C# 5.0 Concepts

Extension Methods

Overview

- What are they?
- Why use them?
- How to create

What are they?

- A language feature to allow extension of types for which you do not have the source code
- A practical means of providing extensibility to a library
- Only have access to the public elements of the class they are extending

Why use them?

- Extend types for which you do not have code
  - Classes, structs, interfaces, generics
- Separate concerns
  - Fluent interfaces are a good example

Declaration

- Creates in a static class of any name
- Static method, with a first parameter with “this”

```
public static class LegacyExtensions
{
  public static string ToLegacyFormat(this DateTime dateTime)
```

Demo: Creating an extension method

C# 5.0 Concepts

Partial Classes and Methods

Overview

- What are they and why use them?

Partial classes

- Split a class definition across files
- Parts pieced together by the compiler
- Attributes are merged

Why use them?

- Allows multiple developers to code different parts of a class at the same time
- Extensively used by Visual Studio for code generation

```
public partial class Employee
{
    public void DoWork()
    {
    }
}

public partial class Employee
{
    public void GoToLunch()
    {
    }
}
```

Partial Methods

- Must be in a partial class
- One part of the class contains the signature
- Another provides the implementation
- If an implementation is not found, the method is optimized away
- Similar to an event

Why use partial methods?

- Provides an extensibility point to base classes
- Extend the app by providing a partial class with only the methods you need to override implemented
  - All others optimized away
- Used a lot in code generation
  - EF uses this heavily
  - WPF Designer too

Example: Partial classes

.NET for C Programmers

LINQ

Overview

- C# and LINQ
- Operators and Queries
- Time allowing, LINQPad

What is LINQ

- Language INtegrated Query
  - https://msdn.microsoft.com/en-us/library/bb308959.aspx
- Built in compiler support for querying of objects
- Has its own mini-language, as well as set of extension methods
- Facilitates query of objects initiating with IEnumerable

LINQ Syntax

- Query Syntax
  - Declarative
  - Part of C#
- Method syntax
  - Extension methods
  - In System.Linq
- Semantically Equivalent

```
var query = from c in customerList
            where c.CustomerId == customerId
            select c;

var query = customerList.Where(c =>
            c.CustomerId == customerId);
```

Demo: LINQ Syntaxes

Required extensions to C#

- Anonymous types
  - Could be very difficult to specify the resulting type
  - So just use ‘var’
- Automatic properties
  - Select can project object and properties, so they need to be able to be implemented automatically (get/set)
- Lambda expressions
  - Use in the ‘where’ extension methods to evaluate objects
  - Could use func/action/delegates, but its too wordy
- Initializers
  - Data being projected may need to have collections assigned to them
- Expression trees
  - Represent the overall expression that will be lazy evaluated
- Extension methods
  - Need to iterate over existing data types for which we don’t have the code

Demo: Extensions to C# for LINQ

Standard Query Operators

- Defined in System.Linq
- Operate on IEnumerable<T>
- Implemented as extension methods

```
var query = customerList.Where(c =>
            c.CustomerId == customerId);
```

Language Integration

- Baked into the compiler

```
var query = from c in customerList
            where c.CustomerId == customerId
            select c;
```

- Compiler converts this to extension method chains

LINQ Query Operators

- **The same standard query operators work everywhere**
  - Objects
  - Relational data
  - XML data
- **Over 50 operators defined**
  - Filtering
  - Projection
  - Joining
  - Partitioning
  - Ordering
  - Aggregating
- **Similar to SQL**
  - Select, From, Where, OrderBy, GroupBy

Deferred Execution

- **Query expression does not execute until we access the result**
  - Treat query expressions as data
  - Allows us to build composited queries
- Extension syntax is just this way
  - Chained extension methods each returning IEnumerable<T>

```
public static class Enumerable
{
  /// <summary>
  /// Filters a sequence of values based on a predicate.
  /// </summary>
  ///
  /// <returns>
  /// An <see cref="T:System.Collections.Generic.IEnumerable`1"/> that contains elements from the input sequence that sa
  /// </returns>
  /// <param name="source">An <see cref="T:System.Collections.Generic.IEnumerable`1"/> to filter.</param><param name="pr
  public static IEnumerable<TSource> Where<TSource>(this IEnumerable<TSource> source, Func<TSource, bool> predicate);
```

IQueryable<T>

- Like IEnumerable<T>, but
  - Works with a data provider
  - Can evaluate the expression tree and make optimizations
  - Data not need be in memory

