help you perform unit testing.
* Since TDD's rise in popularity in 2005, the market for frameworks and tools has exploded.
* One of the most popular types of unit testing tools are unit test frameworks.
* These frameworks let you define your unit test code, control the execution of the tests, and provide an application to run the tests and report on the success of each test in your test suite.
* NUnit is one of the most popular and mature of the .NET unit test frameworks.
  + Note, NUnit is based heavily on JUnit, the original unit test framework written for Java.

**What is a unit test framework?**

* Before unit test frameworks, developers had a difficult time creating executable tests.
* The initial practice was to create a window in your application that was dubbed the “test harness.”
  + This was simply a window with a button for each test. The results of the tests were either a message box or some sort of readout on the form itself.
  + With a button for each test, these windows soon became crowded and unmanageable.
* Some enterprising developers moved to using console applications that would execute each test and output the results to the console.
  + This was a step forward in that developers could spend more time on their test and less on their test harness.
  + It also had the benefit that you could easily have it run automatically from a build server.
* These methods worked to a degree...
  + But they didn't provide a common means for creating, executing, and interrogating tests.
* Unit test frameworks such as NUnit sought to provide those features.
  + Unit test frameworks provide a unified programming model to define your tests as simple classes with methods that call the application code that you want to test.
  + Developers do not need to write their own test harness; the unit test frameworks provide test runners that allow you to execute all your tests with the click of a button.
  + With a unit test framework, you can easily insert, set up, and tear down functionality around your tests.
  + When a test fails, the test runner provides you with information about the failure, including any exception information that is available and a stack trace.

**The basics of NUnit**

Most unit tests you write will have a very simple pattern:

* Perform some activity to set up your test.
* Execute your test.
* Verify its result.
* If necessary, reset your environment.

Your tests themselves are methods that execute and call the methods you are testing.

* These methods must reside in a class, which is called a test fixture.
* The following example shows a class using the TestFixture attribute to indicate that it is a test class, and the Test attribute to indicate the method that is our test

namespace NUnitExample

{

[TestFixture]

public class ExampleTests

{

[Test]

public void TestMethod()

{

Debug.WriteLine(“This is a test”);

}

}

}

Occasionally some setup needs to happen before you run your test, such as populating a dataset, instantiating a class, or setting up an environment variable.

* In such a case, you can use the Setup attribute to define a method that will execute before your test runs

namespace NUnitExample

{

[TestFixture]

public class ExampleTests

{

private string \_testMessage;

[SetUp]

public void SetupForTest()

{

\_testMessage = “This is a test.”;

}

[Test]

public void TestMethod()

{

Debug.WriteLine(\_testMessage);

}

}

}

Of course, if you can have setup code for your tests, it only makes sense that you have code to reset resources used by your tests when they are finished.

namespace NUnitExample

{

[TestFixture]

public class ExampleTests

{

private string \_testMessage;

[SetUp]

public void SetupForTest()

{

\_testMessage = “This is a test.”;

}

[Test]

public void TestMethod()

{

Debug.WriteLine(\_testMessage);

}

[TearDown]

public void TearDownAfterTest()

{

\_testMessage = string.Empty;

}

}

}

Right now the test executes, but it doesn't really test anything.

* You can change that by using an *assert* — a way of telling the test runner application the final result of the test.
* Many different types of asserts available in NUnit test conditions such as
  + Equality of two values
  + Whether two reference type variables point to same object
  + Whether various conditions are met

Example, test verifies whether the length of \_testMessage is greater than 0:

[Test]

public void MessageLengthGreaterThanZero()

{

if (\_testMessage.Length > 0)

{

Assert.Pass();

}

else

{

Assert.Fail();

}

}

Other types of assertions in NUnit:

* **Assert.AreEqual(expected, actual)**
  + This method is overloaded to take any type of value for expected or actual, so long as both arguments are of the same type.
* **Assert.AreSame(expected, actual)**
  + This assert takes reference types as its arguments.
  + It is used to determine whether the object passed in for expected and the argument passed in for actual are the same object.
  + This means that both objects occupy the same space in memory and are not simply copies of one another.
* **Assert.IsTrue(bool)**
  + This assert takes either a Boolean variable or a logical condition that can be evaluated to a Boolean result.
* **Assert.IsNull(object)**
  + This assert will examine a reference type and determine whether it is null.
* **Assert.Greater(x,y)**
  + Evaluates either two value types or two reference types that implement the IComparable interface to determine if x is greater that y (x > y) or if x is greater than or equal to y (x >= y).
* **...and many more**