Extensibility

- **Operator extensibility**
  - We can implement our own operators (they are just extension methods)
  - We can override standard operators for our own types
- **Provider extensibility**
  - A LINQ Provider is a gateway to query-able types
- **PLINQ**
  - Parallel execution of line queries

Demo: Parallel LINQ

Data Sources

- LINQ to Objects
- LINQ to Entities
- LINQ to SQL
- LINQ to XML

- LINQ to anything you write a IQuery<T> provider

LINQ to SQL

```
IEnumerable<Customer> customers =
    from c in context.Customers
    where c.Country == "France"
    orderby c.CustomerID ascending
    select c;
```

```
SELECT [t0].[CustomerID], [t0].[CompanyName], [t0].[ContactName],
       [t0].[ContactTitle], [t0].[Address], [t0].[City],
       [t0].[Region], [t0].[PostalCode], [t0].[Country],
       [t0].[Phone], [t0].[Fax]
FROM [dbo].[Customers] AS [t0]
WHERE [t0].[Country] = @p0
ORDER BY [t0].[CustomerID]
```

.NET for C Programmers

LINQ Operators

Many classes of operators

- Filtering
- Projecting
- Ordering
- Grouping
- Conversions
- Sets
- Aggregation
- Quantifiers
- Generation
- Elements
- Joins

Filtering

Method	Description
.Where	Filters by predicate function
.OfType	Filters on ability to be coerced to a specific type

Sorting

Method	Description
OrderByOrderByDescending	Sort values in ascending or descending order (orderby)
ThenBy/ ThenByDescending	A secondary sort
Reverse	Reversethe order of elements

Sets

Method	Description
Distinct	Remove duplicate values
Except	Returns the differences of two sequences
Intersect	Returns the intersection of two sequences
Union	Returns unique elements from both sequences

Equality in LINQ to Objects

- **Operators that test equality use default IEqualityComparer**
  - Will accept a custom comparer
- **Anonymous types generated by C# compiler are special**
  - Override Equals and GetHashCode
  - Uses all public properties on type to test for equality

Quantifiers

Method	Description
All	Tests if all elements satisfy a condition
Any	Tests if any elements satisfy a condition
Contains	Tests if the sequence contains a specific element

Projection

Method	Description
Select	Projects values in a sequence based on a transformation function
SelectMany	Flattens and projects across multiple sequences

Partitioning

Method	Description
Skip/ SkipWhile	Skip elements until a condition or predicate is met
Take / TakeWhile	Take elements until a condition or predicate is met

Grouping

Method	Description
GroupBy	Group elements from a sequence
ToLookup	Insert elements into a one to many dictionary

Generation

Method	Description
Empty	Returns a empty collection
Range	Generates a sequence of numbers
Repeat	Generates a collection of repeated values
DefaultIfEmpty	Replaces empty collection with collection of 1 default value

Elements

Method	Description
ElementAt/ ElementAtOrDefault	Returns the element at a specified index
First / FirstOrDefault	Returns the first element of a collection
Last / LastOrDefault	Returns the last element of a collection
Single / SingleOrDefault	Returns a single element

Conversions

Method	Description
AsEnumerable	Returns input as IEnumerable<T>
AsQueryable	Converts IEnumerable<T>to IQueryable<T>
Cast	Coerce all elements to a type
OfType	Filters values that can be coerced to a type
ToArray	Converts sequence to an array (immediate)
ToDictionary	Convert sequence to Dictionary<K, V>
ToList	Converts sequence to List<T>
ToLookup	Group elements into an IGrouping<K, V>

Concatenation

Method	Description
Concat	Concatenates two sequences into a single sequence

Joins

Method	Description
Join	Equivalent to an INNER JOIN in SQL, returns a flat hierarchy
GroupJoin	Equivalent to LEFT JOIN in SQL, returns a hierarchy

Aggregation

Method	Description
Aggregate	Computes a custom aggregation on a sequence
Average	Calculates the average value in a sequence
Count	Counts the elements in a sequence
Max	Returns the maximum value in a sequence
Min	Returns the minimum value in a sequence
Sum	Calculates the sum of values in a sequence

.NET for C Programmers

TPL and Async

Overview

- Tasks
- Implementing a task
- Waiting for a task to complete
- Chaining tasks
- Async / await

What is a Task?