## Decoupling with mock objects

*A well-written piece of software tries to limit dependencies.*

* There comes a point, however, where each of your components must be coupled with another to form a greater whole.
* These couplings create a web of dependencies within your application that usually end in an external resource, be it a database, web service, file system, or other resource.
* Mock objects are designed to stand in for these other components in your application and the external resources they sometimes represent.
* This allows you to test your code without having to worry about the consequences of interacting with other resources.

**Why mocking is important**

* *When you write unit tests for a method, your intention is to test only the code in that method.*
  + This is by design; you are attempting to isolate the code under test.
  + The reason for this is that you want to assess the validity of only that piece of code for a given condition.
  + A condition can be defined as not only the input data for that method, but also the context and environment that the code will execute in.
  + By isolating that code, you can ensure that any failing tests point squarely to a problem with that specific method, not a method in a far-flung corner of the system.
  + It's okay to assume that the other components your method works with are correct and have been tested themselves and that you can rely on the quality of these components and tests.
  + **You care about only the method in front of you at that point in time.**
* *Unit tests should also be able to be run quickly.*
  + Even in a small application, if TDD is applied correctly, is it not unusual to have hundreds of unit tests.
  + Your goal is to have developers run these tests frequently during their development
    - Ensures that changes they are making to their methods and classes are not negatively impacting other parts of the system.
    - If only a small amount of work is done between a successful test and a failed test, you need to examine only that small amount of work to determine what went wrong.
    - If these tests take several hours or even just minutes to run, they will not be run frequently enough.
    - Most tests that take a long time to run do so because they are interacting with an external resource, such as a database.
    - **A test that is never run is worthless.**
* *Unit tests should be predictable and consistent.*
  + When writing a method and its corresponding tests, you should be able to be certain that if your method receives X and Y as arguments, Z will always be returned.
  + External resources, and the data they contain, change over time.
  + If your method and the test that invokes it rely on an external resource, you can't guarantee that you will always get the same results for the same input parameters.
  + By mocking this resource, you can be sure that you're always getting consistent and reliable data to test your method against.
* *Unit tests should be able to run without impacting the test results of your teammates.*
  + Most development projects are undertaken by teams of more than one person.
  + Perhaps your test suite contains tests that change the state of a database table and other tests that rely on the data in that table.
  + If so, there is a good chance that if two or more developers run the tests concurrently, they will all have failing tests.
  + They are stepping on each other's data, and none are getting the data they are expecting.
  + Mocking these external resources ensures that you won't cause tests run by other developers to fail based on incorrect or unexpected data.

**Dummies**

Mock is somewhat of a generic term that covers a family of stand-in objects for use in unit testing.

* Dummy objects are simple mocks that stand in for an external resource.
  + They usually return a predefined response for a method when that method is invoked
  + They usually can't vary that response based on the input parameters.
  + Many developers who don't want to incur the overhead of a mocking framework (and don't need the functionality provided by one) use hand-rolled dummy objects in their tests.

**Hand-rolling your own dummy**

The examples in this section, include tests for a class called DependentClass.

* DependentClass has a dependency on a class that implements the interface IDependency.
* DependentClass has a method called get GetValue that takes a string as a parameter.

Here is the definition of IDependency and DependentClass:

internal interface IDependency

{

int GetValue();

}

internal class DependentClass

{

private readonly IDependency \_dependency;

public DependentClass(IDependency dependency)

{

\_dependency = dependency;

}

public int GetValue(string s)

{

return \_dependency.GetValue(s);

}

}

Test that tests the implementation of DependentClass when it has been passed an instance of DummyDependency, a dummy object that implements the IDependency interface:

[TestFixture]

public class DummyTestClass

{

[Test]

public void TestWithADummy()

{

var dependency = new DummyDependency();

var dependentClass = new DependentClass(dependency);

const string param = “abc”;

const int expectedResultOne = 1;

var resultOne = dependentClass.GetValue(param);

Assert.AreEqual(expectedResultOne, resultOne);

}

}

public class DummyDependency : IDependency

{

public int GetValue(string s)

{

return 1;

}

}

* You can easily see the limitations of the dummy object:
  + In spite of whatever value is passed in by the test, the dummy object can react only one way; by returning a value of one.
  + There are occasions where this limitation is not an issue.
  + But for most tests that are verifying business domain logic, a more robust means of mocking is necessary.