- Abstraction of an asynchronous unit of work
- Can be a thread, but my be simply async (like a web request)
- Has defined constructs for completion and continuation
- Feels a lot like a promise in JavaScript

Implementing a Task

- Common to use Task.Factory.StartNew

```
Task.Factory.StartNew(
    () =>
    {
        Console.WriteLine("Task running");
    });
Console.WriteLine("Started task");
```

Knowing when a Task completes

- .ContinueWith method called with the task completes

```
1 reference | 0 authors | 0 changes
private void ex2_completion()
{
    Task.Factory.StartNew(
        () =>
        {
            Console.WriteLine("Task running");
            Task.Delay(2000).Wait();
            Console.WriteLine("Task ending");
        })
        .ContinueWith(a => Console.Write("Task is complete"));
    Console.WriteLine("Started task");
}
```

Cancellation

Async / await

- TPL gets implemented in the compiler
- An async method is run as a Task
- await tells the code to continue the next line as the Task's .ContinueWith
- Collapses chained .ContinueWith's into a simple form of code

Example

- Methods tagged with ‘async’ are scheduled by the compiler as tasks
- They have a special return value that includes Task (void) or Task<T>
- You await an async method
- Any method with await must also be tagged as async

```
0 references | 0 authors | 0 changes
async private void doAsyncAwait()
{
    var result = await doSomething();
    Console.WriteLine("Notice this is after
    Console.WriteLine(result);
}

2 references | 0 authors | 0 changes
private async Task<string> doSomething()
{
    Console.WriteLine("Do something running"
    Task.Delay(5000).Wait();
    return "HI";
}
```

This is the same as

```
2 references | 0 authors | 0 changes
private async Task<string> doSomething()
{
    Console.WriteLine("Do something running");
    Task.Delay(5000).Wait();
    return "HI";
}

1 reference | 0 authors | 0 changes
private void ex3_equivalent()
{
    var t = doSomething();
    t.ContinueWith(
        a =>
        {
            Console.WriteLine("Notice this is after the other method returns");
            Console.WriteLine(a.Result);
        });
    t.Wait();
}
```

Demo: Tasks and Async/Await

.NET for C Programmers

Unit Testing with xUnit.NET

Overview of the module

- Overview of TDD
- Intro to xUnit.net
- Writing tests with xUnit.NET
- Data-Driven Tests (DDT)
- Automatic Data
- Moq’s

What is TDD?

- Code is written in small pieces and then a test immediately written
- The test is written to check all possible inputs and outputs for correctness
- Repeat

- In theory: your write bug free code

Benefits of TDD

- Happier development team
  - Fewer late nights/weekend work
  - More time to add new features
- Happier users
  - Fewer defects reaching production causing annoyance
- Reduced business cost
  - Defects found earlier in development lifecycle
- Reliability
  - Exactly same test code runs each time
  - No variance between runs from Human error
- Faster execution
  - Quicker than a human performing tests manually

What is a Unit Test?

- Testing of one or more methods in your code
- Another piece of code written to test “correctness” of the method
- One or more tests attempt to “cover” all scenarios in the method
- Normally automated

What Makes a Good Test

- Independent & isolated
- Test single behavior / logical thing
- Clear intent / readable
- Don't test the compiler
- Reliable & repeatable
- Production quality code
- Valuable

What is xUnit.net?

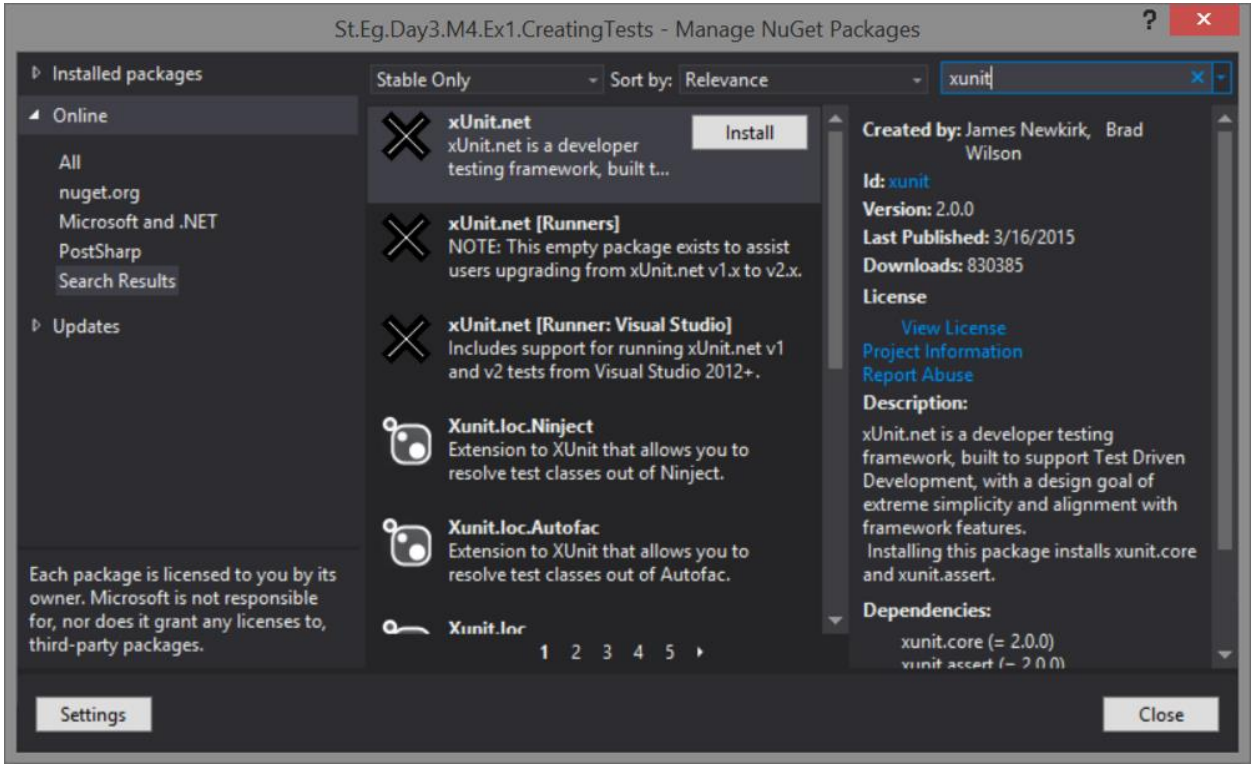
- A unit testing tool for .NET
- Created by the inventory of Nunit
- Open source and free
- http://xunit.github.io/

Why xUnit.Net over Nunit?

- Addresses more testing patterns than Nunit
- Closer to .NET (Nunit tries to be like java version)
- Other:
  - Easier setup of tests
  - Less attributes
  - Extensibility