**Fakes (AKA stubs)**

* Fakes and stubs are a step up from dummy objects in that they can vary their response based on input parameters.
* For example, a stub of a database may return the name Rick Nash for user ID 61 and the name Steve Mason for user ID 1.
  + Aside from that, no logic is invoked.
* A stub generally cannot track how many times a method was called or in what order a sequence of methods were called.

Example stub:

[TestFixture]

public class StubTestClass

{

[Test]

public void TestWithAStub()

{

var dependency = new StubDependency();

var dependentClass = new DependentClass(dependency);

const string param1 = “abc”;

const string param2 = “xyz”;

const int expectedResultOne = 1;

const int expectedResultTwo = 2;

var resultOne = dependentClass.GetValue(param1);

var resultTwo = dependentClass.GetValue(param2);

Assert.AreEqual(expectedResultOne, resultOne);

Assert.AreEqual(expectedResultTwo, resultTwo);

}

}

public class StubDependency : IDependency

{

public int GetValue(string s)

{

if (s == “abc”)

return 1;

if (s == “xyz”)

return 2;

return 0;

}

}

* This stub is able to respond to different stimuli in different specific ways.
* This provides a much more robust way of mocking than dummy objects.

**Mocks – a stup up from fakes & stubs**

*Mocks provide the same functionality as stubs but are more complex.*

* They can have rules defined for them that dictate in what order methods on their API must be called.
* Most mocks can track how many times a method was called and can react based on that information.
* Mocks generally know the context of each call and can react differently in different situations.
* Because of this, mocks require some knowledge of the class they are mocking.

*Example mock:*

* In many APIs you work with as a developer, specific methods have to be called in a specific order and methods need to be called a specific number of times.
  + The method CallMeFirst must be called first
  + The CallMeTwice method must be called two times
  + The CallMeLast method must be the last method called for a particular transaction.

internal interface IDependency

{

int GetValue(string s);

void CallMeFirst();

int CallMeTwice(string s);

void CallMeLast();

}

internal class DependentClass

{

private readonly IDependency \_dependency;

public DependentClass(IDependency dependency)

{

\_dependency = dependency;

}

public int GetValue(string s)

{

return \_dependency.GetValue(s);

}

public void CallMeFirst()

{

\_dependency.CallMeFirst();

}

public void CallMeLast()

{

\_dependency.CallMeLast();

}

public int CallMeTwice(string s)

{

return \_dependency.CallMeTwice(s);

}

}

To enforce these rules, you need to write a somewhat more sophisticated and complex mocking class that the previous two examples:

public class MockDependency : IDependency

{

private int \_callMeTwiceCalled;

private bool \_callMeLastCalled;

private bool \_callMeFirstCalled;

public int GetValue(string s)

{

if (s == “abc”)

return 1;

if (s == “xyz”)

return 2;

return 0;

}

public void CallMeFirst()

{

if ((\_callMeTwiceCalled > 0)|| \_callMeLastCalled)

throw new AssertionException(“CallMeFirst not first method called”);

\_callMeFirstCalled = true;

}

public int CallMeTwice(string s)

{

if (!\_callMeFirstCalled)

throw new AssertionException(

“CallMeTwice called before CallMeFirst”);

if (\_callMeLastCalled)

throw new AssertionException(“CallMeTwice called after CallMeLast”);

if (\_callMeTwiceCalled >= 2)

throw new AssertionException(“CallMeTwice called more than twice”);

\_callMeTwiceCalled++;

return GetValue(s);

}

public void CallMeLast()

{

if (!\_callMeFirstCalled)

throw new AssertionException(

“CallMeLast called before CallMeFirst”);

if (\_callMeTwiceCalled !=2 )

throw new AssertionException(

string.Format(

“CallMeTwice not called {0} times”, \_callMeTwiceCalled));

\_callMeLastCalled = true;

}

}

* To be sure that the methods of the implementation of IDependency used by the DummyClass are used correctly
  + it's necessary to build a mock that not only returns values, but encapsulates all the rules of the API.