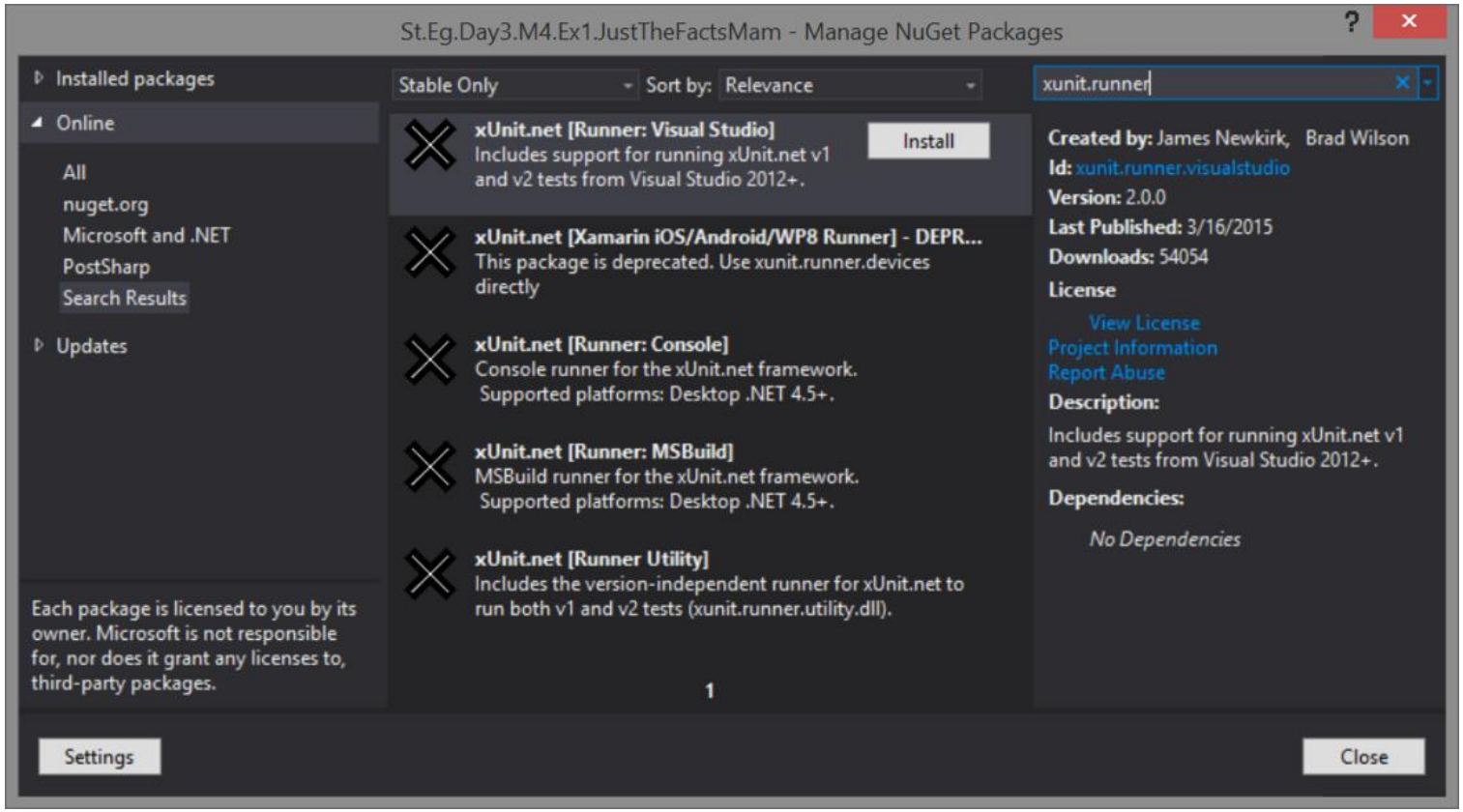
Adding xUnit.Net to your project

- Add the libraries using Nuget
- xUnit.NET is the minimum you need



Also, add the test runner

- The test runner is also on Nuget, and needs to be added
- http://xunit.github.io/docs/running-tests-in-vs.html



Defining a simple test

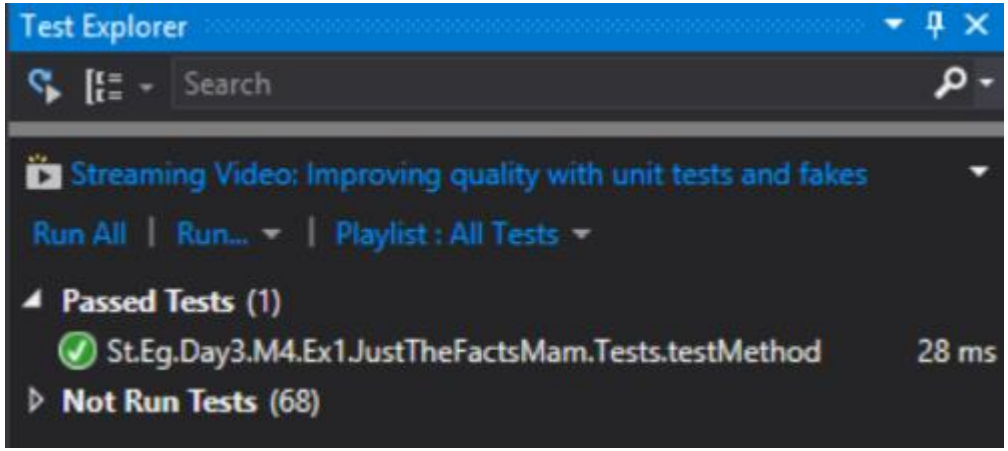
- Create a class
- Add testing methods
- Attribute those methods
- Assert a result

```
[Fact]
| 0 references | 0 authors | 0 changes
public void testMethodAlwaysPasses()
{
    Assert.True(true);
}
```

- A test that always passes:

Test explorer show your results

- Shows pass/fail
- Run time
- Lets you select tests to run



Demo: Writing an xUnit.net Test from Scratch

- Create new project (normally a library)
- Reference production code project
- Install xUnit.net NuGet package
- Write a test class and methods
- Build, tests will show in the test explorer
- Run!

xUnit.net Test Attributes

- Inform xUnit.net how you want the test method to be run by using an attribute

Attribute	Description
Fact	Test method does not take parameters
Theory	Test method takes parameters (data-driven)
Trait	Assigns metadata to a test method

Asserts

- Check conditions and throw exceptions
- Exception thrown means the test failed

Assert.
Equal
NotEqual
NotSame
Same
Contains
DoesNotContain
DoesNotThrow
InRange
…

Theories

- Facts are tests which are always true. They test invariant conditions.
- Theories are tests which are only true for a particular set of data.
- With a Theory, you specify data to be passed to the test