*Hand-rolling mocks is inefficient:*

* Time consuming
* Introduces brittleness to the codebase
* *It is for these reasons that most developers choose to employ a mocking framework and avoid hand-rolling mocks.*

*Test using MockDependency:*

[TestFixture]

public class MockTestClass

{

[Test]

public void TestWithAMock()

{

var dependency = new MockDependency();

var dependentClass = new DependentClass(dependency);

const string param1 = “abc”;

const string param2 = “xyz”;

const int expectedResultOne = 1;

const int expectedResultTwo = 2;

dependentClass.CallMeFirst();

var resultOne = dependentClass.CallMeTwice(param1);

var resultTwo = dependentClass.CallMeTwice(param2);

dependentClass.CallMeLast();

Assert.AreEqual(expectedResultOne, resultOne);

Assert.AreEqual(expectedResultTwo, resultTwo);

}

}

**Should you use a dummy, stub, or mock?**

The answer is that it depends.

* In general, you'll want to favor fakes and stubs.
* Mocks are useful when you need to replicate a more complex interaction with a component,
  + but they usually require more configuration and overhead than is needed for most unit tests.
* Favoring stubs also ensures that you are designing your system to be loosely coupled;
  + requiring a calling method to know an extensive set of rules to use an API is not very loose.

**Best and worst practices**

* *Dependency Injection*
  + Mocking is a key concept that makes TDD a viable way of writing software.
  + To use mocking effectively, your application should use Dependency Injection.
  + In short, this means that instead of statically creating objects that your class is dependent on as part of the class's internal instantiation process...
    - ...you should provide the class with instances of those objects that conform to the interface needed by the dependency.
  + This makes it easy to substitute mocked objects for actual objects that would be used by the fully integrated application.
* *Design for the interface, not the implementation*
  + When consuming another class or resource as a dependency, your concern should not be how it performs its tasks, just what the interface is.
  + You should use interfaces when designing and building your service's classes to abstract the functionality from the API.
  + This design not only makes your code less brittle and more open for extension, but it also makes mocking easier and more efficient.
* *Try to limit dependencies*
  + Most code needs to be dependent on something.
  + It could be a database to store and retrieve data, or a web service to authenticate a user, or another domain service in your application.
  + You should work to limit the things that your code is dependent on.
  + Not only do a large number of dependencies signify a brittle system,
    - but such a system also is more difficult to mock and test effectively.
* *Do not mock private methods*
  + When you are writing a test and your code is dependent on another class, you should mock only the public methods.
  + Even then, you should mock only those you will use directly; don't overmock.
  + Mocking private methods requires knowledge of the internal function of the service you are mocking -
    - knowledge that, according to the encapsulation rule, you should not have.
  + As a consumer of a service, you should only concern yourself with the methods on the public interface
* *Don’t cheat*
  + As you continue with your practice of TDD, you'll be tempted at times to take shortcuts with your mocking.
  + Maybe it's because a particular dependency requires a mock that is a little more complicated than you're used to.
  + Maybe the stubbed method must return a complex data or object graph that you just don't feel like creating.
  + Don't let yourself get caught in that trap.
  + Tests form the quality baseline for your applications.
  + Your tests rely on mocks and stubs to ensure that they can correctly interact with the various dependencies your system will contain.
  + If you take shortcuts with your mocks, your tests and, by extension, your software will suffer.

# Refactoring With Confidence

## What is Refactoring?

*Refactoring* is the act of changing the internal implementation of a class or method with the aim of making the code more readable and maintainable.

* Refactoring also reduces the code's overall complexity **without changing the external behavior of the class or method.**
* These alterations can be as simple as changing the name of a method or variable to moving methods from one class to another or even splitting large classes into several smaller ones.
* Refactoring allows you to continuously change and improve your code.

## Why Refactor?

When practicing TDD, the goal when initially writing a method or creating a class is to simply make the test pass and nothing more.

* At this point you're not necessarily looking for style points or to make your code reusable or elegant.
* You're simply trying to make the tests “all green,” meaning that the tests all pass.
* Once you've accomplished this, the next step is to improve your code.
* In practice this is a very practical approach.
* Many developers spend too much time trying to make their code elegant and beautiful the first time through.
* They end up missing some important pieces of business functionality that they then have to somehow work into their code.
* By starting from a point of complete business functionality before making your code beautiful, you ensure that the top priority — working business code — is met before anything else.

**Fearless Refactoring**

* *Unit tests help you make sure that no matter what you change in the name of refactoring, your code still meets the business need.*
* This is the genesis of the term *fearless refactoring*.

**“Test-first” is important**

*Test-first* can be a little difficult to become comfortable with

* T**est-first is the most important tenet of TDD**
* By employing this test-first philosophy,
  + you can ensure that you never write code that does not add some value to the application.
* Once you have a test, your goal is to write just enough code to make that test pass.
  + *No more, no less.*
* Your strategy should be how to get from point A (a failing test) to point B (all tests passing) with the shortest and straightest line possible.
* Don't worry about other aspects of the system yet.
* Also, don't worry about an aspect of the feature that is not in the tests but that you are almost certain the business will want.
  + For example; a feature specifies that a method take an integer as an input parameter.
  + You *just know* that that input parameter has be a value of 100 or less.
  + Your test should *only* test for numbers; there is nothing in the feature about limiting to numbers with values of 100 or less.
  + Until that feature gets scheduled, and a unit test is created for it, it's not a requirement.
  + What happens if that feature never actually gets scheduled?
    - At best you've written code that provides no value but that must be maintained along with the rest of the code base.
    - At worst you've needlessly overcomplicated what should have been a simple piece of functionality.
  + The software development term **YAGNI stands for You Aren't Going to Need It.**
    - If that feature gets scheduled, at that point you can write those tests and the necessary code to make them pass.
    - For now, worry only about the test for the current feature you are working on.

**Refactor after your tests pass**

* After you've concluded that your test passes and you haven't broken anything else in the application (all tests pass), it's time to look at potential areas to improve.
  + Maybe variable or method names need to be changed
  + Maye you’ve duplicated functionality in your new code
  + Maybe you have a function that is a bit long, and breaking it up would improve its readability.
  + *It’s time to refactor...*

**How to refactor**

* Start by making a small improvement to your code, such as renaming a variable.
* Once you've done that, run your tests to verify that the external functionality of your class or method has not changed.
* Don't run the tests for only the code you just wrote; run them all.
  + This is necessary to ensure that you have not caused an adverse side effect somewhere else in the code base.
* When you're satisfied that your change has not had any adverse effect, improve the next issue with your code, and run the tests again.
* Repeat as needed.
* **This practice is known as “red, green, refactor.”**
  + This means that you start with a test that fails because you haven't implemented the logic to make it pass yet.
  + Then you implement the test so that it passes.
  + Finally, you refactor to improve the code while not breaking the test.

**Refactor to improve maintainability**

* When practicing TDD, your initial goal when writing code is simply to make the tests pass.
* As time goes on and the application and the business it supports evolve, the application must change to either support new functionality,
  + or change how the current business logic performs its function.
  + *The ability to quickly and easily make these changes is important.*
* Refactoring code for readability is an important step in managing an application.
  + Not just for other developers, but for ourselves.
  + Every developer has had the unfortunate experience of looking at code he or she wrote in the past and not remembering what, how, or why the code does what it does.
* Refactoring should work to simplify the code wherever possible.
  + This includes making sure that methods are short, control structures are simple, and variable and method names are descriptive and clear.
  + Think of refactoring for maintainability as leaving your future self a hint so that you can remember what motivated you to write the code in the first place.

## Code Smells

**No code is perfect**

* Developers who practice TDD still strive to make their code as good as possible.
* A key skill is the ability to evaluate code and to quickly identify common potential trouble spots without having to run the application.
  + These common problems are called *code smells*.

**What is a code smell?**

Developers have always needed to solve a common recurring series of problems in code.

* These problems eventually found a series of common, widely known, widely used solutions.
* These solutions became known as *patterns*.
* As a corollary, over the many years that applications have been developed, developers have always made many common recurring mistakes.
* These mistakes, and the problems they tend to cause, are called *antipatterns*.
* *Code smells* are simply a collection of commonly known and widely found code-based antipatterns.
* *Refactoring* is the technique used to eliminate code smells and TDD provides the tests required to refactor with confidence.