Demo: Theories

Inline data for Theories

- Theories can have attributes to supply data
- Test is run once for each InlineData attribute

```
[Theory]
[InlineData(3)]
[InlineData(5)]
[InlineData(6)]
0 references | 0 authors | 0 changes
public void MyFirstTheoryInline(int value)
{
    Debug.Listeners.Add(new DefaultTraceListener());
    Console.WriteLine(value);
    Assert.True(IsOdd(value));
}
```

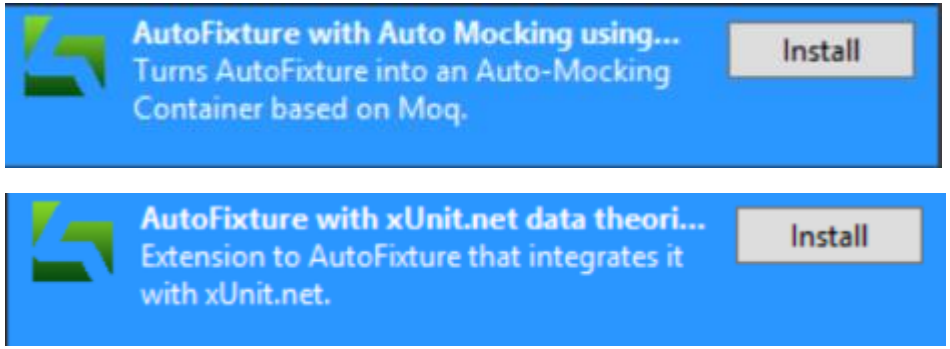
Capturing output from tests

```
public class TheoryTests
{
    private readonly ITestOutputHelper output;

    0 references | 0 authors | 0 changes
    public TheoryTests(ITestOutputHelper output)
    {
        this.output = output;
    }

    [InlineData(3)]
    [InlineData(5)]
    [InlineData(6)]
    [Theory]
    0 references | 0 authors | 0 changes
    public void MyFirstTheoryInline(int value)
    {
        output.WriteLine("{0}", value);
        Assert.True(IsOdd(value));
    }
```

Demo: Inline Theory Data

AutoData tests

- Allows the automatic creation of data for tests
- This can simplify test setup and make them more reusable



Moq’s

- What if you don’t want to provide a real implementation in the test?
  - Why would you want to do that?
  - Possibly you are testing a database and don’t‘ have a connection
- Also, we often want to test that methods were actually called in the test, and in the correct order

Demo: Moq’d object tests

Advanced

- dotCover
- NCrunch

Ncover / dotCover

NCover Source View

c:\Users\Russell\Documents\Visual Studio 2010\Projects\Calc\CalcTest\CalcTest.cs

```
6
7   namespace CalcTest
8   {
9       [TestFixture]
10      public class CalcTest
11      {
12          [Test]
13          public void TestAddition()
14          {
15              var result = Calculator.CalcController.CalcController.Addition(1, 1);
16              Assert.AreEqual(2, result);
17          }
18
19          [Test]
20          public void TestSubtraction()
21          {
22              var result = Calculator.CalcController.CalcController.Subtraction(2, 2);
23              Assert.AreEqual(0, result);
24          }
25
26          [Test]
27          public void TestMultiplication()
28          {
29              var result = Calculator.CalcController.CalcController.Multiplication(3, 3);
30              Assert.AreEqual(8, result);
31          }
```

Uncovered | Visited Count: 0

NCrunch

Common.Models.ActionModel | ActionModel(TaskModel taskModel)

```
        public bool CanIncrementIndex
        {
            get
            {
                return _index < Parent.Actions.Max(x => x.Index);
            }
        }

        public void IncrementIndex()
        {
            if (!CanIncrementIndex) return;

            ActionModel nextAction = Parent.Actions.FirstOrDefault(h => h.Index == (_index + 1));
            if (nextAction == null) throw new ApplicationException("Invalid indexing of actions.

            nextAction.Index = _index;
            _index = _index + 1;
        }

        public bool CanDecrementIndex
        {
            get { return _index > 0; }
        }
```

These arrows indicate that the lines are covered by tests

.NET for C Programmers

Intro to CodedUI

What is CodedUI

- Automated user interface testing
- A framework from Microsoft for testing multiple platform GUIs
  - WPF
  - Winforms
  - Web,
  - Other…
- Records interaction and replays the tests in an automated manner

Demo: CodedUI