**Common code smells**

*Duplicate code and similar classes*

public class WidgetService

{

private const double PricePerWidget = 1.5;

public double GetQuoteForWidgets(int quantity)

{

return PricePerWidget \* quantity;

}

public string PlaceOrderForWidgets(int quantity)

{

var invoice = new Invoice

{

TotalPrice = PricePerWidget \* quantity

};

return invoice.InvoiceNumber;

}

}

* The functionality for determining the total cost for an order of widgets is duplicated (X = PricePerWidget multiplied by quantity).
* Violates the SRP principle
  + “Single Responsibility Principle”
* This code smell is not confined to duplicate code in the same class.

public class WidgetService

{

private const double PricePerWidget = 1.5;

public string PlaceOrderForWidgets(int quantity)

{

var invoice = new Invoice

{

TotalPrice =

PricePerWidget\*quantity\*1.15

};

return invoice.InvoiceNumber;

}

}

public class DoDadService

{

private const double PricePerDoDad = 2.25;

public string PlaceOrderForDoDad(int quantity)

{

var invoice = new Invoice

{

TotalPrice =

PricePerDoDad\*quantity\*1.15

};

return invoice.InvoiceNumber;

}

}

* This leads to maintenance and quality issues down the road
  + Must update same logic in many different places
* Solution
  + Refactor common code into single class/method that is used by dependent classes

*Big Classes and Big Methods*

* Bigger is not better.

public string PlaceOrderForWidgets(int quantity, string customerNumber)

{

var invoice = new Invoice

{

InvoiceNumber = Guid.NewGuid().ToString(),

TotalPrice = PricePerWidget\*quantity,

Quantity = quantity

};

var customer = \_customerService.GetCustomer(customerNumber);

invoice.CustomerName = customer.CustomerName;

invoice.CustomerAddress = customer.CustomerAddress;

invoice.CustomerBillingInformation =

customer.CustomerBillingInformation;

double tax;

switch (invoice.CustomerAddress.State.ToUpper())

{

case “OH”:

tax = invoice.TotalPrice\*.15;

break;

case “MI”:

tax = invoice.TotalPrice\*.22;

break;

case “NV”:

tax = invoice.TotalPrice\*.05;

break;

default:

tax = 0.0;

break;

}

var shippingPrice = invoice.TotalPrice \* .1;

invoice.TotalPrice += shippingPrice;

invoice.TotalPrice += tax;

var paymentAuthorizationCode =

\_paymentProcessingService.ProcessPayment(

invoice.TotalPrice,

customer.CustomerBillingInformation);

invoice.Approved = ! string.IsNullOrEmpty(paymentAuthorizationCode);

\_invoiceService.Post(invoice);

return invoice.InvoiceNumber;

}

* May issues with this code
  + Most obvious being how long it is.
  + When you see a method this long, the first thing to look for are violations of the SRP (single responsibility principle).
  + In this case five different business functions:
    - Creates an invoice
    - Associates the invoice with a customer
    - Determines tax based on the customers state
    - Determines shipping costs
    - Authorizes payment
  + At least five different reasons that this method might have to change
  + Unacceptable
    - Hard to test
    - Complex and difficult to follow
    - Breeding ground for bugs
* Solution
  + Extract into multiple methods
    - One for each logic responsibility

*Comments*

* Unless you’re writing low-level assembly / C code where the meaning of the instructions themselves are unclear, comments should be used sparingly
* Lots of comments in the code is an indication that the code needs to be refactored to be cleaner and more self-documenting
  + Well-written code should be easy to follow
  + Comments become a crutch to write sloppy code
  + If you find yourself wanting to add a comment...
    - ask yourself if you could refactor the code to be self-documenting
* Comments are not enforced by the compiler and therefore easily drift out of date relative to the code
* Comments add noise
* In general, comments should only be used to:
  + Document the public / protected API’s of services & libraries
    - These comments should use special syntaxes (e.g. .NET XML doc comments) so that tools can be used to generate public API documentation
    - Do not document the internals of systems in this way – it is overkill
  + Document blocks of code (i.e. algorithms) that are not self-explanatory
* In terms of tracking who changed what in a code base, source control systems do a much better job than comments do.
  + Source control systems, such as Team Foundation Server, Subversion, and Git, can track commits developers make to a repository and what was in that commit better than expecting a developer to leave a comment.

*Bad Names*

public double GetValue(int a, int b)

{

var answer = (a\*a)\*b\*P;

return answer;

}

* What?
* The method name and variable names are poor
* Solution:

public double GetVolumeOfACylinder(int radius, int height)

{

var volumeOfACylinder = (radius\*radius)\*height\*Pi;

return volumeOfACylinder;

}

**Too Much If/Switch**

public double CalculatePrice(int quantity, string cutomerState,

string customerStatus)

{

var basePrice = quantity\*PricePerWidget;

switch (cutomerState)

{

case “OH”:

if (quantity >= 1000 && quantity < 9999)

{

basePrice = basePrice\*.95;

}

else if(quantity >= 10000)

{

basePrice = basePrice\*.90;

}

break;

case “MI”:

switch (customerStatus)

{

case “Premier”:

basePrice = basePrice\*.85;

break;

case “Preffered”:

basePrice = basePrice\*90;

break;

case “Standard”:

basePrice = basePrice\*.95;

break;

}

break;

default:

if (quantity > 10000)

{

basePrice = basePrice\*.95;

}

break;

}

return basePrice;

}

* The high number of evaluations (If and Switch blocks) in this code introduce quite a bit of complexity and decrease the code's readability.
* Even if the simple code in this example revealed a defect it could be difficult to easily diagnose and correct.
* As an added problem, this method spits in the eye of the SRP:
  + This method contains three different pricing algorithms.
* Solution:
  + Extract into multiple methods
    - Use virtual functions with inheritance to run the appropriate logic

**Try/Catch Bloat**

It's easy to get carried away with what gets placed inside a Try/Catch block:

public Customer GetCustomer(string customerId)

{

try

{

var command = new SqlCommand();

var reader = command.ExecuteReader();

var customer = new Customer();

while (reader.Read())

{

customer.CustomerId = customerId;

customer.CustomerName = reader[“CustomerName”].ToString();

customer.CustomerStatus = reader[“CustomerState”].ToString();

customer.LoyaltyProgram =

reader[“CustomerLoyaltyProgram”].ToString();

}

return customer;

}

catch (Exception exception)

{

\_logger.LogException(exception);

var customer = new Customer {CustomerStatus = “unknown”};

return customer;

}

}

* The code in the Try block is doing two separate tasks:
  + creating the ADO.NET command to get the customer from the database
  + mapping the data from the reader to a customer object
* The Catch block contains another violation of SRP
  + The method contains code to handle an exception that may result from the call to the database
  + The current algorithm for handling exceptions that arise when retrieving a customer is to log the exception and then return a customer object with a status of “unknown.”
  + Tomorrow the procedure could be completely different. This represents another reason that this method would have to change.
  + This code all applies only to the attempt to retrieve the customer from the DB, not to map the customer once it’s been retrieved
* Solution:
  + Extract into two methods
    - One to load the customer from the DB
    - Another to map the customer

public Customer GetCustomer(string customerId)

{

try

{

return GetCustomerFromDataStore(customerId);

}

catch (Exception exception)

{

return HandleDataStoreExceptionWhenRetrievingCustomer(exception);

}

}

private static Customer GetCustomerFromDataStore(string customerId)

{

var command = new SqlCommand();

var reader = command.ExecuteReader();

var customer = new Customer();

while (reader.Read())

{

customer.CustomerId = customerId;

customer.CustomerName = reader[“CustomerName”].ToString();

customer.CustomerStatus = reader[“CustomerState”].ToString();

customer.LoyaltyProgram = reader[“CustomerLoyaltyProgram”].ToString();

}

return customer;

}

private Customer HandleDataStoreExceptionWhenRetrievingCustomer(

Exception exception)

{

\_logger.LogException(exception);

var customer = new Customer { CustomerStatus = “unknown” };

return customer;

}

# Conclusion

## Go forth and test!

* Projects that are developed using TDD are robust:
  + Code will more naturally adhere to core software development principles such as SOLID as a side-effect of having to be written in a way that is highly testable
  + Changes can be made to the code with confidence that existing functionality has not been broken since the full suite of tests can be run
  + All code that is written will provide value since no code will be written without a test that verifying “in-scope” business functionality
* Try developing using TDD
  + Once you get used to this methodology you’ll wonder how you ever developed without it